KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

MODULE 3

Object-oriented design using the UML, Design patterns, Implementation issues, Open-source development - Open-source licensing - GPL, LGPL, BSD. Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. Informal Review, Formal Technical Reviews, Post-mortem evaluations. Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing, Debugging, White box testing, Path testing, Control Structure testing, Black box testing, Testing Documentation and Help facilities. Test automation, Test-driven development, Security testing. Overview of DevOps and Code Management - Code management, DevOps automation, Continuous Integration, Delivery, and Deployment (CI/CD/CD). Software Evolution - Evolution processes, Software maintenance

Object-oriented design using the UML

System context and interactions

Architectural design

Object class identification

Design models

Interface specification

Object-oriented design using the UML

- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state.
- Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions.
- When the design is realized as an executing program, the objects are created dynamically from these class definitions. Objects include both data and operations to manipulate that data.
- Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system.
- To develop a system design from concept to detailed, object-oriented design, we need to:
  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.
- This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- understanding 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. Eg: we need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.
- System context models and interaction models present complementary views of the relationships between a system and its environment
    1. A **system context model** is a structural model that demonstrates the other systems in the environment of the system being developed.
    2. An **interaction model** is a dynamic model that shows how the system interacts with its environment as it is used.
- The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations.
- When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model.

System context and interactions

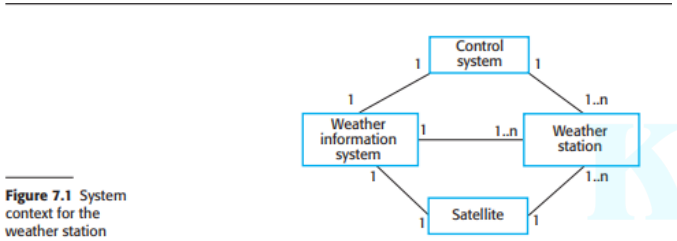


Figure 7.1 System context for the weather station

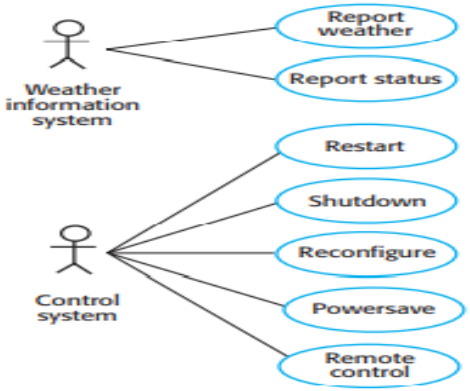


Figure 7.2 Weather station use cases

The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

- the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands.

System context and interactions

- Use case description – report whether

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.

Architectural design

- Once the interactions between the software system and the system's environment have been defined, use this information as a basis for designing the system architecture.
- Combine this knowledge with the general knowledge of the principles of architectural design and with more detailed domain knowledge.
- Identify the major components that make up the system and their interactions.
- Design the system organization using an architectural pattern such as a layered or client–server model.

- high-level architectural design for the weather station software is shown in figure. The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as communication link in 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 system.
- When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. 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 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The Transmitter and Receiver objects are concerned with managing communications, and the WeatherData object encapsulates the information that is collected from the instruments and transmitted to the weather information system.

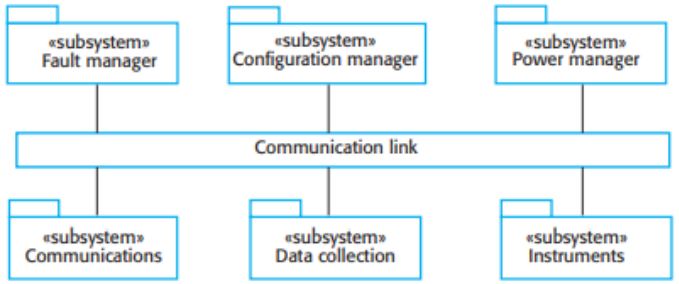


Figure 7.4 High-level architecture of weather station

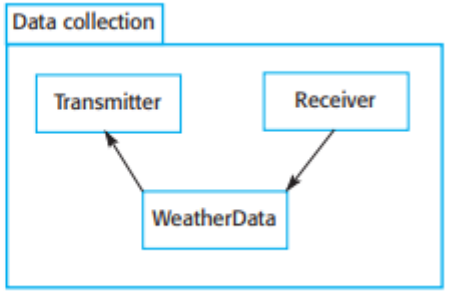


Figure 7.5 Architecture of data collection system

Object class identification

- Some ideas about the essential objects in the system that is being designed is mandatory.
- As the understanding of the design develops, refine these ideas about the system objects.
- The use case description helps to identify objects and operations in the system.
- From the description of the Report weather use case, it is obvious that it is needed to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the general object classes in the system.

