MANAGEMENT OF SOFTWARE SYSTEMS

Module 3

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

February 11, 2023

Outline

Object-oriented design using uml

- Object-oriented design processes involve designing object classes and the relationships between these classes.
- To develop a system design from concept to detailed, object-oriented design, you need to(Common activites in these process):
  1. Understand and define the context and the external interactions with the system.
  2. Design the system architecture.
  3. Identify the principal objects in the system.
  4. Develop design models.
  5. Specify interfaces

System context and interactions

- The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment.
- System context models and interaction models present complementary views of the relationships between a system and its environment:
  - **A system context model** is a structural model that demonstrates the other systems in the environment of the system being developed.
  - **An interaction model** is a dynamic model that shows how the system interacts with its environment as it is used.

System context and interactions

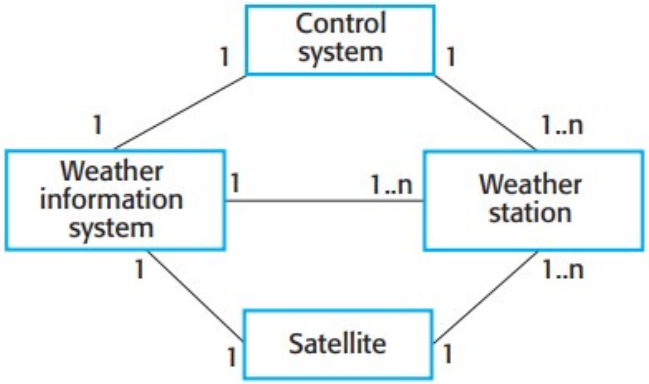


Figure 1: System context for the weather station

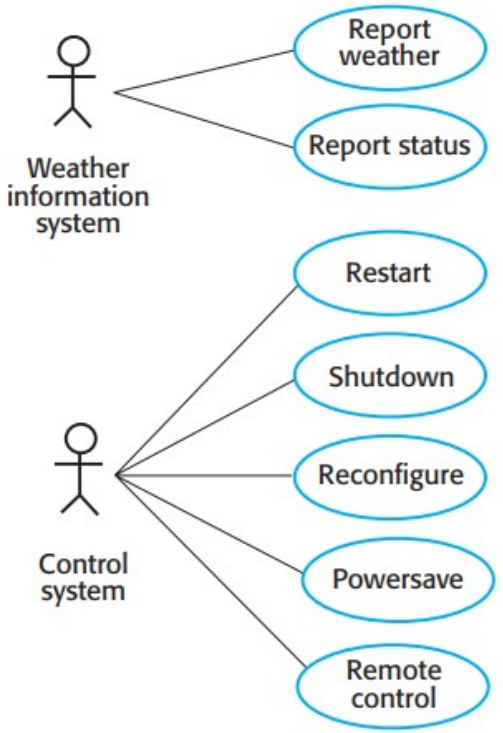


Figure 2: Weather station use cases

Object-oriented design using the UML

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	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 5-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 future.

Figure 3: Use case description—Report weathe

Architectural Design:

- Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture.
- **Identify the major components** that make up the system and their interactions.
- Then design the system organization using an architectural pattern such as a layered or client–server model.

Object-oriented design using the UML

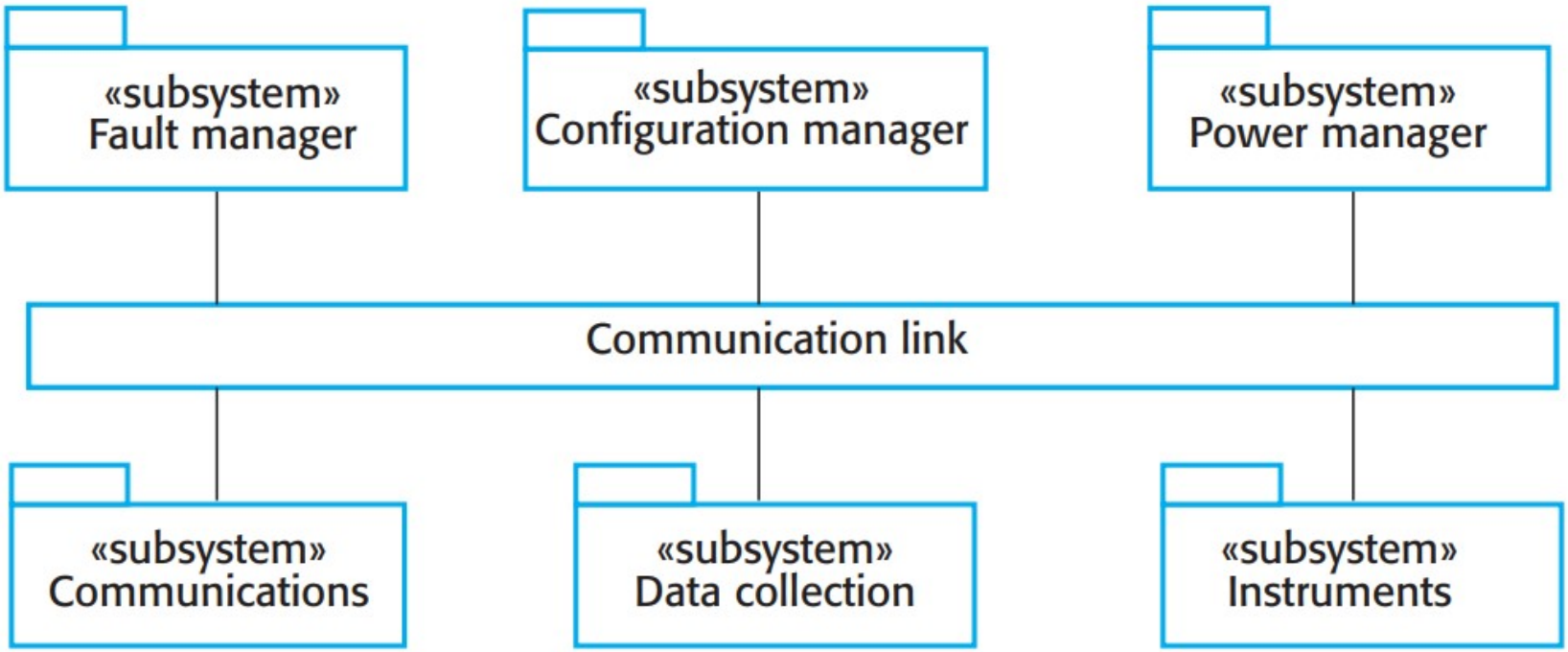


Figure 4: High-level architecture of weather station

Architectural Design Cont..

- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as Communication link in the figure.
- Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.
- The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

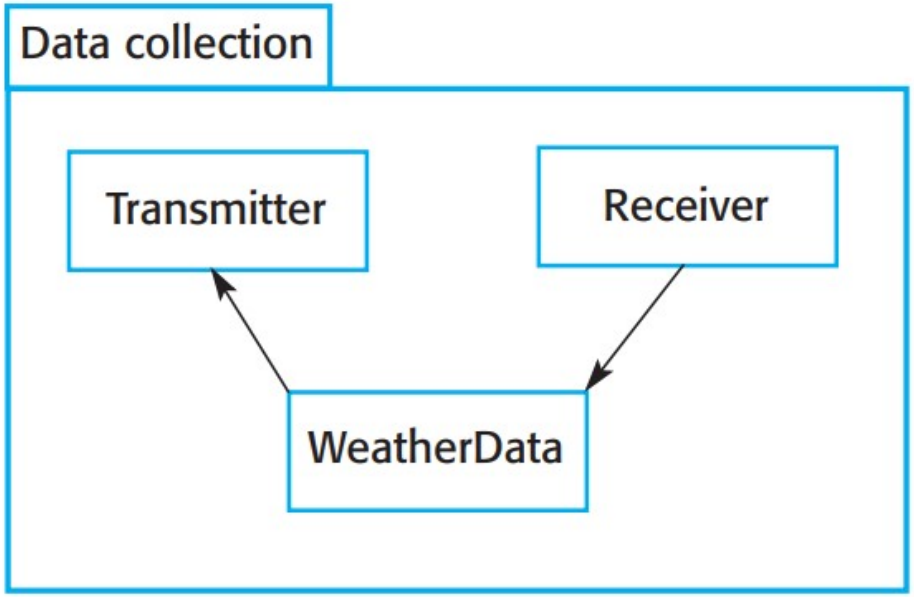


Figure 5: Architecture of data collection system

Object class identification

- 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.
- Various proposals are made to identify object classes in object oriented systems
  - Use a **grammatical analysis approach** based on a natural language description of the system.
    - Eg:Objects and attributes are nouns; operations or services are verbs
  - Use **tangible entities (things)** in the application domain such as aircraft, roles such as manager or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units.
  - Use a **scenario-based analysis** where various scenarios of system use are identified and analysed and identify the required objects, attributes, and operations

Object class identification

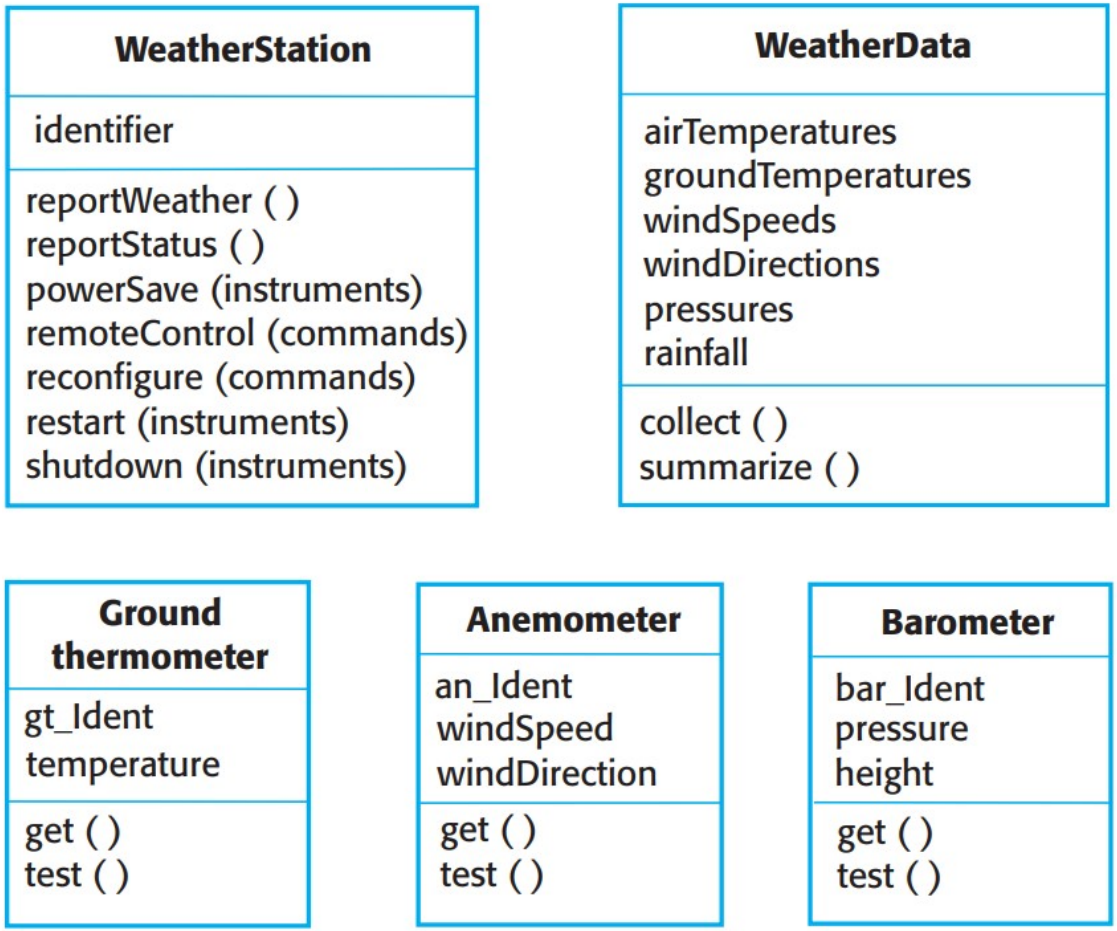


Figure 6: Weather station objects

Design models

- **Develop Design models** which **show the objects and object classes and relationships between these entities**.
- When the UML is used to develop a design, two kinds of design models are normally developed:
  - **Structural models**, which describe the static structure of the system using object classes and their relationships.
  - **Dynamic models,** which describe the dynamic structure of the system and show the interactions between the system objects.
- Examples of design models
  - **Subsystem models** that show logical groupings of objects into coherent subsystems.
  - **Sequence models** that show the sequence of object interactions.
  - **State machine models** that show how individual objects change their state in response to events.
  - Other models include **use-case models aggregation models, generalisation models**, etc. ,

Object-oriented design using the UML

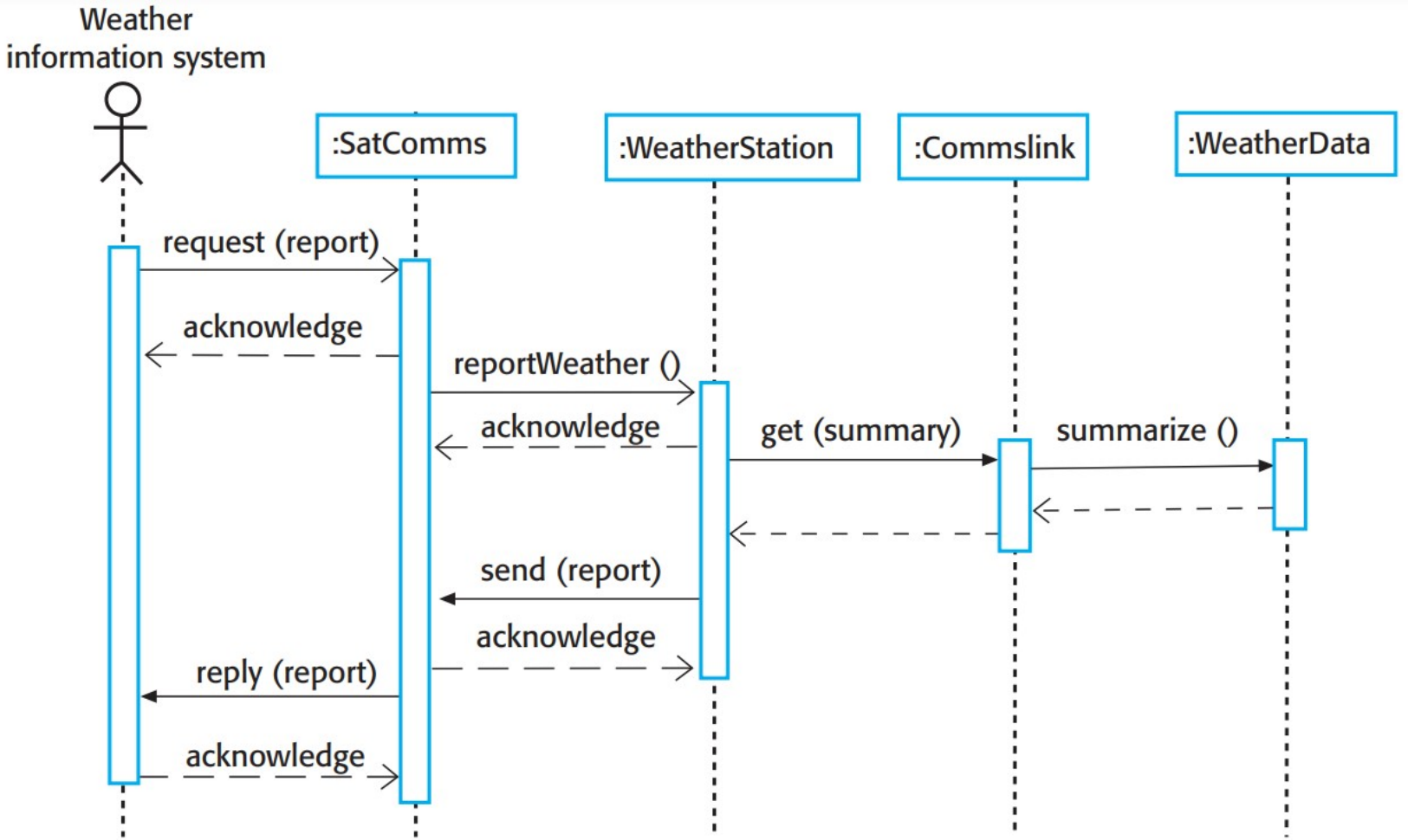


Figure 7: Sequence diagram describing data collection

Object-oriented design using the UML

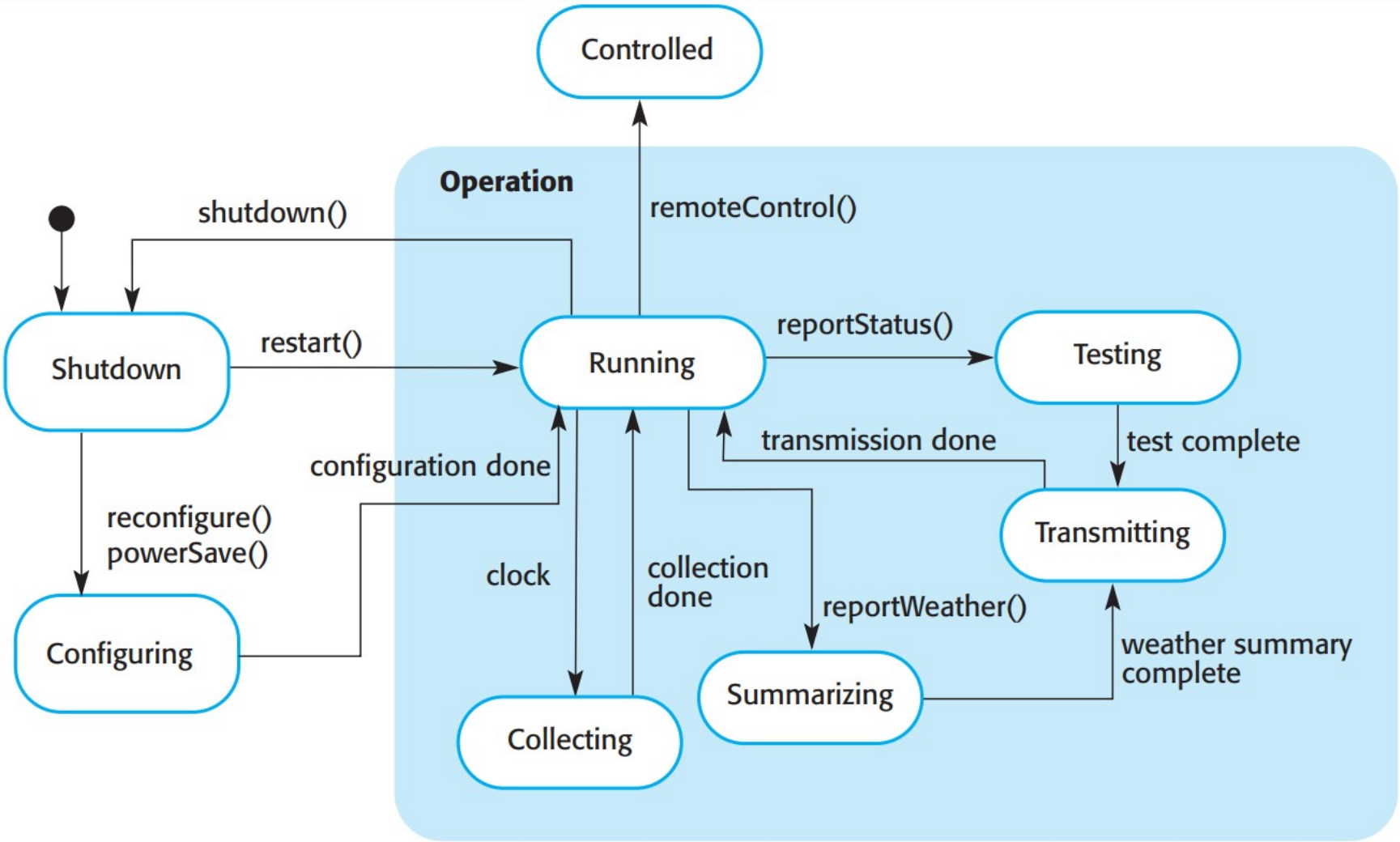


Figure 8: Weather station state diagram

Interface specification

- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects
- Interfaces can be specified in the UML using the same notation as a class diagram. There is no attribute section and the UML stereotype << *interface* >> should be included in the name part.

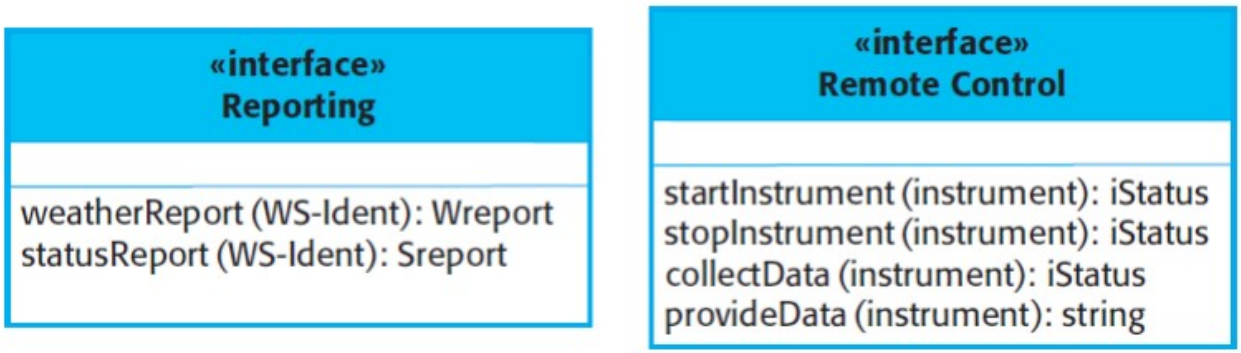


Figure 9: Weather station interfaces

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**.
- The four essential elements of design patterns
  - **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

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.
The UML model of the pattern is shown in Figure 7.12.
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.

Figure 10: The Observer pattern description.

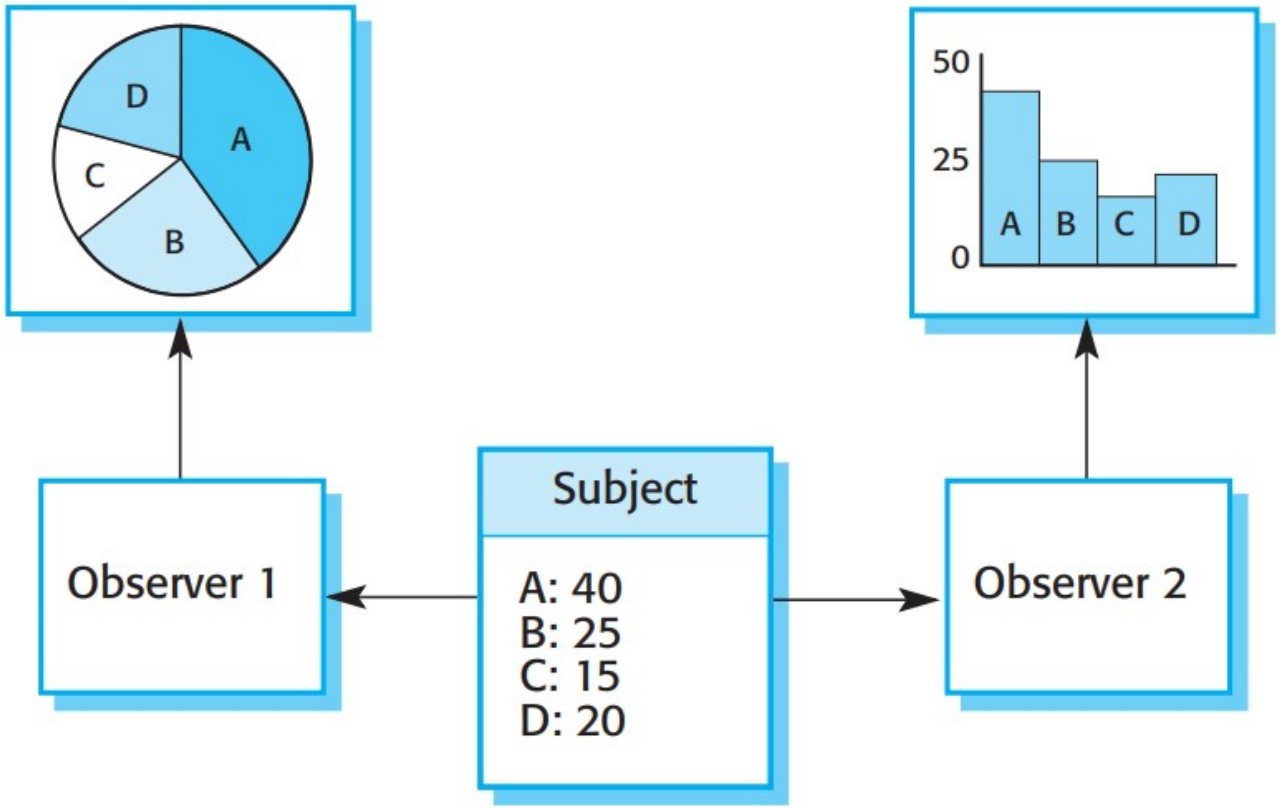


Figure 11: The Observer pattern Graphical representation

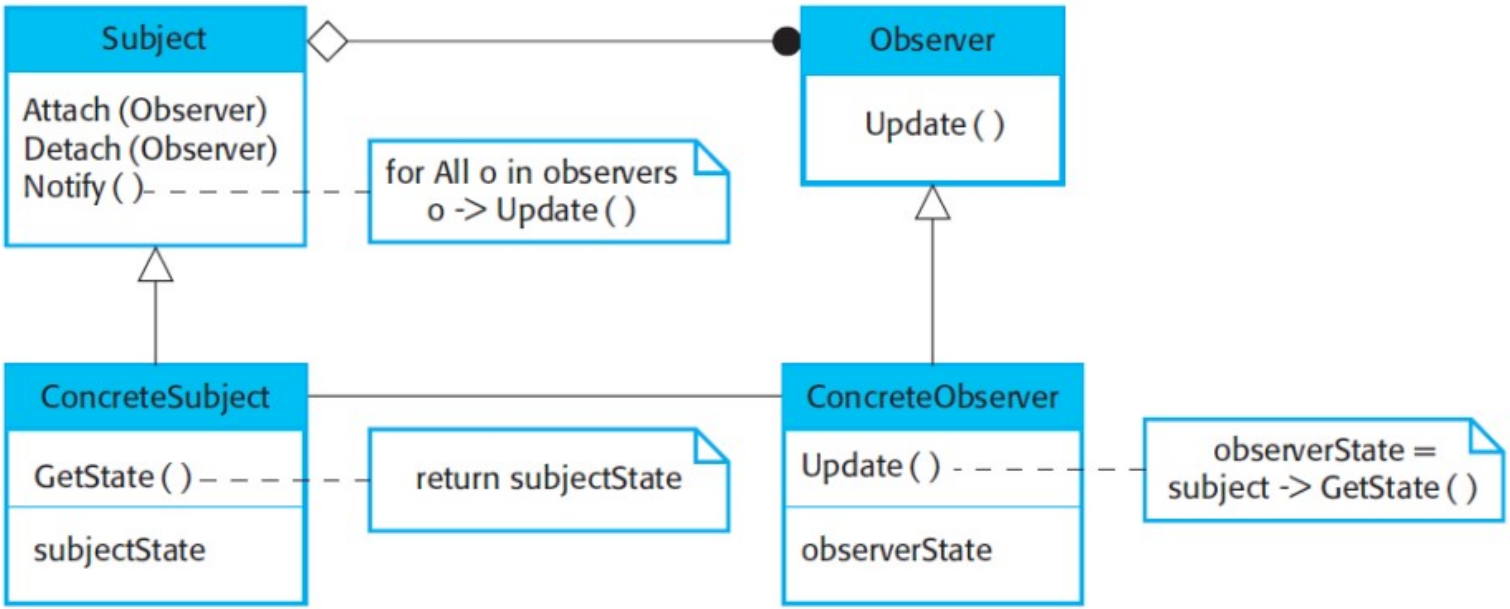


Figure 12: A UML model of the Observer pattern

Design Patterns

- There are four essential description elements and also include a brief statement of what the pattern can do.
- This pattern can be used in situations where different presentations of an object's state are required.
- Graphical representations are normally used to illustrate the object classes in patterns and their relationships.
- These supplement the pattern description and add detail to the solution description.

Implementation issues

- Implementation is a critical stage where an executable version of the software is developed.
- Some of the aspects of implementation are:
  1. Reuse
  2. Configuration management
  3. Host-target development

Implementation issues

1 **Reuse**

- Most modern software is constructed by reusing existing components or systems.
- When developing software, make use of existing code as much as possible.
- Software reuse is possible at a number of different 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.

Implementation issues

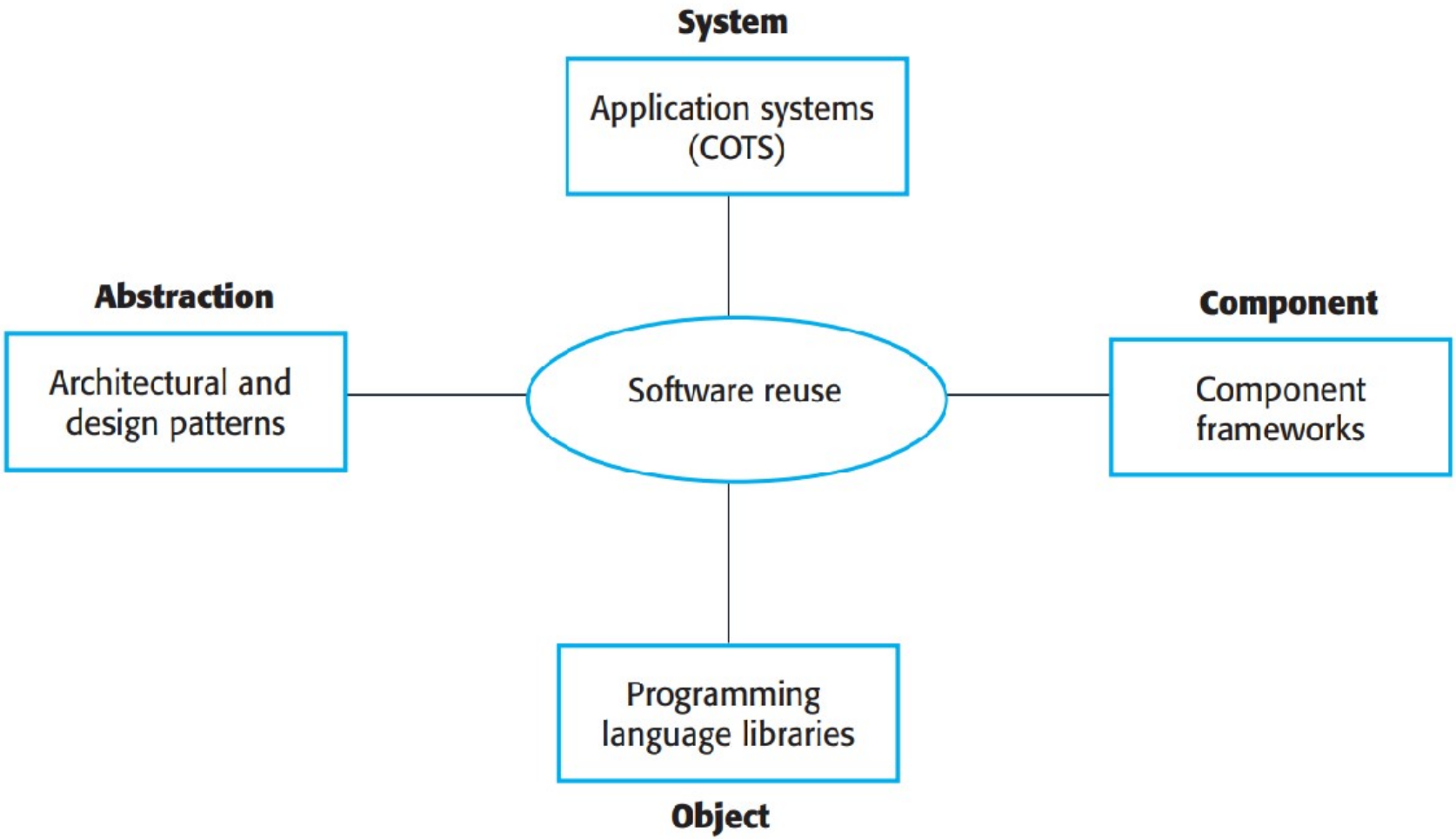


Figure 13: Software reuse

Implementation issues

Costs of reuse:

- 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.

2 **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.

Implementation issues

There are three fundamental configuration management activities:

- **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.