various ways of identifying object classes in object-oriented systems were suggested:

- 1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs .
- 2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on
- 3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations.

several knowledge sources are used to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or analyses of existing systems.

Object class identification

- In the wilderness weather station, object identification is based on the tangible hardware in the system.
- Five object classes are shown in the figure.
- The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

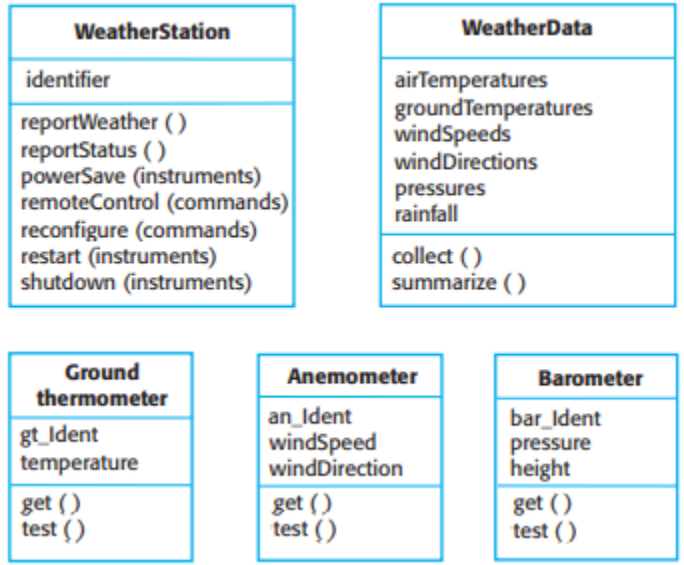


Figure 7.6 Weather station objects

Design models

- Design models show the objects or object classes in a system.
- They also show the associations and relationships between these entities.
- These models are the bridge between the system requirements and the implementation of a system.
- They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.
- The level of detail that you need in a design model depends on the design process used.
- Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.
- if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.
- An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models.

Design models

- When you use the UML to develop a design, you should develop **two kinds of design model**:

1. **Structural models**, which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.

2. **Dynamic models**, which describe the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions

Three UML model types are particularly useful for adding detail to use case and architectural models:

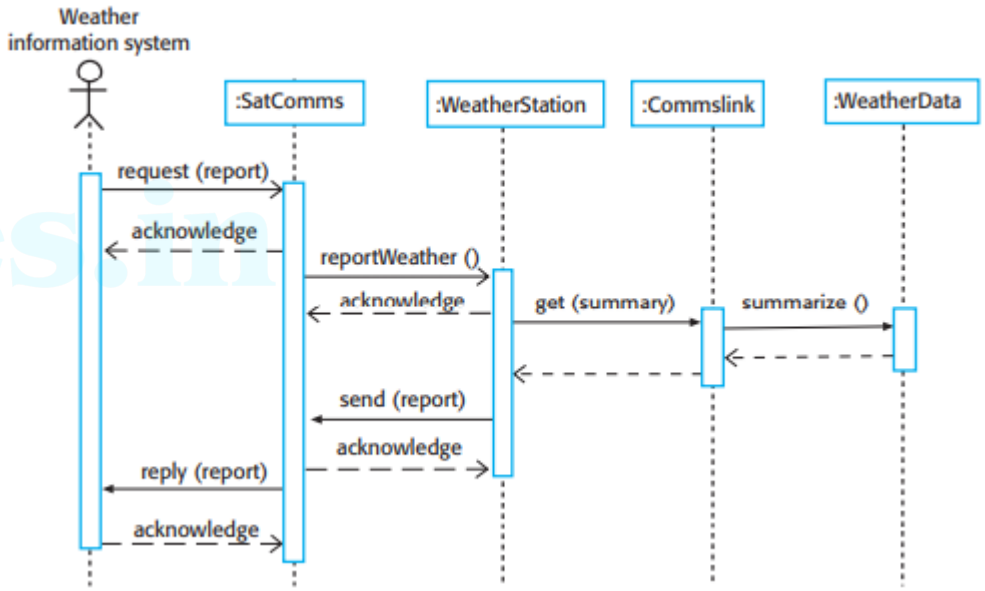
1. **Subsystem models**, which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.