Implementation issues

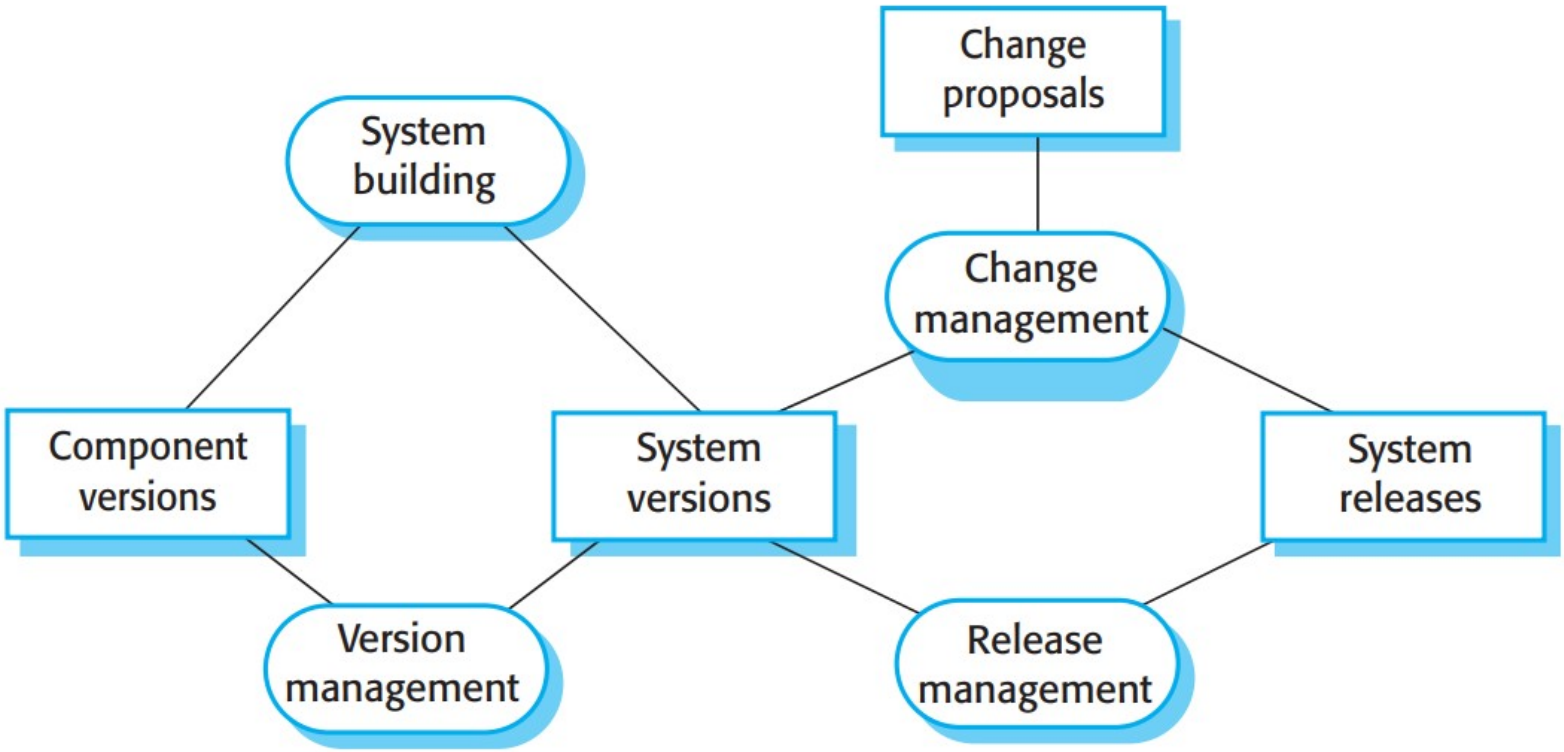


Figure 14: Configuration management

3 **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.

Implementation issues

A software development platform should provide a range of tools to support software engineering processes. These may 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.
- Tools to support refactoring and program visualization
- Configuration management tools to manage source code versions and to integrate and build systems.

Implementation issues

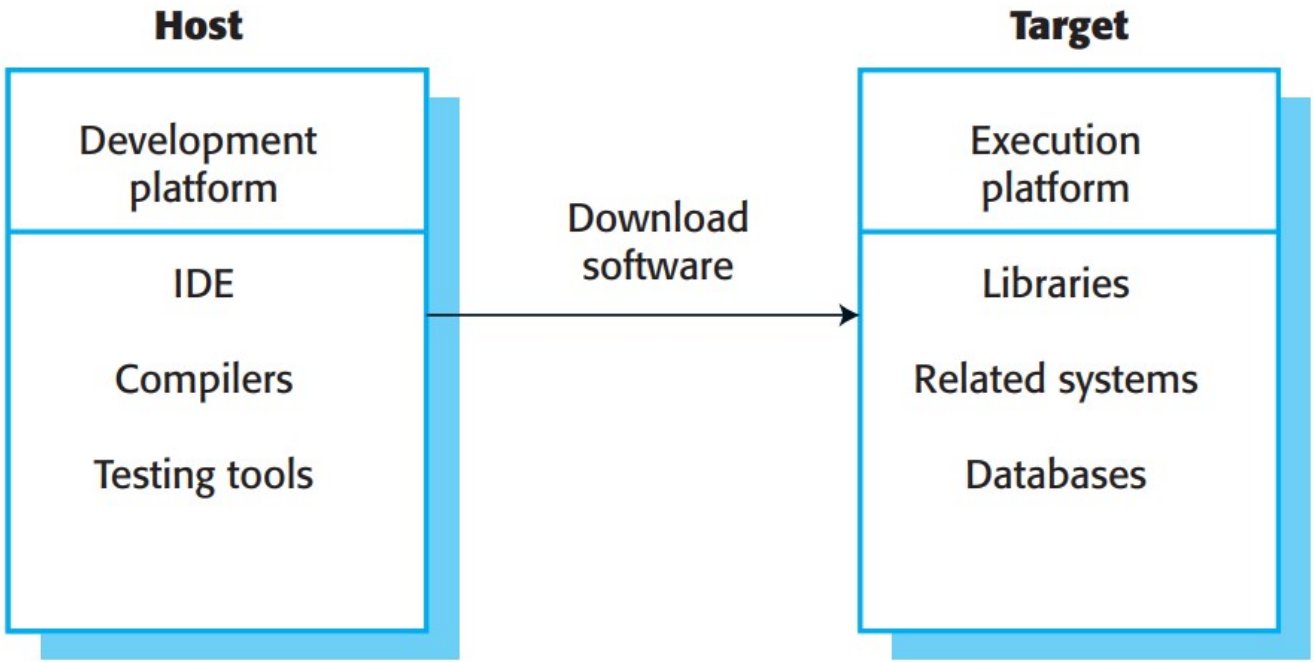


Figure 15: Host-target development

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 (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- 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.
- 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.

Open source development

Open source development: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 proprietar y systems that are sold.

Review Techniques

- Software reviews are a "filter" for the software process.
- **Reviews are applied at various points during software engineering and serve to uncover errors and defects that can then be removed.**
- Software reviews "purify" software engineering work products, including requirements and design models, code, and testing data.
- A review—any review—is a way of using the diversity of a group of people to
  - **Point out needed improvements** in the product of a single person or team;
  - **Confirm those parts** of a product in which **improvement is either not desired or not needed;**

Cost impact of Software Defects

- Within the context of the software process, the terms defect and fault are synonymous.
- Both imply a quality problem that is discovered after the software has been released to end users.
- **The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software.**
- The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.
- **Review techniques have been shown to be up to 75 percent effective in uncovering design flaws.**
- By **detecting and removing** a large percentage of these errors, the review process substantially **reduces the cost of subsequent activities in the software process.**

DEFECT AMPLIFICATION AND REMOVAL

- A defect amplification model can be used to **illustrate the generation and detection of errors during the design and code generation actions of a software process**.
- To conduct reviews, you must expend time and effort, and your development organization must spend money

Review Techniques

- A box represents a software engineering action. During the action, errors may be inadvertently generated.
- Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through.
- In some cases, errors passed through from previous steps are amplified (amplification factor, x) by current work.
- The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

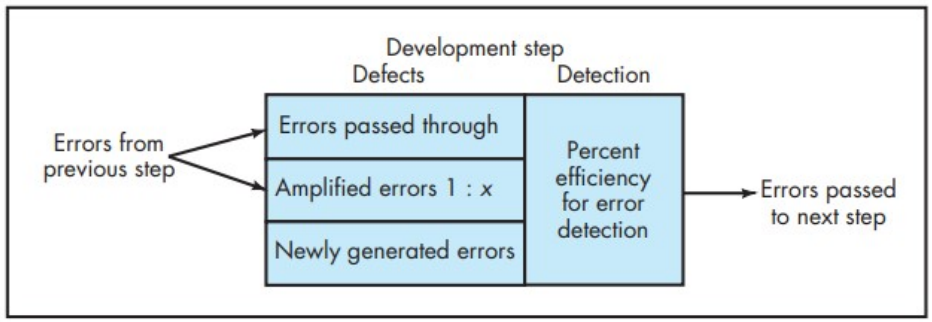


Figure 16: Defect amplifi cation model

Review Techniques

- Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption).
- Ten preliminary design defects are amplified to 94 errors before testing commences.
- Twelve latent errors are released to the field

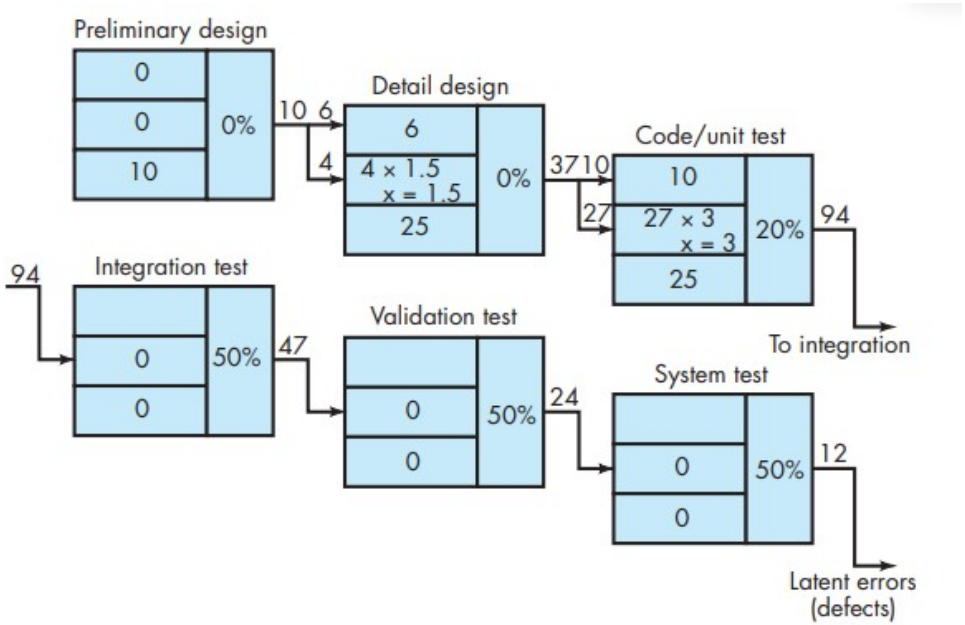


Figure 17: Defect amplifi cation— no reviews

Review Techniques

- In this case, 10 initial preliminary (architectural) design errors are amplified to 24 errors before testing commences.
- Only three latent errors exist.
- The relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established.

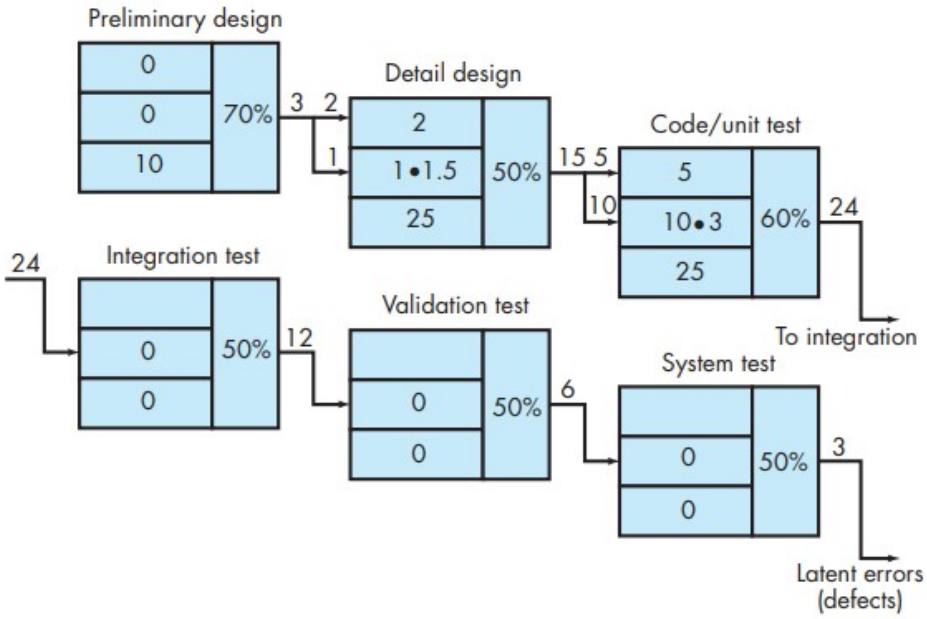


Figure 18: Defect amplifi cation— reviews conducted

Review Techniques

- Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units.
- When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

REVIEW METRICS AND THEIR USE

- Technical reviews are one of many actions that are required as part of good software engineering practice.
- Each action requires dedicated human effort.
- Since available project effort is finite, it is important for a software engineering organization to understand the effectiveness of each action by defining a set of metrics that can be used to assess their efficacy.
- Although many metrics can be defined for technical reviews, a relatively small subset can provide useful insight.

Review Techniques

The following review metrics can be collected for each review that is conducted:

- **Preparation effort,** E_p —the effort (in person-hours) required to review a work product prior to the actual review meeting
- **Assessment effort,** E_a — the effort (in person-hours) that is expended during the actual review
- **Rework effort,** E_r — the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- **Work product size, WPS** —a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- **Minor errors found,** Err_{minor} —the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- **Major errors found,** Err_{major} —the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

ANALYZING MATRICES

- The total review effort and the total number of errors discovered are defi ned as:

$$Ereview = Ep + Ea + Er$$

$$Errtot = Errminor + Errmajor$$

- Error density represents the errors found per unit of work product reviewed.

$$Errordensity = Errtot/WPS$$

Cost-Effectiveness of Reviews

- It is difficult to measure the cost-effectiveness of any technical review in real time.
- A software engineering organization can assess the effectiveness of reviews and their cost benefit only after reviews have been completed, **review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured**

$$Effortsavedpererror = Etesting - Ereviews$$

Effort expended with and without reviews

- Effort expended when reviews are used does increase early in the development of a software increment.
- Testing and corrective effort is reduced.
- The deployment date for development with reviews is sooner than the deployment date without reviews.
- Reviews don't take time, they save it.

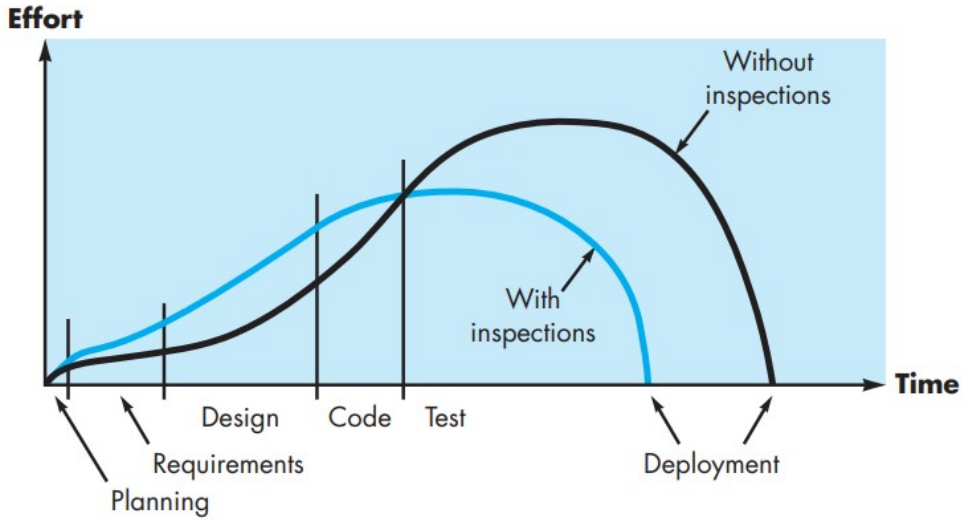
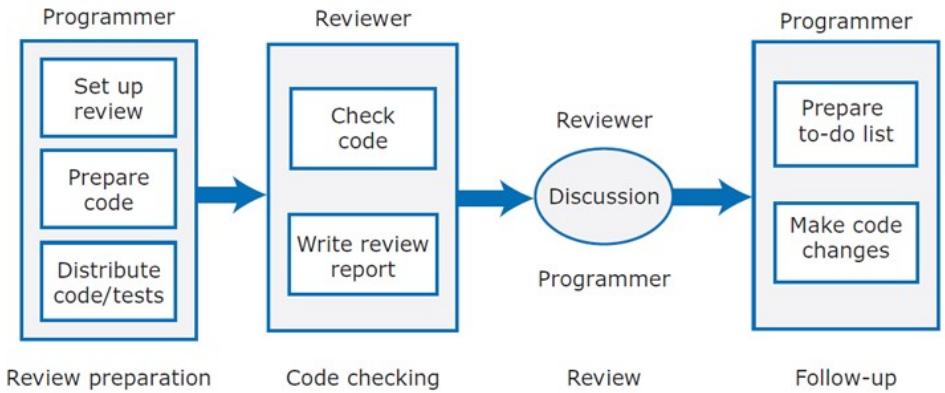


Figure 19: Effort expended with and without reviews

CODE REVIEWS

- **Code reviews that focus on security issues should be included as part of the implementation activities.**
- These code reviews should be based on the appropriate security objectives and threats identified in the system design activities.
- Code reviews are known to **reduce the number of product defects prior to testing**, which in turn **eliminates potential security holes and improves software quality**.



Code review activities

Activity	Description
Set up review	The programmer contacts a reviewer and arranges a review date.
Prepare code	The programmer collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.
Distribute code/tests	The programmer sends code and tests to the reviewer.
Check code	The reviewer systematically checks the code and tests against their understanding of what they are supposed to do.
Write review report	The reviewer annotates the code and tests with a report of the issues to be discussed at the review meeting.
Discussion	The reviewer and programmer discuss the issues and agree on the actions to resolve these.
Make to-do list	The programmer documents the outcome of the review as a to-do list and shares this with the reviewer.
Make code changes	The programmer modifies the code and tests to address the issues raised in the review.

REVIEWS : A FORMALITY SPECTRUM

- Technical reviews should be applied with a level of formality that is appropriate for the product to be built, the project time line, and the people who are doing the work.
- The formality of a review increases when
  - Distinct roles are explicitly defined for the reviewers,
  - There is a sufficient amount of planning and preparation for the review,
  - A distinct structure for the review (including tasks and internal work products) is defined
  - A set of specific tasks would be conducted based on an agenda that was developed before the review occurred.
  - The results of the review would be formally recorded, and the team would decide on the status of the work product based on the outcome of the review.
  - Members of the review team might also verify that the corrections made were done properly

REVIEWS : A FORMALITY SPECTRUM cont..