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

3. **State machine models**, which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic model

Design models

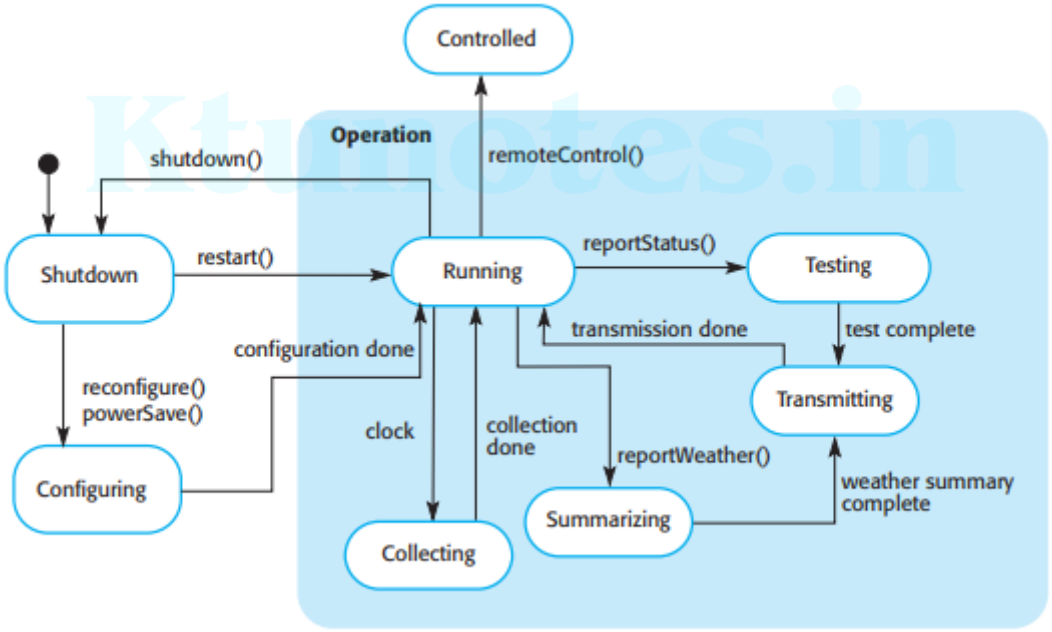
- Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place.
- When documenting a design, you should produce a sequence model for each significant interaction.
- If you have developed a use case model, then there should be a sequence model for each use case that you have identified



Sequence diagram describing data collection

Design models

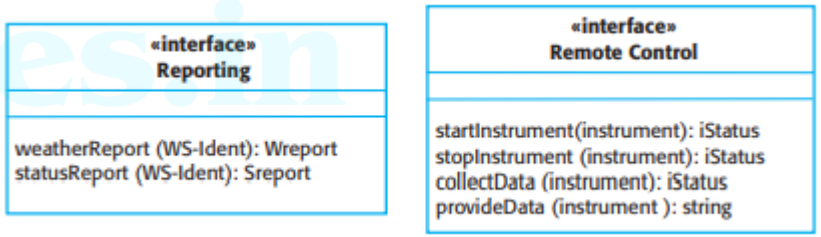
- Weather station state diagram



Interface specification

- An important part of any design process is the specification of the interfaces between the components in the design.
- We need to specify interfaces so that objects and subsystems can be designed in parallel.
- Once an interface has been specified, the developers of other objects may assume that interface will be implemented.
- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects.
- Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL).
- Details of the data representation should not be included in an interface design, as attributes are not defined in an interface specification. However, operations to access and update data should be included.
- As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable.

- There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects “implement” interfaces.
- Equally, a group of objects may all be accessed through a single interface.



- Figure shows two interfaces that may be defined for the weather station. The lefthand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object.

Design patterns

Design patterns

- The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification.
- Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used.
- Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is one that is equally applicable to any kind of software design.

Design patterns

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.