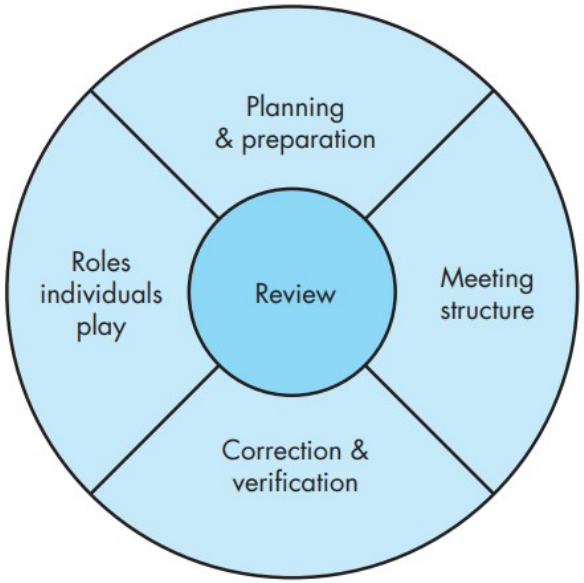


Figure 20: Reference model for technical reviews

Review Techniques

Technical reviews:

- Informal reviews
- Formal technical reviews.

Review Techniques

Informal reviews:

- Informal reviews **include a simple desk check of a software engineering work product with a colleague, a casual meeting** (involving more than two people) for the purpose of reviewing a work product.
- One way to **improve the efficacy of a desk check review** is to **develop a set of simple review checklists for each major work product** produced by the software team.
- Checklist for interfaces
  - Is the layout designed using standard conventions? Left to right? Top to bottom?
  - Does the presentation need to be scrolled?
  - Are color and placement, typeface, and size used effectively?
  - Are all navigation options or functions represented at the same level of abstraction?
  - Are all navigation choices clearly labeled?

Informal reviews Cont...

- Any errors or issues noted by the reviewers are recorded by the designer for resolution at a later time
- **Pair programming** can be characterized as a continuous desk check. Rather than scheduling a review at some point in time
- Pair programming encourages continuous review as a work product (design or code) is created.
- The benefit is immediate discovery of errors and better work product quality as a consequence.

FORMAL TECHNICAL REVIEWS

- A formal technical review (FTR) is a **software quality control activity performed by software engineers** (and others).
- The objectives of an FTR are:
  - **To uncover errors in function, logic, or implementation** for any representation of the software;
  - **To verify that the software** under review **meets its requirements;**
  - **To ensure that the software has been represented according to predefined standards;**
  - **To achieve software that is developed in a uniform manner;** and
  - To **make projects more manageable**

FORMAL TECHNICAL REVIEWS Cont..

- FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.
- The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.
- The FTR is actually a class of reviews that includes walkthroughs and inspections.
- Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

FORMAL TECHNICAL REVIEWS

- The Review Meeting
- Review Reporting and Record Keeping
- Review Guidelines

[1].**The Review Meeting**

- Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:
  - Between **three and five people** (typically) should be **involved** in the review.
  - **Advance preparation should occur** but should require no more than two hours of work for each person.
  - The duration of the review meeting should be **less than two hours.**
- The **focus** of the FTR is **on a work product** (e.g., a portion of a requirements model, a detailed component design, source code for a component).

The Review Meeting cont..

- The individual who has developed the work product—**the producer— informs the project leader that the work product is complete and that a review is required.**
- Each reviewer is **expected to spend between one and two hours reviewing the product, making notes.**
- The review meeting is **attended by the review leader, all reviewers, and the producer**
- **One of the reviewers takes on the role of a recorder,** that is, the individual who records (in writing) all important issues raised during the review.

The Review Meeting cont..

- The FTR **begins with an introduction of the agenda and a brief introduction by the producer.**
- The producer **then proceeds to "walk through"** the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each.
- **At the end** of the review, all attendees of the FTR must decide whether to:
  - **accept** the product **without further modification,**
  - **reject the product** due to severe errors (once corrected, another review must be performed), or
  - **accept the product provisionally** (minor errors have been encountered and must be corrected, but no additional review will be required).
- After the decision is made, all FTR **attendees complete a sign-off, indicating their participation** in the review and their concurrence with the review team's findings.

[2]. Review Reporting and Record Keeping

- During the FTR, a **reviewer (the recorder) actively records all issues that have been raised.**
- These are **summarized at the end** of the review meeting, and a review **issues list is produced**
- In addition, a formal technical review summary report is completed.
- A review **summary report answers three questions:**
  - What was reviewed?
  - Who reviewed it?
  - What were the findings and conclusions?

Review Reporting and Record Keeping cont..

- The review issues list serves two purposes:
  - to **identify problem areas** within the product and
  - to serve as an action item **checklist that guides the producer as corrections are made.**
- An issues list is normally attached to the summary report.

[3.] Review Guidelines

- Minimum set of guidelines for formal technical reviews:
  1. Review the product, not the producer
  2. Set an agenda and maintain it.
  3. Limit debate and rebuttal.
  4. Enunciate problem areas, but don't attempt to solve every problem noted..
  5. Take written notes.
  6. Limit the number of participants and insist upon advance preparation.
  7. Develop a checklist for each product that is likely to be reviewed.
  8. Allocate resources and schedule time for FTRs..
  9. Conduct meaningful training for all reviewers.
  10. Review your early reviews

POST-MORTEM EVALUATIONS(PME)

- Post-mortem evaluation (PME) as a **mechanism to determine what went right and what went wrong** when software engineering process and practice are applied in a specific project.
- Unlike an FTR that focuses on a specific work product, a **PME examines** the entire software project, focusing on both **" excellences and challenges"** a PME is attended by members of the software team and stakeholders.
- The intent is to **identify excellences and challenges** and to **extract lessons learned from both**.
- The objective is to **suggest improvements to both process and practice** going forward

Software testing strategies

- Testing is a set of activities that can be planned in advance and conducted systematically.
- Any testing strategy must incorporate,
  - Test planning
  - Test-case design
  - Test execution
  - Resultant data collection and evaluation.

Software testing strategies

Characteristics of software testing strategies

- **To perform effective testing**, you should **conduct effective technical reviews** . By doing this, many errors will be eliminated before testing commences.
- Testing **begins at the component level** and **works "outward" toward the integration of the entire computer-based system**.
- **Different testing techniques** are appropriate **for different software engineering approaches** and **at different points in time.**
- Testing is **conducted by the developer** of the software **and** (for large projects) **an independent test group.**
- **Testing and debugging are different activities**, but debugging must be accommodated in any testing strategy.

Verification and Validation

- **Verification** refers to the set of tasks that **ensure that software correctly implements a specific function.**
- **Validation** refers to a different set of tasks that **ensure that the software that has been built is traceable to customer requirements.**
- Boehm states this another way:
  - **Verification:** *"Are we building the product right?"*
  - **Validation:** *"Are we building the right product?"*

Software testing strategies

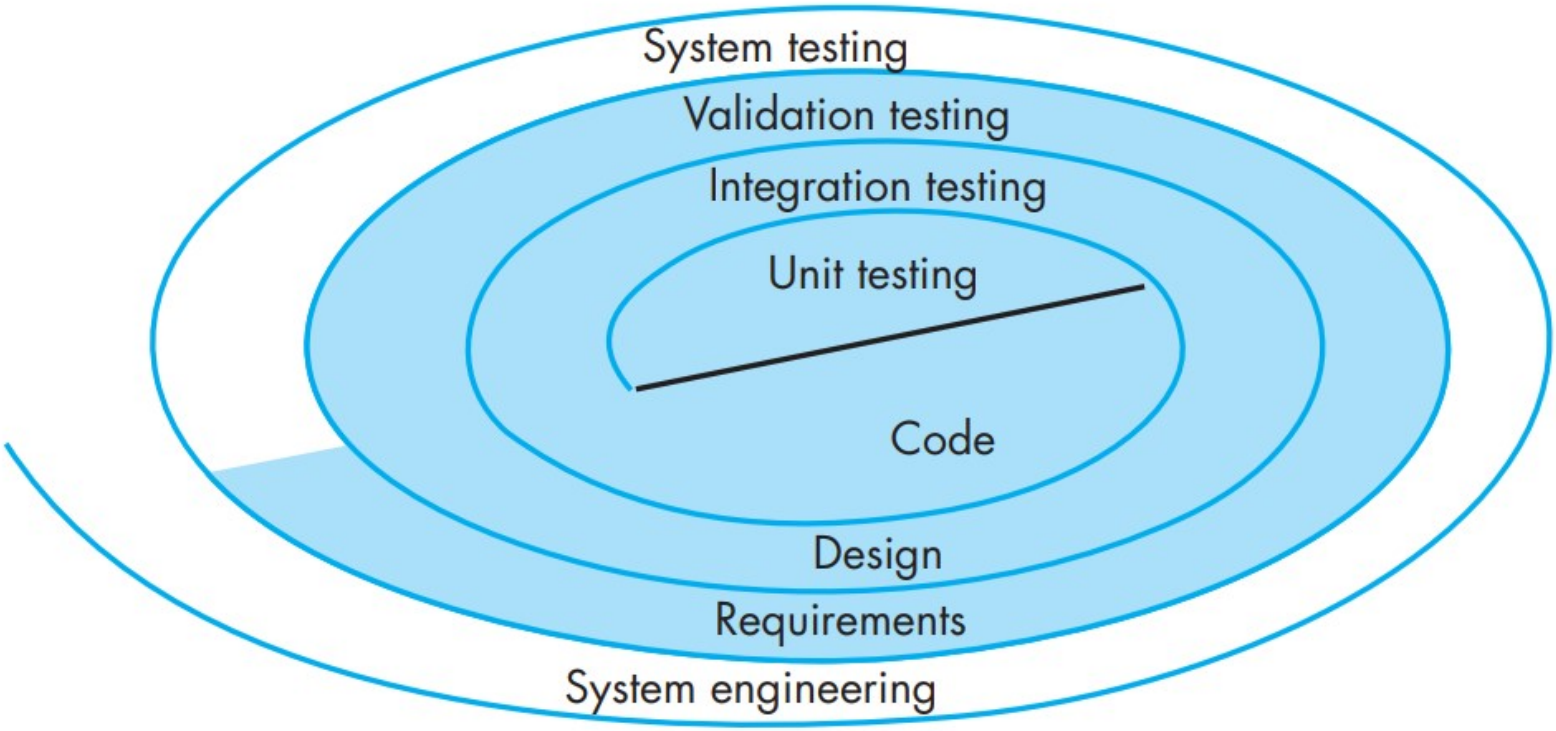


Figure 21: Testing strategy

Software testing strategies

1. UNIT TESTING
   - Unit Test Considerations
2. INTEGRATION TESTING
   i. Top-Down Integration
   ii. Bottom-Up Integration
   iii. Regression Testing
   iv. Smoke Testing
3. VALIDATION TESTING
4. SYSTEM TESTING
5. DEBUGGING

UNIT TESTING

- Unit testing focuses **verification** effort on the **smallest unit** of software design(the software component or module).
- Using the **component-level design description** as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The relative **complexity of tests and the errors those tests uncover is limited** by the constrained scope established for unit testing.
- The unit test **focuses on the internal processing logic and data structures** within the boundaries of a component.
- This type of testing **can be conducted in parallel for multiple components.**

UNIT TESTING

Unit Test Considerations.

- The **module interface is tested** to **ensure that information properly flows into and out of the program unit** under test.
- **Local data structures are examined** to **ensure that data stored temporarily maintains its integrity** during all steps in an algorithm's execution.
- **Boundary conditions are tested** to **ensure that the module operates properly at boundaries established** to limit or restrict processing.
- **All independent paths through the control structure** are exercised to **ensure that all statements in a module have been executed at least once.**
- And finally, **all error-handling paths are tested.**
- **Data flow across a component interface is tested before any other testing is initiated.**
- If data do not enter and exit properly, all other tests are moot.

Unit Test Considerations Cont.....

- Selective testing of execution paths is an essential task during the unit test.
- Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.
- Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries.
- A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur.
- This approach is called antibugging.
- Unfortunately, there is a tendency to incorporate error handling into software and then never test the error handling.
- If error-handling paths are implemented, they must be tested.

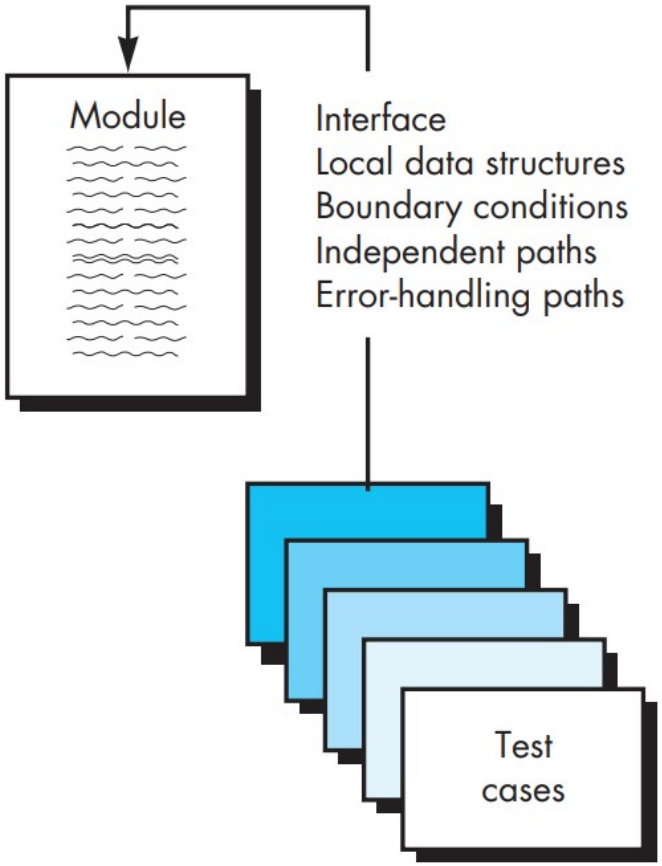


Figure 22: Unit test

INTEGRATION TESTING

- Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which **individual software modules are combined and tested as a group.**
- Integration testing is conducted to **evaluate** the compliance of a **system or component with specified functional requirements.**
- The objective is to **take unit-tested components and build a program structure** that has been dictated by design
- Conducting tests to uncover errors associated with interfacing.

INTEGRATION TESTING Vont...

- **Incremental integration** is the antithesis of the big bang approach.
- The **program is constructed and tested in small increments,**
  - where errors are easier to isolate and correct;
  - interfaces are more likely to be tested completely;
  - and a systematic test approach may be applied.
- A number of different incremental integration strategies are discussed below.
  1. Top-Down Integration
  2. Bottom-Up Integration
  3. Regression Testing
  4. Smoke Testing

INTEGRATION TESTING

1. Top-Down Integration:
   - **An incremental approach** to construction of the software architecture.
   - **Modules are integrated by moving downward** through the control hierarchy, **beginning with the main control module** (main program).
   - Modules subordinate to the main control module are **incorporated** into the structure in either a **depth-first or breadth-first manner.**
2. Bottom-Up Integration
   - **Begins** construction and testing with **atomic modules** (i.e., components at the lowest levels in the program structure).
   - Because **components are integrated from the bottom up**, the **functionality** provided by components subordinate to a given level is **always available** and the need for stubs is eliminated.

3 Regression Testing

- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked.
- Side effects associated with these changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, **regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.**
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

4 Smoke Testing

- Smoke testing is a type of software testing which **ensures that the major functionalities of the application are working fine.**
- It is a non-exhaustive testing with very **limited test cases** to **ensure that the important features are working fine and we are good to proceed with the detailed testing.**
- Smoke testing is an integration testing approach that is commonly used when product software is developed.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

VALIDATION TESTING

- The process of evaluating software during the development process or at the end of the development process **to determine whether it satisfies specified business requirements.**
- Validation Testing **ensures that the product actually meets the client's needs.** It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.
- **Quality control comes under validation testing.**
- Testing **focuses on user-visible actions and user-recognizable output** from the system.
- Validation can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

SYSTEM TESTING

- System Testing is a type of software testing that is performed on a **complete integrated system to evaluate the compliance of the system with the corresponding requirements.**
- System Testing is **carried out** on the whole system in the context of **either system requirement specifications or functional requirement specifications or in the context of both.**
- System testing **tests** the **design and behavior of the system** and also **the expectations of the customer.**

SYSTEM TESTING

System Testing Process:

- Test Environment Setup
- Create Test Case
- Create Test Data
- Execute Test Case
- Defect Reporting
- Regression Testing
- Log Defects
- Retest: If the test is not successful then again test is performed.

SYSTEM TESTING

Types of System Testing:

- **Performance Testing:** Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.
- **Load Testing:** Load Testing is a type of software Testing which is carried out to determine the behavior of a system or software product under extreme load.
- **Stress Testing:** Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.
- **Scalability Testing:** Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

DEBUGGING

- Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.
- The debugging process will usually have one of two outcomes:
  - The cause will be found and corrected or
  - The cause will not be found

DEBUGGING

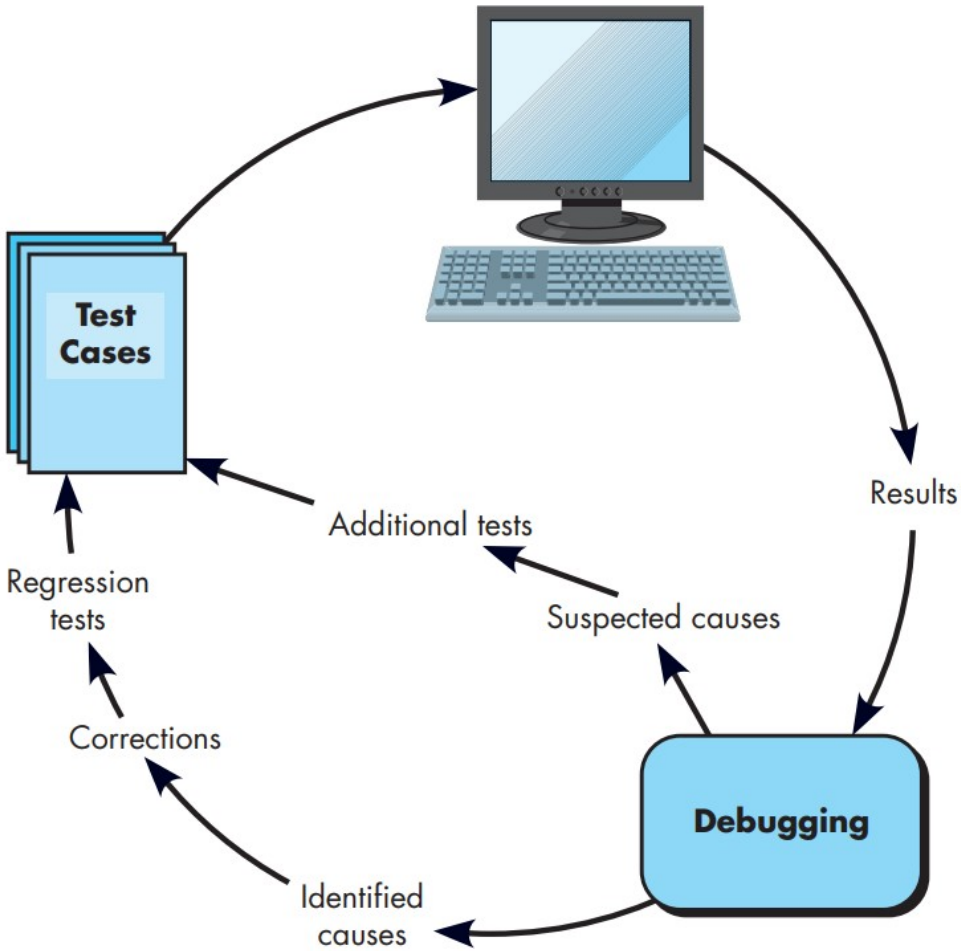


Figure 23: The debugging process

DEBUGGING

Strategies:

- **The brute force:** It is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- **Backtracking:**This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases.
- **Cause elimination method:**In this approach, once a failure is observed, the symptoms of the failure are noted. Based on the failure symptoms, the causes which could have contributed to the symptom are developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

White box testing

White box testing

- White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of software testing that **tests internal structures or working of an application.**
- White-box testing can be applied at the unit, integration and system levels of the software testing process.

White box testing

White box testing cont..

- Using white-box testing methods, you can derive test cases that
  - guarantee that all independent paths within a module have been exercised at least once,
  - exercise all logical decisions on their true and false sides,
  - execute all loops at their boundaries and within their operational bounds, and
  - exercise internal data structures to ensure their validity.
- It is mostly done by software developers
- Knowledge of implementation is required.
- It is the inner or the internal software testing.

Path testing

- Basis Path Testing in software engineering is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program.
- The objective of basis path testing is to **define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.**
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Path testing

Path Testing Process:

- Draw the corresponding control flow graph of the program in which all the executable paths are to be discovered.(Control Flow Graph)
- After the generation of the control flow graph, calculate the cyclomatic complexity of the program using the following formula.(Cyclomatic Complexity)
  - McCabe's *Cyclomatic Complexity* $= E - N + 2P$
    Where, E = Number of edges in control flow graph
    N = Number of vertices in control flow graph
    P = Program factor
- Make a set of all the path according to the control flow graph and calculated. The cardinality of set is equal to the calculated cyclomatic complexity. (Make Set)
- Create Test Cases: Create test case for each path of the set obtained in above step.

Path testing

Path Testing Techniques:

- **Control Flow Graph:** The program is converted into control flow graph by representing the code into nodes and edges.
- **Decision to Decision path:** The control flow graph can be broken into various Decision to Decision paths and then collapsed into individual nodes.
- **Independent paths:** Independent path is a path through a Decision-to-Decision path graph which cannot be reproduced from other paths by other methods.

Control Structure testing

- Control structure testing is a group of white-box testing methods.
- Control structure testing is used to increase the coverage area by testing various control structures present in the program.
- The different types of testing performed under control structure testing are as follows
  1. Condition Testing
     - Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors.
  2. Data Flow Testing
     - he data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program.
  3. Loop Testing
     - Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction.

Control Structure testing

There are four different classes of loops

- Simple Loop
- Nested Loop
- Concatenated Loop
- Unstructured Loop

Control Structure testing

Simple Loop Testing

- A simple loop is tested in the following way:
  1. Skip the entire loop.
  2. Make 1 pass through the loop.
  3. Make 2 passes through the loop.
  4. Make m passes through the loop where $m < n$, n is the maximum number of passes through the loop.
  5. Make $n, n-1, n+1$ passes through the loop.

Nested Loop Testing

- A nested loop is tested in the following way.
  1. Start the innermost loop.
  2. Conduct a simple loop test for the innermost loop.
  3. Work outward, conducting a test for the next loop keeping all other loops at a minimum.
  4. Continue until all the loops are tested.

Control Structure testing

Concatenated Loop Testing

- To test the concatenated loops, the procedure is
  1. if the loops are independent then test them as simple loops
  2. otherwise test them as nested loops.

Unstructured Loop Testing

- To test the unstructured loops we needs to restructure their design.

Control Structure testing

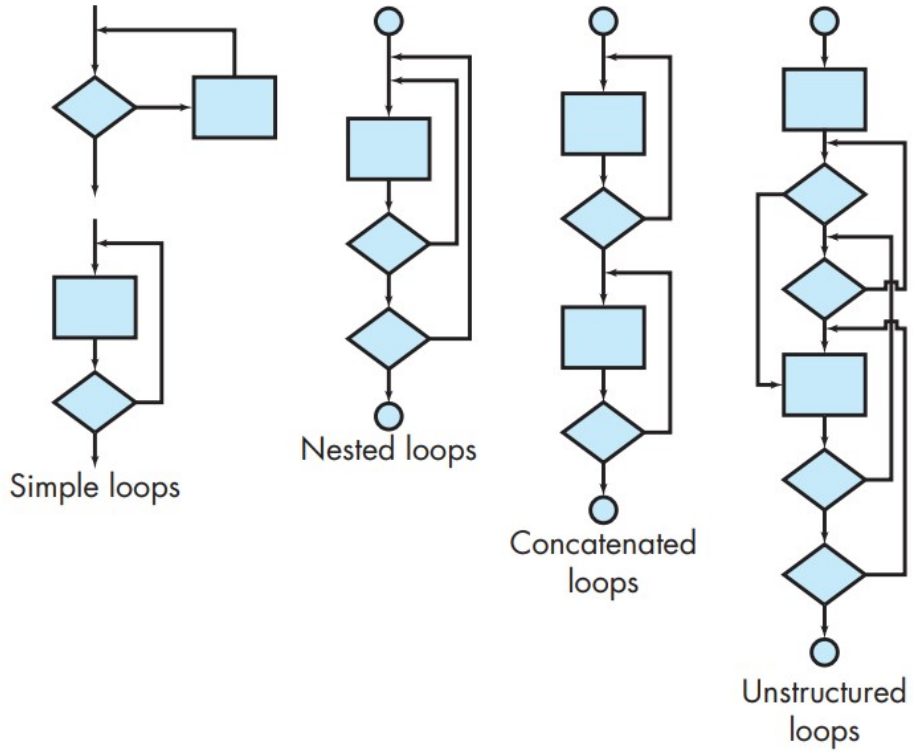


Figure 24: Classes of Loops

Black box testing

Black box testing

- Black Box Testing is a software testing method in which the functionalities of software applications are **tested without having knowledge of internal code structure, implementation details and internal paths.**
  - That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
  - incorrect or missing functions,
  - interface errors,
  - errors in data structures or external database access,
  - behavior or performance errors, and
  - initialization and termination errors.
- Blackbox testing tends to be applied during later stages of testing.

Test documentation

- Test documentation is documentation of artifacts created before or during the testing of software.
- It helps the testing team to estimate testing effort needed, test coverage, resource tracking, execution progress, etc.
- The main reason behind creating test documentation is to either reduce or remove any uncertainties about the testing activities.
- Helps you to remove ambiguity which often arises when it comes to the allocation of tasks
- Documentation not only offers a systematic approach to software testing, but it also acts as training material to freshers in the software testing process
- Test documentation helps you to offer a quality product to the client within specific time limits.

Test documentation

Types of Testing Documents	Description
Test policy	It is a high-level document which describes principles, methods and all the important testing goals of the organization.
Test strategy	A high-level document which identifies the Test Levels (types) to be executed for the project.
Test plan	A test plan is a complete planning document which contains the scope, approach, resources, schedule, etc. of testing activities.
Requirements Traceability Matrix	This is a document which connects the requirements to the test cases.
Test Scenario	Test scenario is an item or event of a software system which could be verified by one or more Test cases.
Test case	It is a group of input values, execution preconditions, expected execution postconditions and results. It is developed for a Test Scenario.
Test Data	Test Data is a data which exists before a test is executed. It used to execute the test case.
Defect Report	Defect report is a documented report of any flaw in a Software System which fails to perform its expected function.
Test summary report	Test summary report is a high-level document which summarizes testing activities conducted as well as the test result.

Test documentation

INFO

Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document's table of contents and index robust, accurate, and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

Test automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- Can run the test and the test passes if the unit returns the expected result.
- Normally, hundreds or thousands of executable tests are developed for a software product.

Test automation

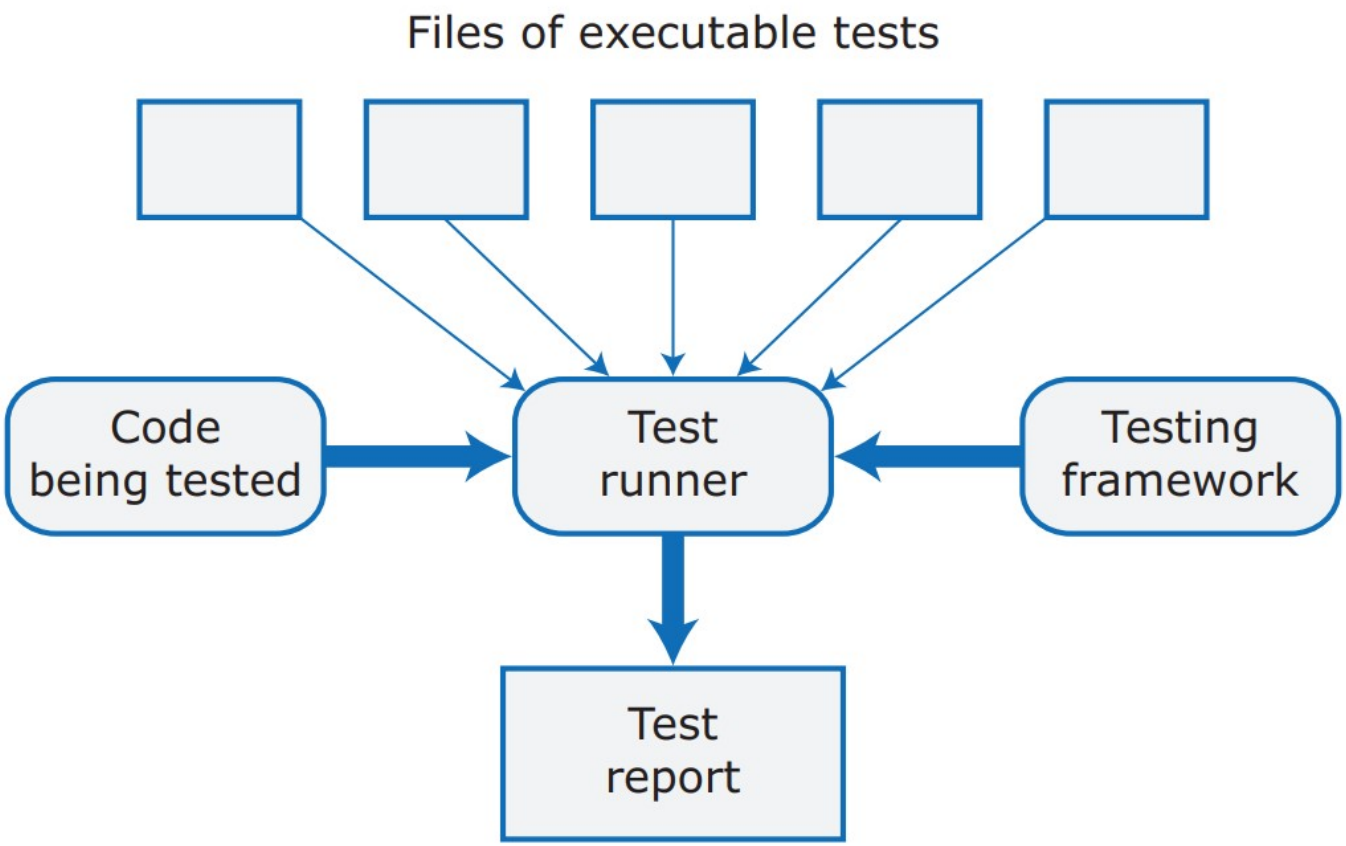


Figure 25: Automated testing

Test automation

Automated tests

- It is good practice to structure automated tests into three parts:
  - **Arrange** can set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
  - **Action** can call the unit that is being tested with the test parameters.
  - **Assert** can make an assertion about what should hold if the unit being tested has executed successfully.For example AssertEquals, which checks if its parameters are equal.
- Can use equivalence partitions to identify test inputs, should have several automated tests based on correct and incorrect inputs from each partition.

Automated tests

Testing Pyramid

- Mike Cohn proposed the testing pyramid, suggests that 70% of automated tests should be unit tests, 20% feature tests(called these service tests), and 10% system tests (UI tests).
- The implementation of system features usually involves integrating functional units into components and then integrating these components to implement the feature.
- If you have good unit tests, then be confident that the individual functional units and components that implement the feature will behave as expected.

Test automation

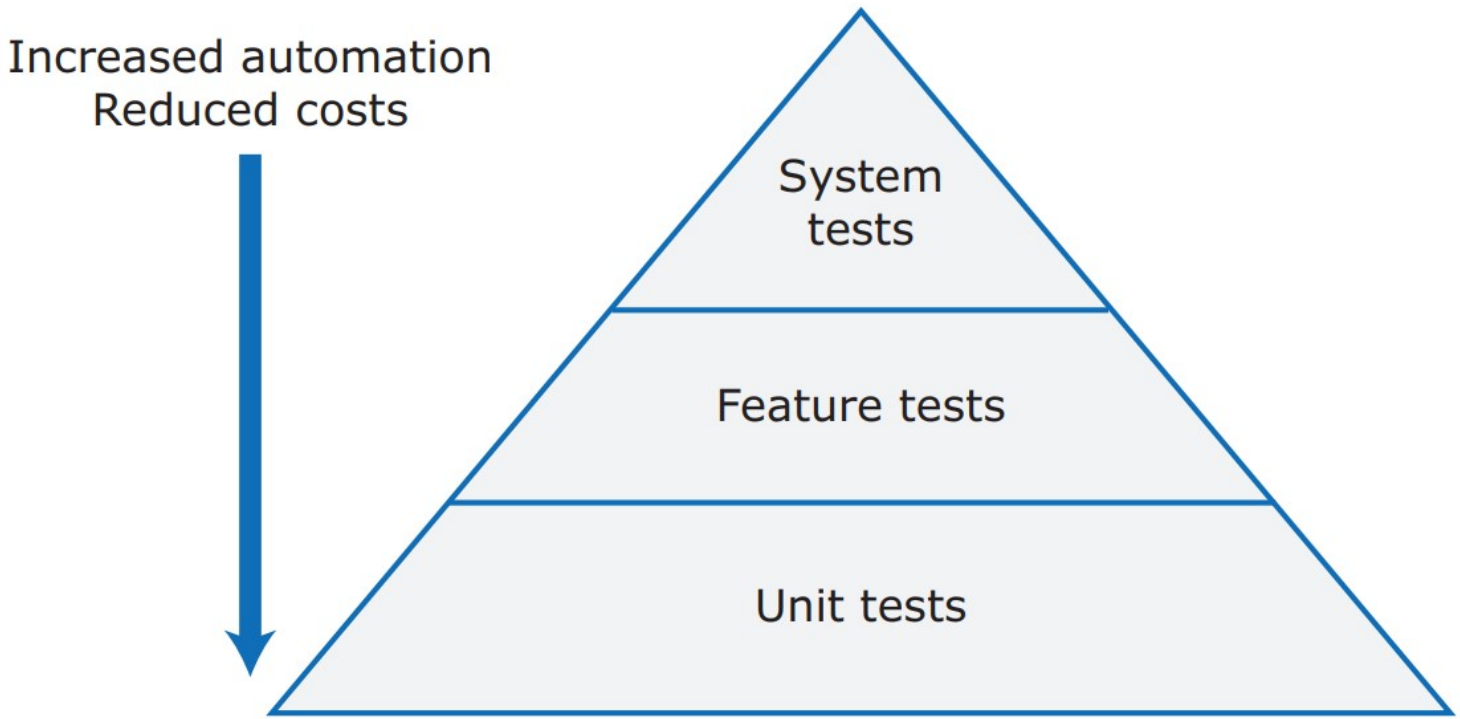


Figure 26: The test pyramid

Test automation

Automated feature testing

- **Generally, users access features through the product's graphical user interface (GUI)**.
- **GUI-based testing is expensive to automate** so it is best to design your product **so that its features can be directly accessed through an API** and not just from the user interface.
- The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