2. A description of the problem area that explains when the pattern may be applied.

3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.

4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

- Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

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 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.

Design patterns

- The above pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in the below figure, which shows two different graphical presentations of the same dataset.

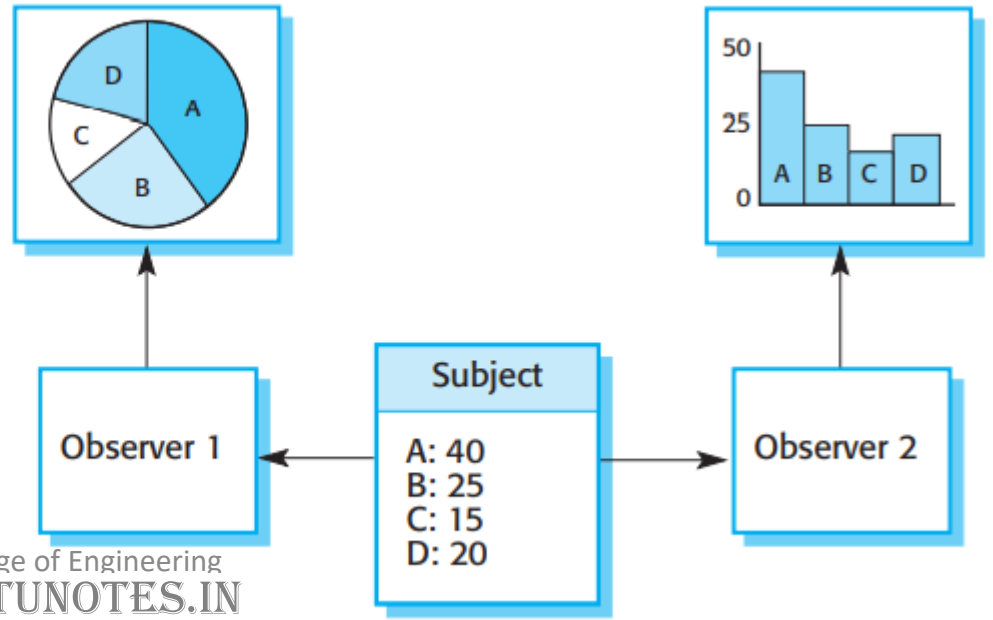
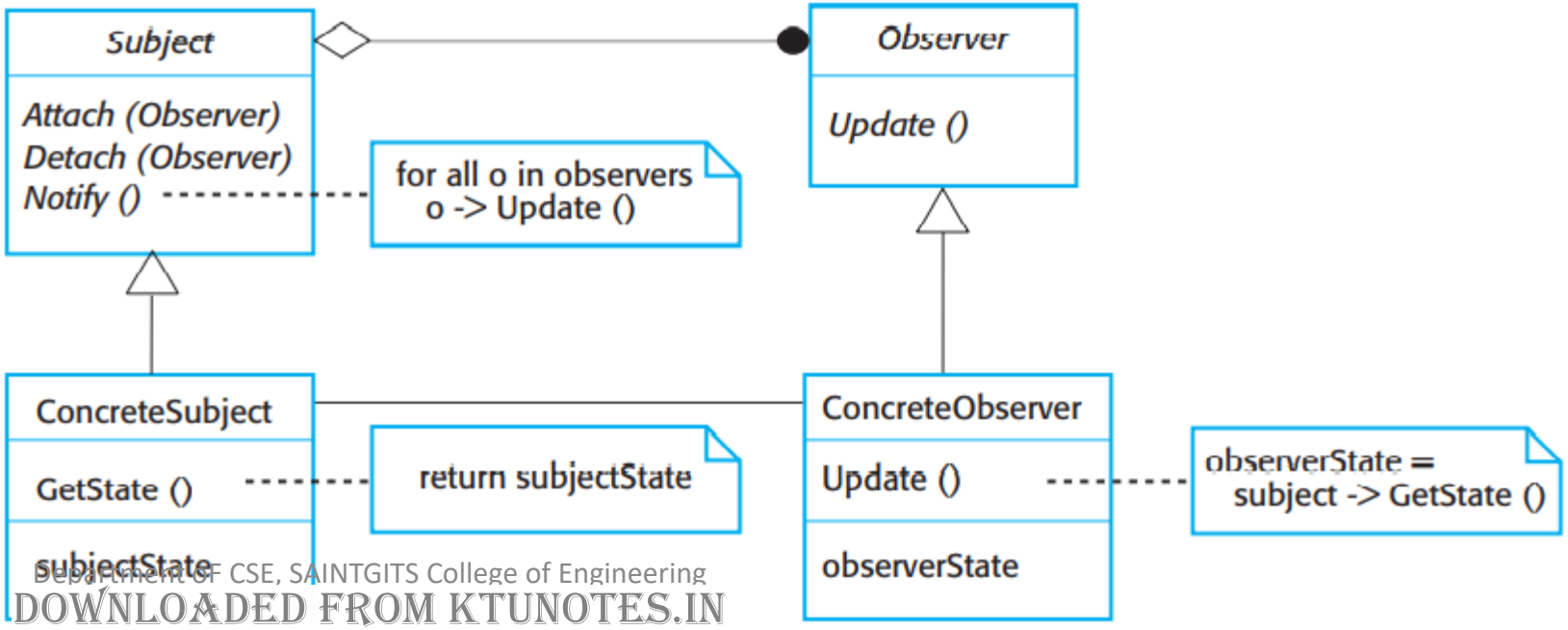