Test automation

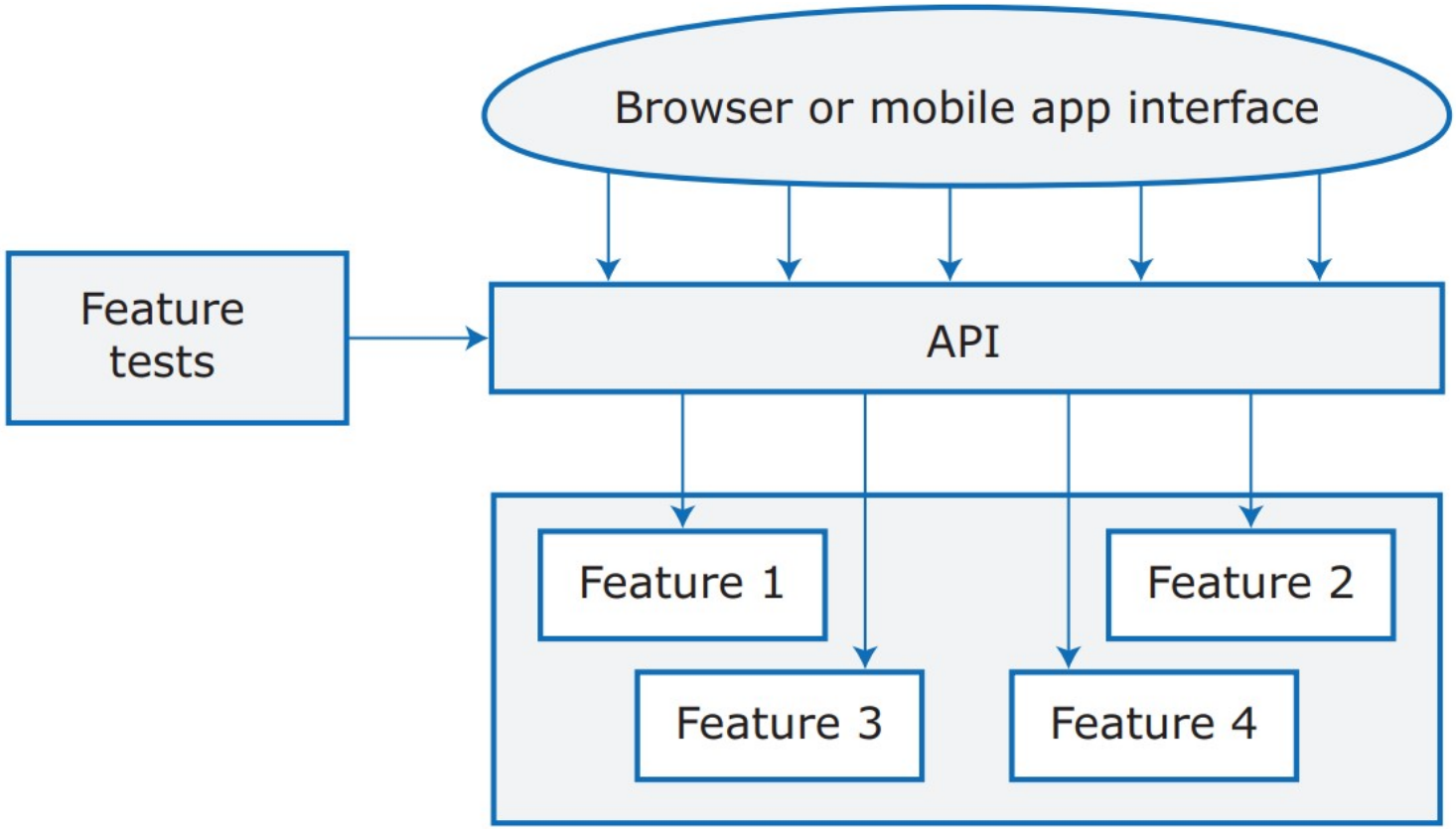


Figure 27: Feature testing through an API

Test-driven development

- Test-driven development (TDD) is an approach to program development that is based around the general idea that you should w**rite an executable test or tests for code** that you are writing **before you write the code**.
- It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- **Test-driven development works best for the development of individual program units** and **it is more difficult to apply to system testing**.
- Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

Test-driven development

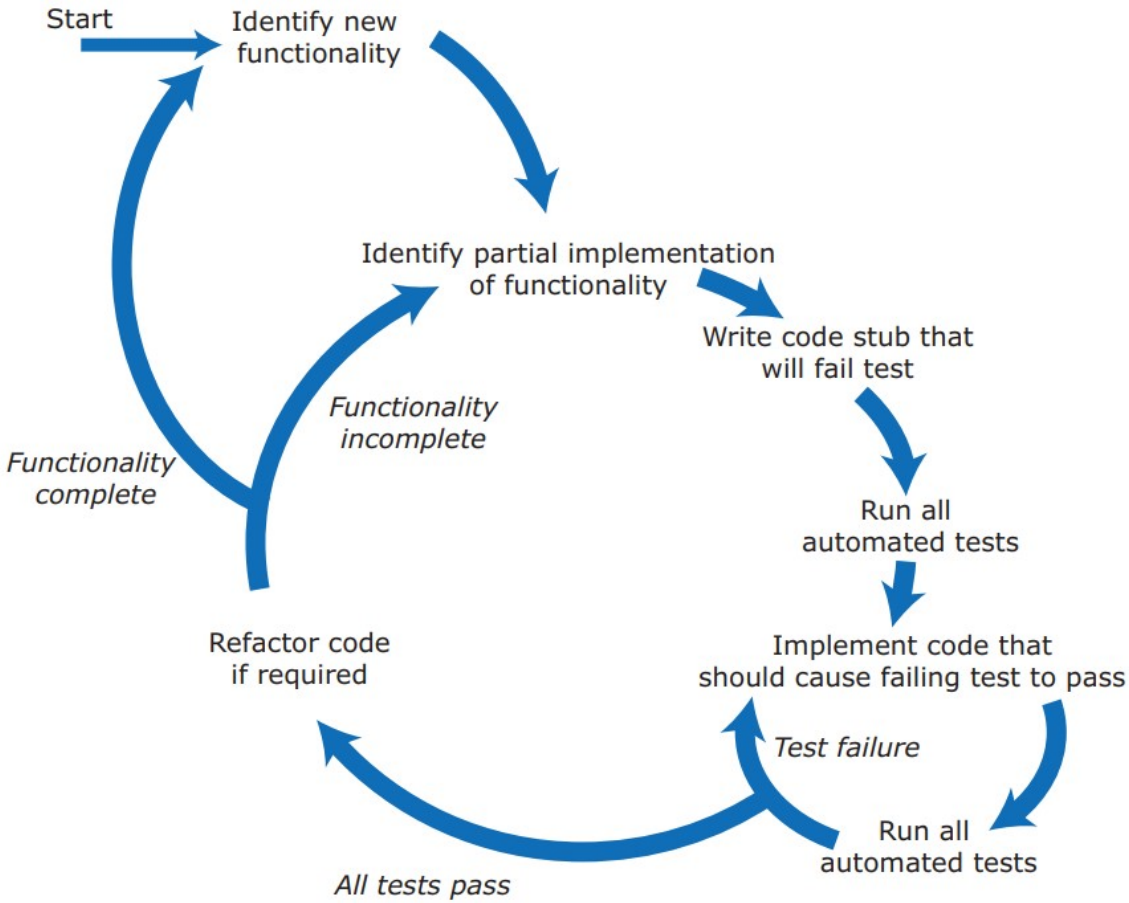


Figure 28: Test-driven development

Test-driven development

- Assume that we have identified some increment of functionality to be implemented.
- Test-driven development relies on automated testing. Every time you add some functionality, you develop a new test and add it to the test suite.
- All of the tests in the test suite must pass before you move on to developing the next increment.

Test-driven development

Activity	Description
Identify partial implementation	Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
Write mini-unit tests	Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
Write a code stub that will fail test	Write incomplete code that will be called to implement the mini-unit. You know this will fail.
Run all automated tests	Run all existing automated tests. All previous tests should pass. The test for the incomplete code should fail.
Implement code that should cause the failing test to pass	Write code to implement the mini-unit, which should cause it to operate correctly.
Rerun all automated tests	If any tests fail, your code is incorrect. Keep working on it until all tests pass.
Refactor code if required	If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

Figure 29: Stages of test-driven development

Benefits of test-driven development

- It is a systematic approach to testing in which tests are clearly linked to sections of the program code.
  - This means it is been confident that the tests cover all of the code that has been developed and that there are no untested code sections in the delivered code.This is the most significant benefit of TDD.
- The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.
- Debugging is simplified because, when a program failure is observed, it can immediately link this to the last increment of code that have added to the system.
- It is argued that TDD leads to simpler code as programmers only write code that's necessary to pass tests. They don't over-engineer their code with complex features that aren't needed.

Security testing

- Security testing aims to **find vulnerabilities** that may be exploited by an attacker and to **provide convincing evidence that the system is sufficiently secure.**
- The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users' data and identity.
- Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

Security testing

Risk-based security testing

- A risk-based approach to security testing involves **identifying common risks and developing tests to demonstrate that the system protects itself from these risks**.
- Can also **use automated tools** that scan the system to **check for known vulnerabilities**, such as unused HTTP ports being left open.
- Based on the risks that have been identified, then design tests and checks to see if the system is vulnerable.
- It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files.

Security testing

Examples of security risks

- Unauthorized attacker gains access to a system using authorized credentials
- Authorized individual accesses resources that are forbidden to them
- Authentication system fails to detect unauthorized attacker
- Attacker gains access to database using SQL poisoning attack
- Improper management of HTTP session
- HTTP session cookies revealed to attacker
- Confidential data are unencrypted
- Encryption keys are leaked to potential attackers

DevOps and Code Management

DevOps

- **To speed up the release and support processes**, an alternative approach called DevOps has been developed.
- **DevOps** (development + operations) **integrates development, deployment, and support, with a single team responsible for all of these activities**.
- Three factors led to the development and widespread adoption of DevOps:
  - **Agile** software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.
  - **Amazon** re-engineered their software around services and introduced an approach in which a service was developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.
  - It became possible to release software as a service, running on a **public or private cloud**. Software products did not have to be released to users on physical media or downloads.

DevOps and Code Management

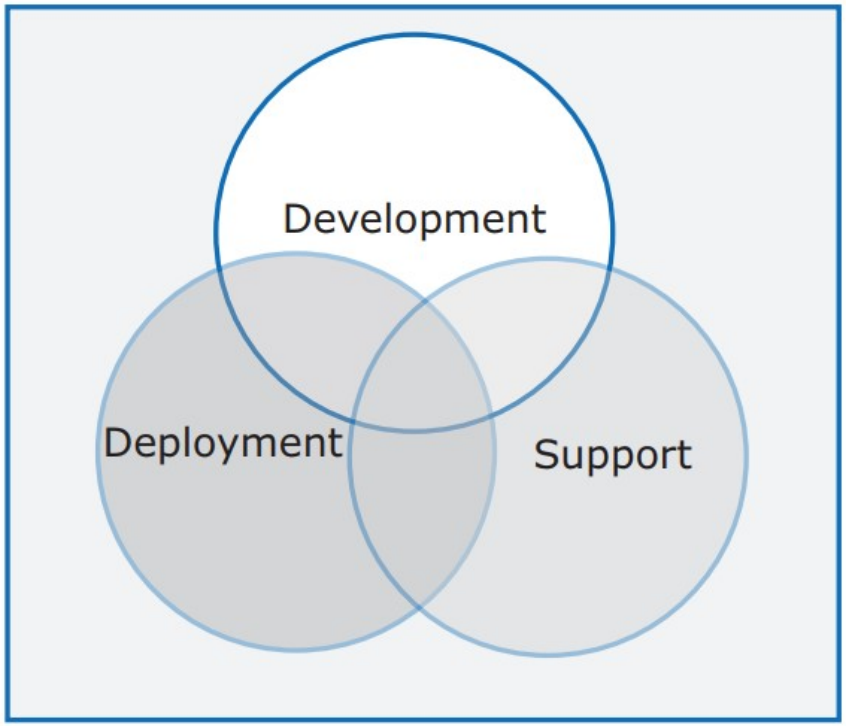


Figure 30: Multi-skilled DevOps team

DevOps principles

- Everyone is responsible for everything.
- Everything that can be automated should be automated.
- Measure first, change later.
  - DevOps should be driven by a measurement program where you collect data about the system and its operation. Then use the collected data to inform decisions about changing DevOps processes and tools.

Benefits of DevOps

- Faster deployment
- Reduced risk
- Faster repair
- More productive teams

Code Management

Alice and Bob worked for a company called FinanceMadeSimple and were team members involved in developing a personal finance product. Alice discovered a bug in a module called TaxReturnPreparation. The bug was that a tax return was reported as filed but sometimes it was not actually sent to the tax office. She edited the module to fix the bug. Bob was working on the user interface for the system and was also working on TaxReturnPreparation. Unfortunately, he took a copy before Alice had fixed the bug and, after making his changes, he saved the module. This overwrote Alice's changes, but she was not aware of this.

The product tests did not reveal the bug, as it was an intermittent failure that depended on the sections of the tax return form that had been completed. The product was launched with the bug. For most users, everything worked OK. However, for a small number of users, their tax returns were not filed and they were fined by the revenue service. The subsequent investigation showed the software company was negligent. This was widely publicized and, as well as a fine from the tax authorities, users lost confidence in the software. Many switched to a rival product. FinanceMadeSimple failed and both Bob and Alice lost their jobs.

Figure 31: A code management problem

Code Management

Code management

- Code management is a **set of software-supported practices that is used to manage an evolving codebase**.
- Code management need to **ensure that**
  - **changes made** by **different developers do not interfere** with each other and to create different product versions.
- Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.

Code Management

Code management and DevOps

- Source code management, combined with automated system building, is essential for professional software engineering.
- In companies that use DevOps, a modern code management system is a fundamental requirement for 'automating everything'.
- Not only does it store the project code that is ultimately deployed, it also stores all other information that is used in DevOps processes.
- DevOps automation and measurement tools all interact with the code management system

DevOps and Code Management

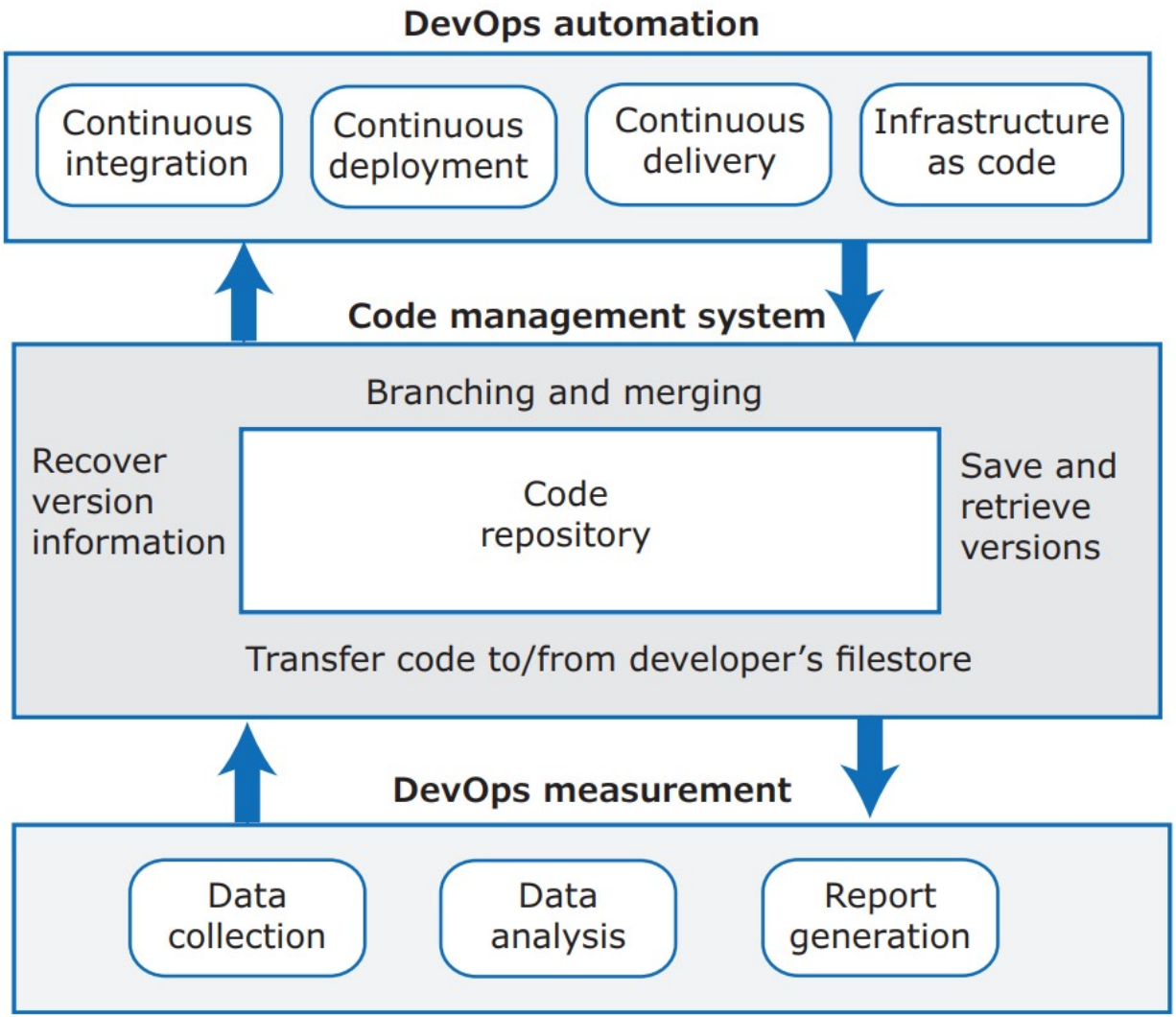


Figure 32: Code management and DevOps

Code Management

Fundamentals of source code management

- Code management systems provide a set of features that support four general areas:
  - **Code transfer** Developers take code into their personal file store to work on it then return it to the shared code management system.
  - **Version storage and retrieval** Files may be stored in several different versions and specific versions of these files can be retrieved.
  - **Merging and branching** Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.
  - **Version information** Information about the different versions maintained in the system may be stored and retrieved

Fundamentals of source code management

Feature	Description
Version and release identification	Managed versions of a code file are uniquely identified when they are submitted to the system and can be retrieved using their identifier and other file attributes.
Change history recording	The reasons changes to a code file have been made are recorded and maintained.
Independent development	Several developers can work on the same code file at the same time. When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes.
Project support	All of the files associated with a project may be checked out at the same time. There is no need to check out files one at a time.
Storage management	The code management system includes efficient storage mechanisms so that it doesn't keep multiple copies of files that have only small differences.