Figure 7.11 Multiple displa

Design patterns

- 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. Figure shows the representation in UML of the Observer pattern



Design patterns

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).

2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).

4. Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

Design patterns

- Patterns support high-level, concept reuse.
- Using patterns means that you reuse the ideas but can adapt the implementation to suit the system you are developing.
- Patterns are a great idea, but you need experience of software design to use them effectively.

Implementation issues

Reuse

Configuration management

Host-target development

Implementation issues

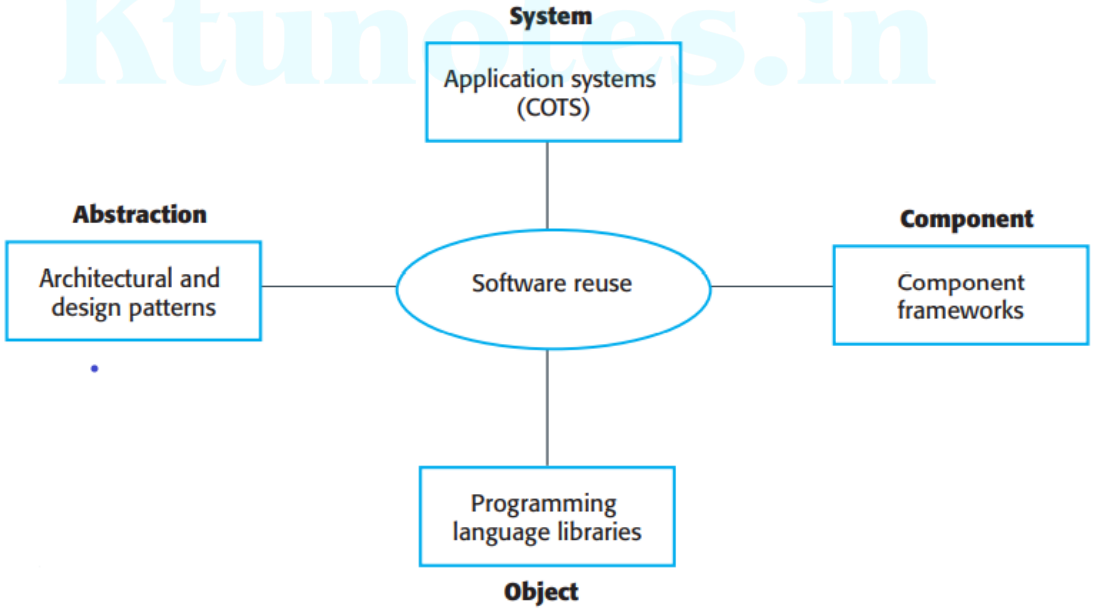
- system implementation - create an executable version of the software.
- Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.
- Aspects of implementation that are particularly important to software engineering:
  - Reuse
  - Configuration management
  - Host-target development

Reuse

- Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
- The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering

Reuse

- Software reuse is possible at a number of different levels, as shown in Figure



Reuse

Different levels of software reuse:

1. The abstraction level: At this level, we don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns are ways of representing abstract knowledge for reuse.