Figure 33: Features of source code management systems

Code repository

- All source code management systems have the general form with a shared repository and a set of features to manage the files in that repository:
  - All source code files and file versions are stored in the repository, as are other artefacts such as configuration files, build scripts, shared libraries and versions of tools used.
  - Files can be transferred to and from the repository and information about the different versions of files and their relationships may be updated. Specific versions of files and information about these versions can always be retrieved from the repository.

Code Management

Git

- Git is an **open-source Version Control System (VCS)**, it is completely free.
- Git is designed to work in small to large level projects.
- Git will help to merge and maintain the history of code changes.
- Github is the repository where all the source code is kept by Git users.
- GitHub offers local branching and multiple workflows.
- Instead of only keeping the copies of the files that users are working on, **Git maintains a clone of the repository on every user's computer**
- A **fundamental concept in Git** is the **"master branch,"** which is the current master version of the software that the team is working on.

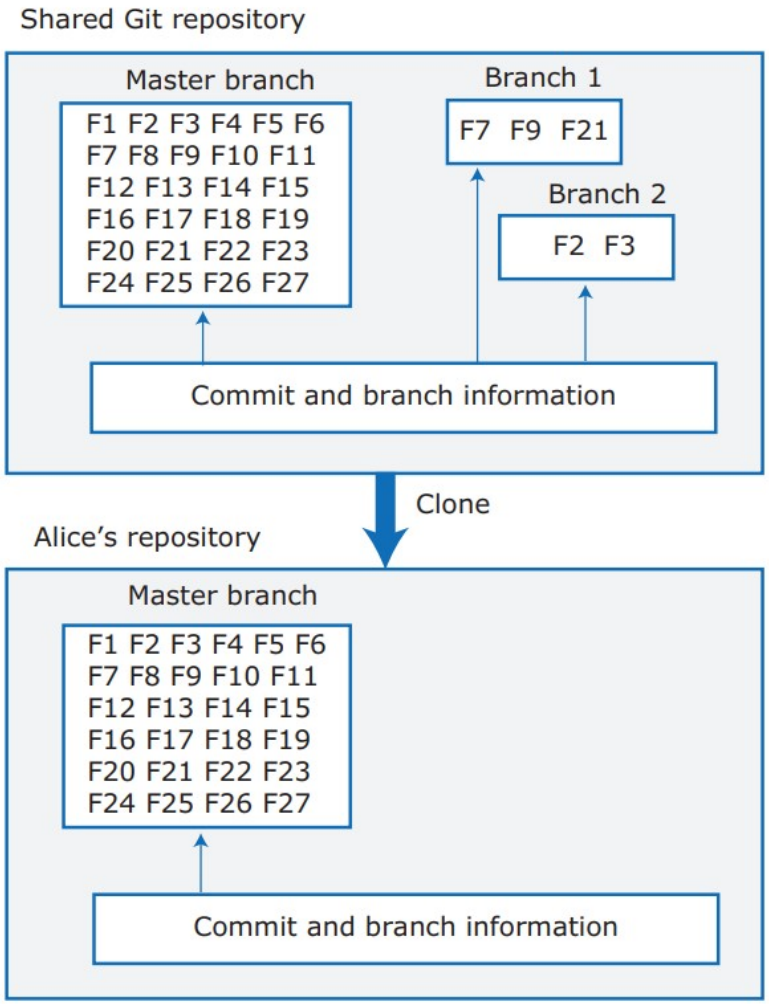


Figure 34: Repository cloning in Git

Code Management

Benefits of distributed code management

- Resilience
  - **Everyone working on a project has their own copy** of the repository. **If the shared repository is damaged** or subjected to a cyberattack, work can continue, and the **clones can be used to restore the shared repository**. People can work offline if they don't have a network connection.
- Speed
  - Committing changes to the repository is a fast, local operation and does not need data to be transferred over the network.
- Flexibility
  - **Local experimentation is much simpler.** Developers can safely experiment and try different approaches without exposing these to other project members. With a centralized system, this may only be possible by working outside the code management system.

Code Management

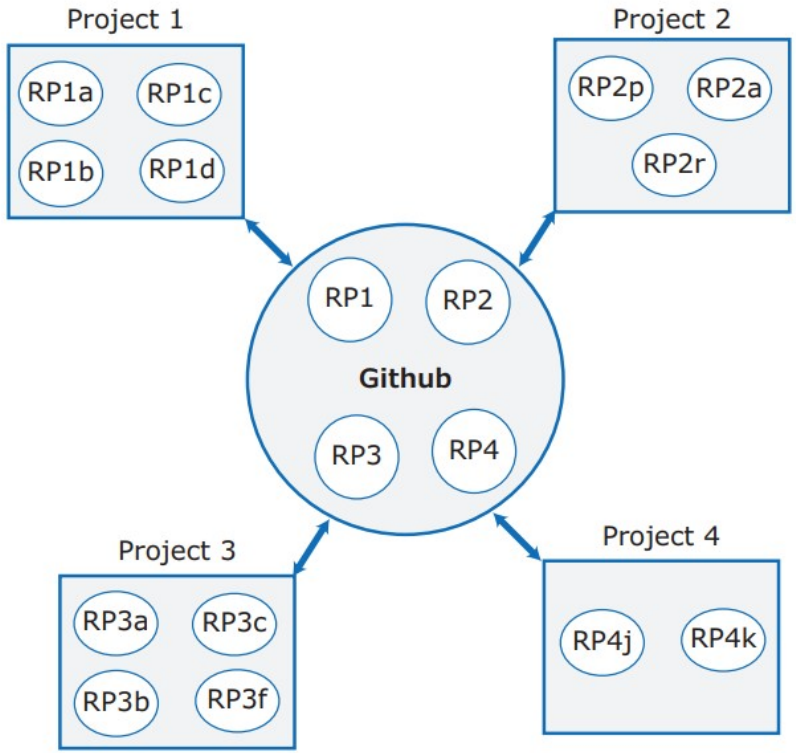


Figure 35: Git repositories

Code Management

Git Cont..

- Most software product companies now use Git for code management.
- For teamwork, Git is organized around the notion of a shared project repository and private clones of that repository held on each developer's computer .
- A company may use its own server to run the project repository. However, many companies and individual developers use an external Git repository provider.
- Several Git repository **hosting companies, such as Github and Gitlab**, host thousands of repositories on the cloud.

Git Cont..

- Figure shows four project repositories on Github, RP1–RP4. RP1 is the repository for project 1, RP2 is the repository for project 2, and so on. Each of the developers on each project is identified by a letter (a, b, c, etc.) and has an individual copy of the project repository.
- Developers may work on more than one project at a time, so they may have copies of several Git repositories on their computer.
- For example, developer a works on Project 1, Project 2, and Project 3, so has clones of RP1, RP2, and RP3.

DevOps automation

- By using **DevOps with automated support,** can dramatically **reduce the time and costs for integration, deployment and delivery.**
- Everything that can be, should be automated is a fundamental principle of DevOps.
- As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.
- Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.

DevOps automation

Aspect	Description
Continuous integration	Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
Continuous delivery	A simulation of the product's operating environment is created and the executable software version is tested.
Continuous deployment	A new release of the system is made available to users every time a change is made to the master branch of the software.
Infrastructure as code	Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infastructure model.

Figure 36: Aspects of DevOps automation

Continuous integration

- Continuous integration simply means that an **integrated version of the system is created and tested every time a change is pushed to the system's shared repository.**
- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product
- The **advantage of continuous** integration compared to less frequent integration is that it is **faster to find and fix bugs in the system.**
- If a small change are made and some system tests then fail, the problem almost certainly lies in the new code that have pushed to the project repo.
- Focus on this code to find the bug that's causing the problem.

DevOps automation

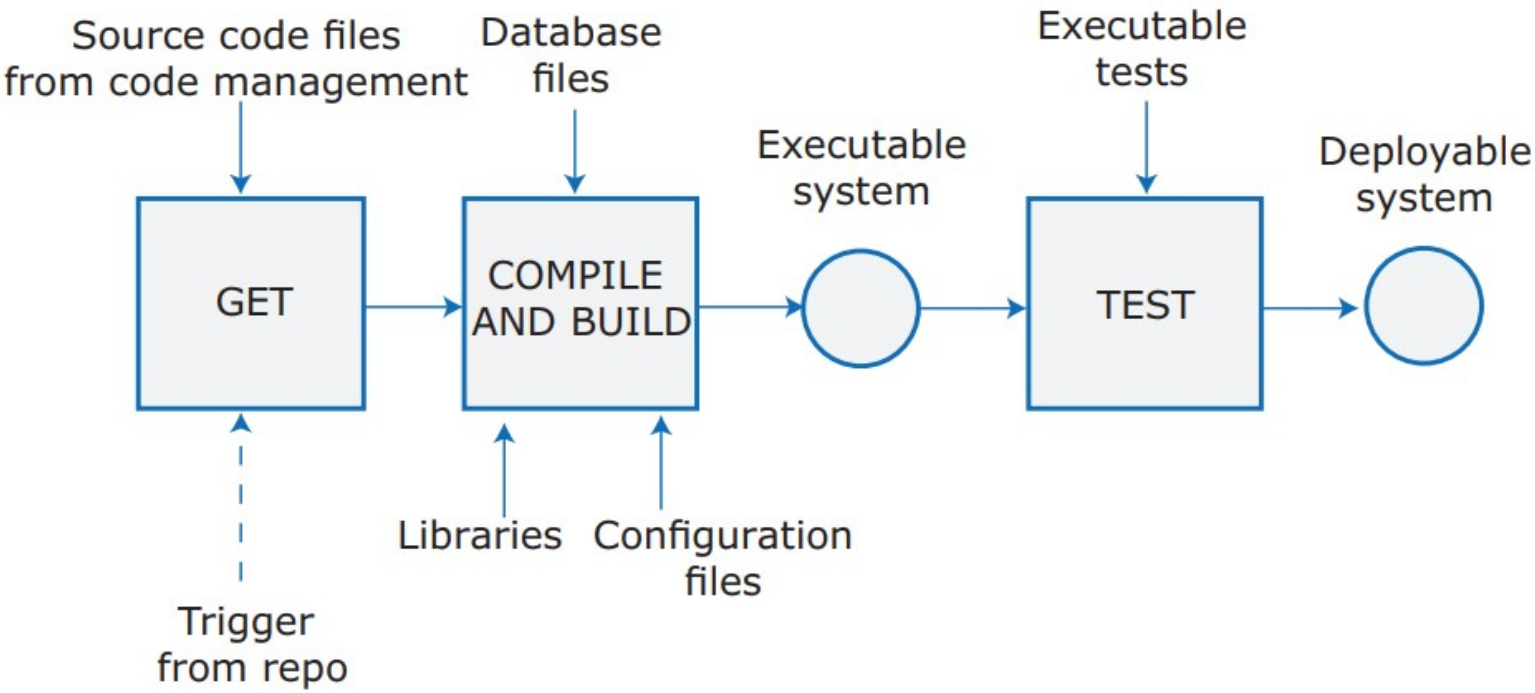


Figure 37: Continuous integration

DevOps automation

Continuous delivery and deployment

- Continuous integration means creating an executable version of a software system whenever a change is made to the repository. The CI tool builds the system and runs tests on your development computer or project integration server.
- However, the real environment in which software runs will inevitably be different from your development system.
- When your software runs in its real, operational environment bugs may be revealed that did not show up in the test environment.
- **Continuous delivery means that, after making changes to a system, you ensure that the changed system is ready for delivery to customers.**
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.

DevOps automation

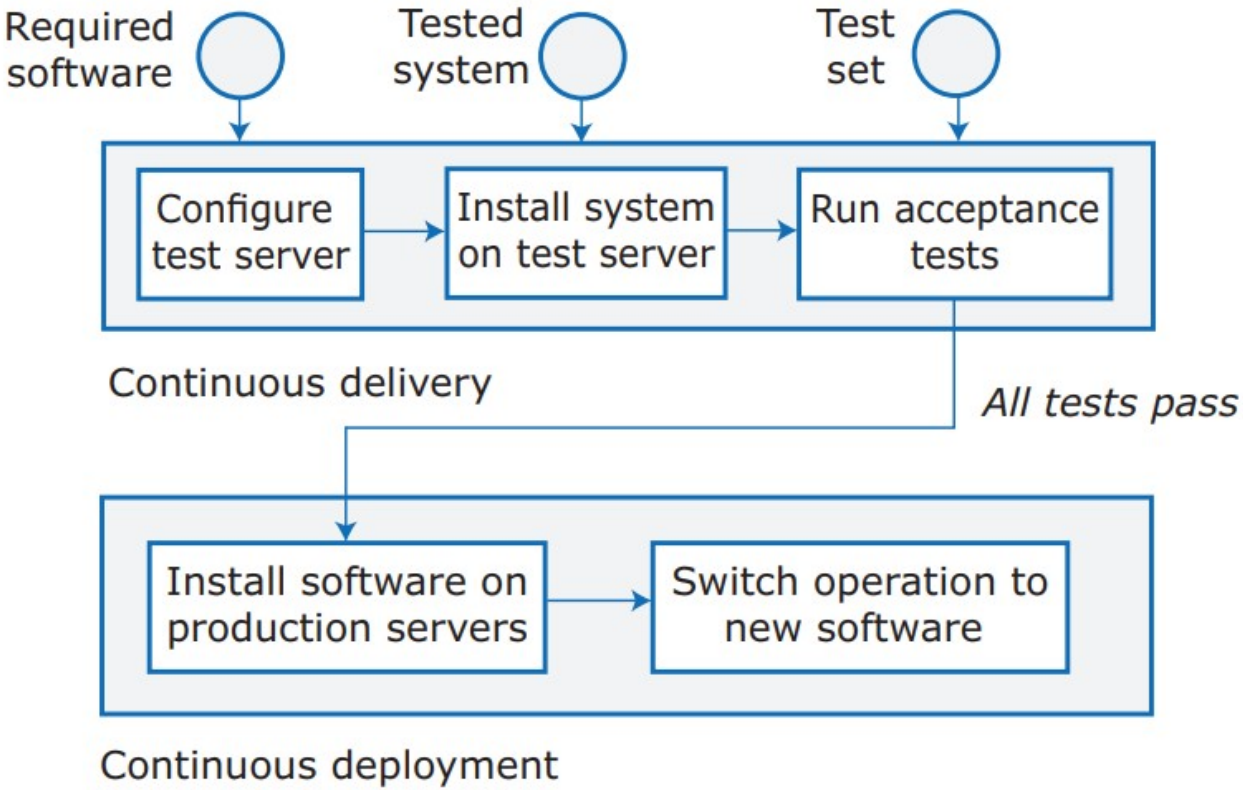


Figure 38: Continuous delivery and deployment

DevOps automation

Benefit	Explanation
Reduced costs	If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and takes time, but you can recover these costs quickly if you make regular updates to your product.
Faster problem solving	If a problem occurs, it will probably affect only a small part of the system and the source of that problem will be obvious. If you bundle many changes into a single release, finding and fixing problems are more difficult.
Faster customer feedback	You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.
A/B testing	This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can measure and assess how new features are used to see if they do what you expect.