2. The object level: At this level, we directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need

3. The component level: Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own.

4. The system level: At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system

Reuse

costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-the shelf systems, these costs can be very high.

3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

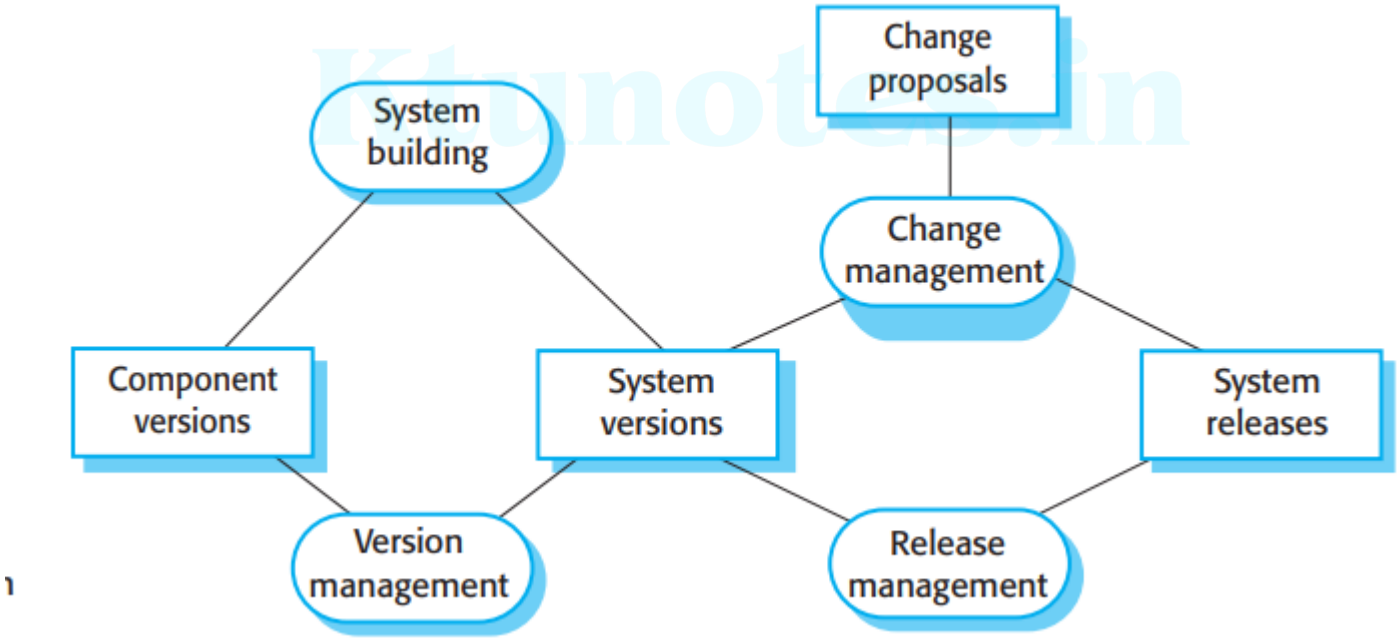
4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

Configuration management

- During the development process, many different versions of each software component are created.
- If you don’t keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.
- change happens all the time, so change management is absolutely essential.
- When several people are involved in developing a software system, you have to make sure that team members don’t interfere with each other’s work.
- also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done.
- **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

- Four fundamental configuration management activities:



Configuration management

Four fundamental configuration management activities:

1. **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. They stop one developer from overwriting code that has been submitted to the system by someone else.

2. **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.

3. **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.

4. **Release management**, where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Configuration management