Figure 39: Benefits of continuous deployment

DevOps automation

Infrastructure as code

- **In an enterprise environment**, there are usually many **different physical or virtual servers** (web servers, database servers, file servers, etc.) **that do different things.** These **have different configurations** and run different software packages.
- It is therefore **difficult to keep track of the software installed on each machine.**
- The idea of **infrastructure as code was proposed as a way to address this problem**. Rather than manually updating the software on a company's servers, **the process can be automated using a model of the infrastructure written in a machine-processable language.**
- **Configuration management (CM) tools** such as Puppet and Chef **can automatically install software and services on servers according to the infrastructure definition**

DevOps automation

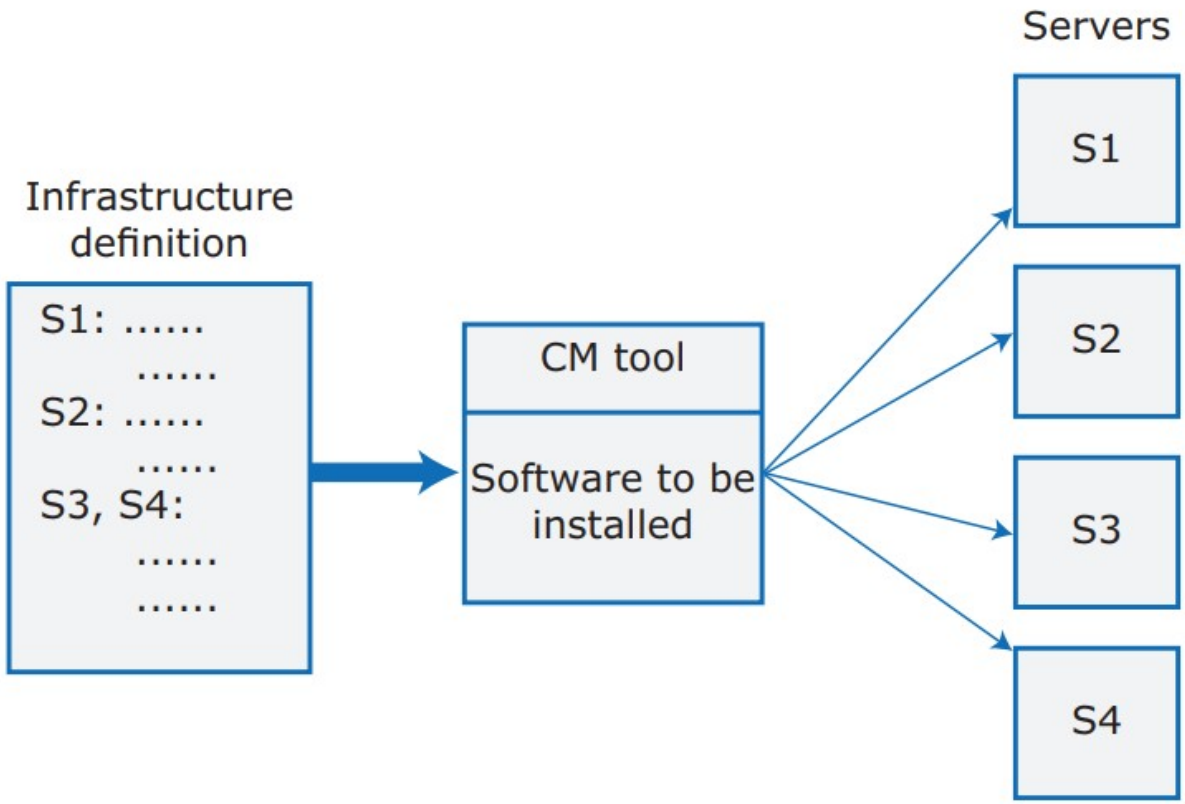


Figure 40: Infrastructure as code

DevOps automation

Characteristic	Explanation
Visibility	Your infrastructure is defined as a stand-alone model that can be read, discussed, understood, and reviewed by the whole DevOps team.
Reproducibility	Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.
Reliability	In managing a complex infrastructure, system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.
Recovery	Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to an older version and reinstall the environment that you know works.

Figure 41: Characteristics of infrastructure as code

Software Evolution

- Organisations have huge investments in their software systems - they are critical business assets.
- To maintain the value of these assets to the business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

Software Evolution

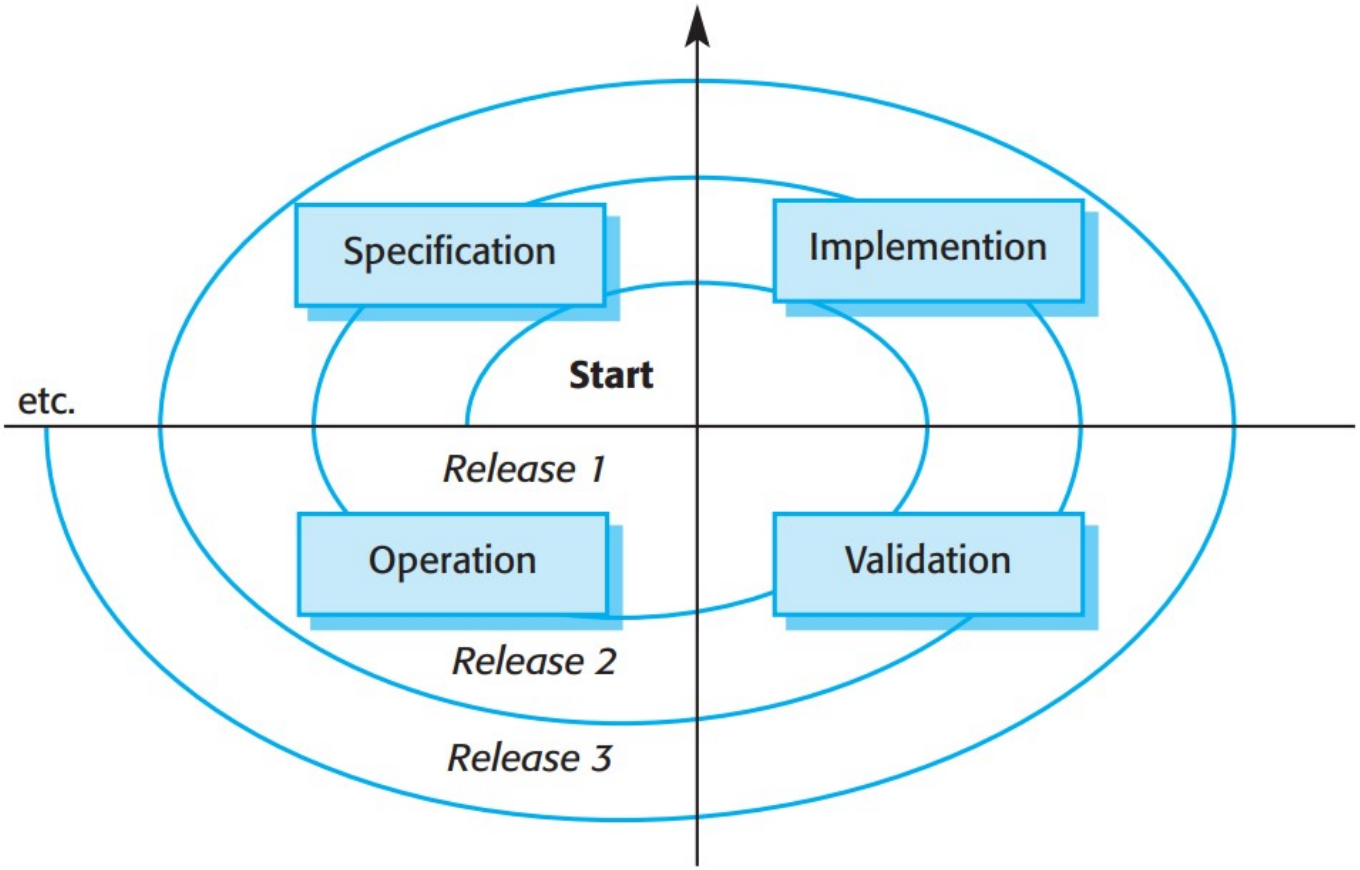


Figure 42: A spiral model of development and evolution

Software Evolution

- Software engineering is a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system.
- You start by creating release1 of the system.
- Once delivered, changes are proposed, and the development of release2 starts almost immediately.
- In fact, the need for evolution may become obvious even before the system is deployed, so later releases of the software may start development before the current version has even been released.

Software Evolution

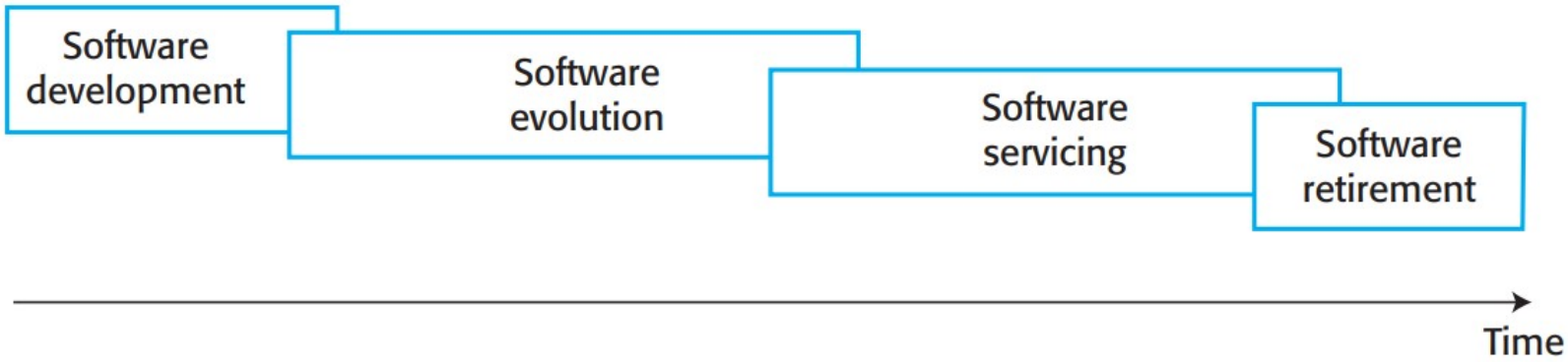


Figure 43: Evolution and servicing

- Alternative view of the software evolution life cycle for business systems
- In this model, they distinguish between evolution and servicing.
- **Evolution** is the phase in which significant changes to the software architecture and functionality are made.
- During **servicing**, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure

Evolution processes

- Software evolution processes depend on
  - the type of software being mainted.
  - the development processses used
  - the skills and experience of the people involved.
- Proposal for change are the driver for system evolution.
  - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- Change identification and evolution continues throughout the system lifetime

Software Evolution

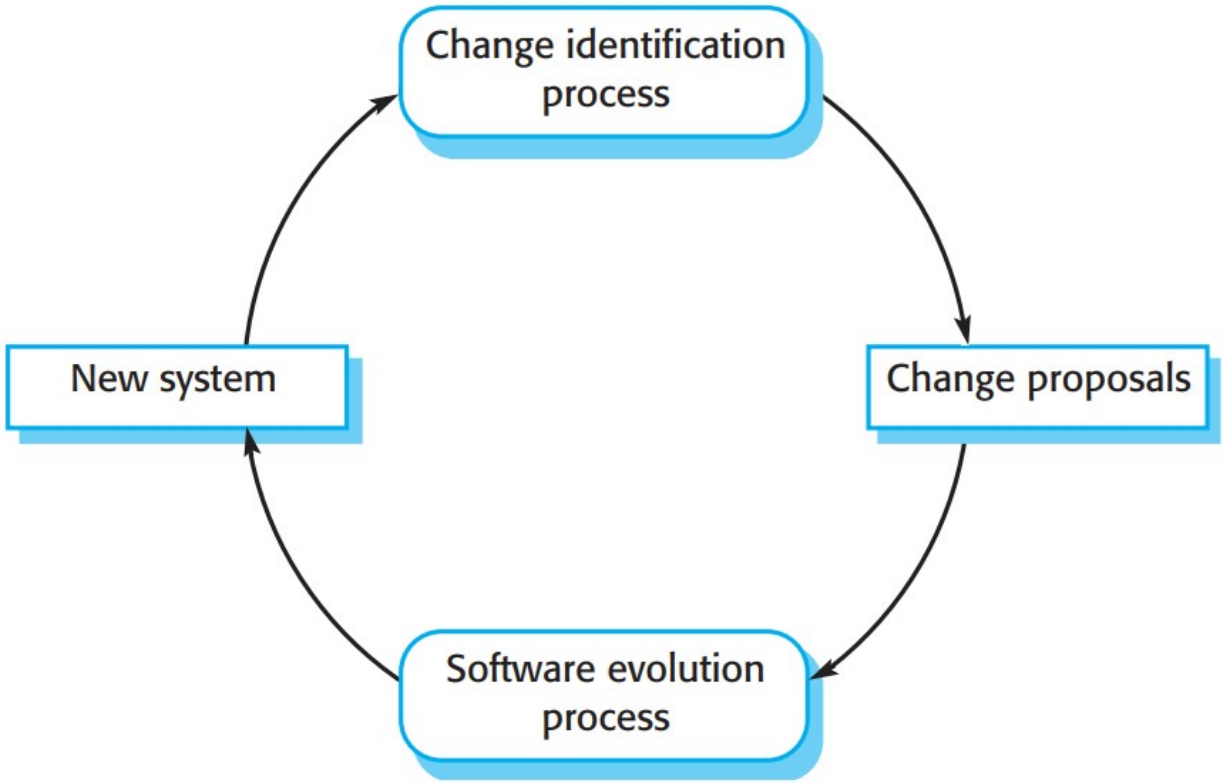


Figure 44: Change identification and evolution processes

Software Evolution

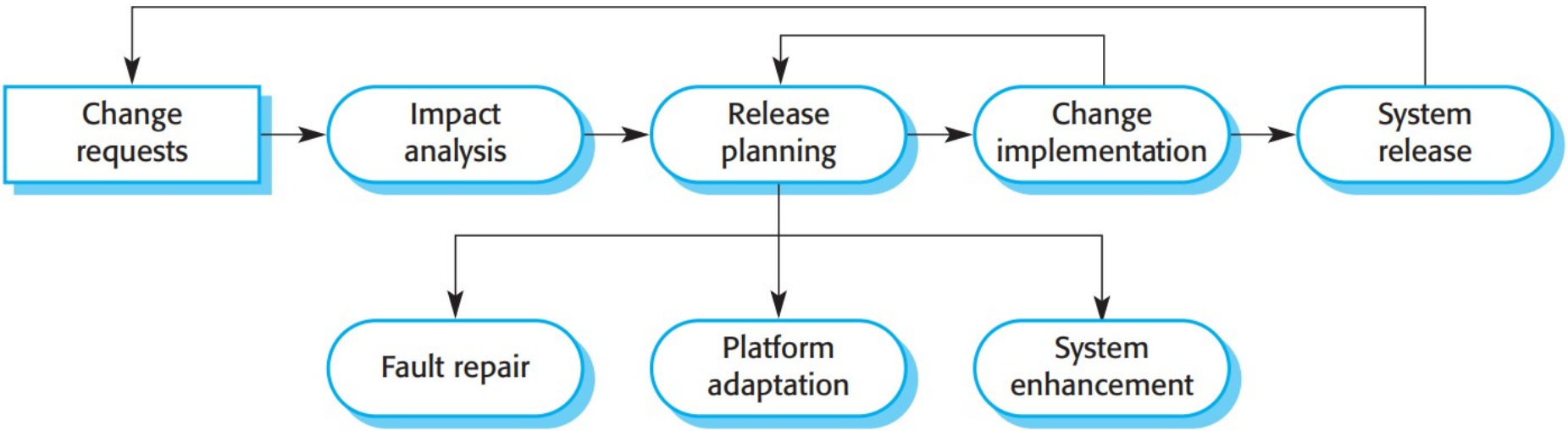


Figure 45: A general model of the software evolution process

Software Evolution

- The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.
- The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.
- If the proposed changes are accepted, a new release of the system is planned.
- During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered.
- A decision is then made on which changes to implement in the next version of the system.
- The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

Software maintenance

Software maintenance

- Software maintenance is the general **process of changing a system after it has been delivered.**
- The term is usually applied to custom software, where separate development groups are involved before and after delivery.
- The changes made to the software **may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements.**

Software maintenance

Types of maintenance

- **Fault repairs** to fix bugs and vulnerabilities. Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
- **Environmental adaptation** to adapt the software to new platforms and environments. This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
- **Functionality addition** to add new features and to support new requirements. This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance

Software maintenance

Two important term

- Maintenance Prediction
- Change Prediction

Software maintenance

Maintenance Prediction:

- Maintenance prediction **is concerned with trying to assess the changes that may be required in a software system** and with **identifying those parts of the system that are likely to be the most expensive to change.**

Change Prediction:

- **Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment.**
- Some systems have a very **complex relationship** with their external environment, and changes to that environment inevitably **result in changes to the system.**
- To evaluate the relationships between a system and its environment, you should look at:
  - The number and complexity of system interfaces
  - The number of inherently volatile system requirements
  - The business processes in which the system is used

Software Reengineering

- Reengineering may **involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data**.
- The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Refactoring

- Refactoring is the process of **making improvements to a program to slow down degradation through change.**
- It means **modifying a program to improve its structure, reduce its complexity,** or make it easier to understand.
- Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach.
- **When you refactor a program, you should not add functionality but rather should concentrate on program improvement.**
- You can therefore think of refactoring as **"preventative maintenance" that reduces the problems of future change.**
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system