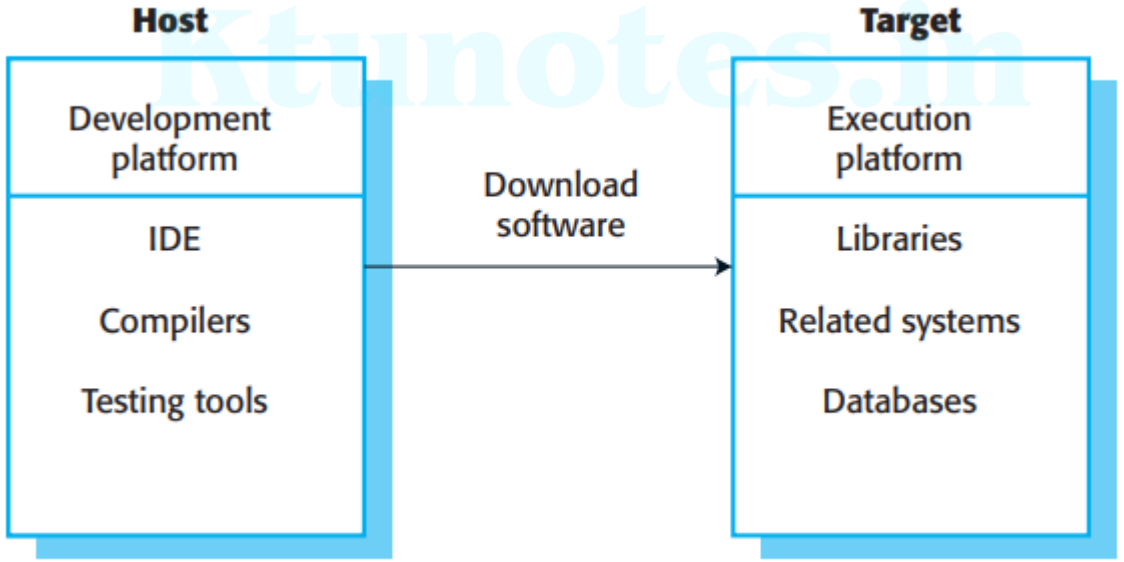
- Software configuration management tools support each of the above activities.
- These tools are usually installed in an integrated development environment, such as Eclipse.
- Version management may be supported using a version management system such as Subversion or Git ,which can support multi-site, multi-team development.
- System integration support may be built into the language or rely on a separate toolset such as the GNU build system.
- Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed

Host-target development

- Production software does not usually execute on the same computer as the software development environment.
- Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

Host-target development

- Figure : Host target development
- Most professional software development is based on a host-target model.



Host-target development

- Software is developed on one computer (the host) but runs on a separate machine (the target).
- More generally, we can talk about a development platform (host) and an execution platform (target). 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.
- Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another.
- However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.
- Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware.
- However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures. If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software.
- It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system

Host-target development

A software development platform should provide a range of tools to support software engineering processes.

- These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers .

Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are now usually installed within an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface. Generally, IDEs are created to support development in a specific programming language such as Java.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org).

Host-target development

- you need to make decisions about how the developed software will be deployed on the target platform. For distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. The hardware and software requirements of a component: If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
2. The availability requirements of the system: High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
3. Component communications: If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators and its real-time operating system.

Open-source development

Open source development

Open source licensing

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.
- There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code. 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. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality.
- However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.
- Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

- It is usually cheap or even free to acquire open-source software.
- You can normally download open-source software without charge.
- However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low.
- The other key benefit of using open-source products is that widely used open-source systems are very reliable.
- They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system.
- Bugs are discovered and repaired more quickly than is usually possible with proprietary software

- For a company involved in software development, there are two open-source issues that have to be considered:

1. Should the product that is being developed make use of open-source components?

2. Should an open-source approach be used for its own software development

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical.

If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems.

However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option.

You may have to integrate your software with existing systems that are incompatible with available pen-source systems. Even then, however, it could be quicker and cheaper to modify the open-source system rather than redevelop the functionality that you need.

Many software product companies are now using an open-source approach to development, especially for specialized systems

Open-source licensing

- Although 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.
- Legally, the developer of the code (either a company or an individual) owns the code.
- They can place restrictions on how it is used by including legally binding conditions in an open-source software license .
- Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source.
- Others are willing to allow their code to be used without this restriction.
- The developed systems may be proprietary and sold as closed-source systems

Open-source licensing

- Most open-source licenses are variants of one of three general models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This 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. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal 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. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Open-source licensing

- Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source.
- If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open source software in its development.
- If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem.
- However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions

Open-source licensing

companies managing projects that use open source should:

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

Review Techniques

- Cost impact of Software Defects
- DEFECT AMPLIFICATION AND REMOVAL
- REVIEW METRICS AND THEIR USE
- REVIEWS : A FORMALITY SPECTRUM
- INFORMAL REVIEWS
- FORMAL TECHNICAL REVIEWS

- software reviews are a “filter” for the software process.
- That is, 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:

1. Point out needed improvements in the product of a single person or team;
2. Confirm those parts of a product in which improvement is either not desired or not needed.
3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

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 (or to another framework activity in the software process).
- **Error** - to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end user (or to another framework activity in the software process).
- If software process improvement is considered, a quality problem that is propagated from one process framework activity (e.g., modeling) to another (e.g., construction) can also be called a “defect” (because the problem should have been found before a work product (e.g., a design model) was “released” to the next activity.
- 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.
- A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process.
- However, 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.
- The model is illustrated schematically in Figure .
- 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 20.1

Defect amplification model

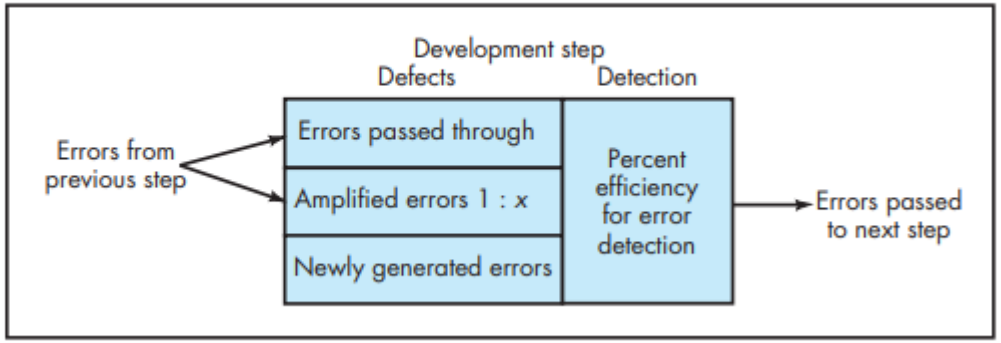


FIGURE 20.2

Defect amplification—no reviews

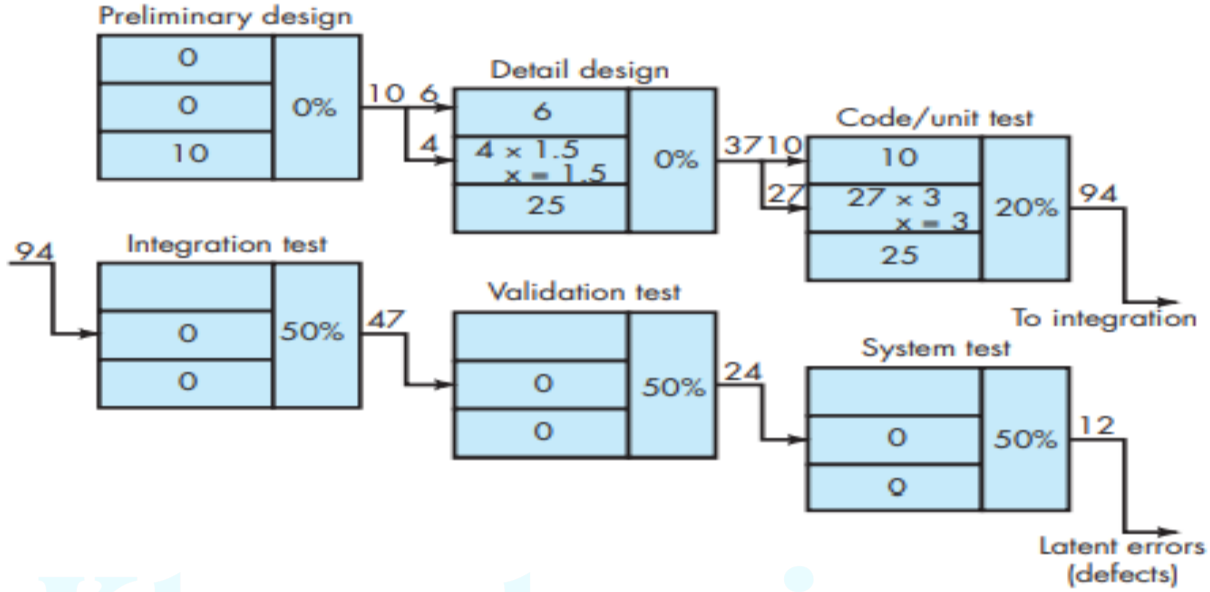
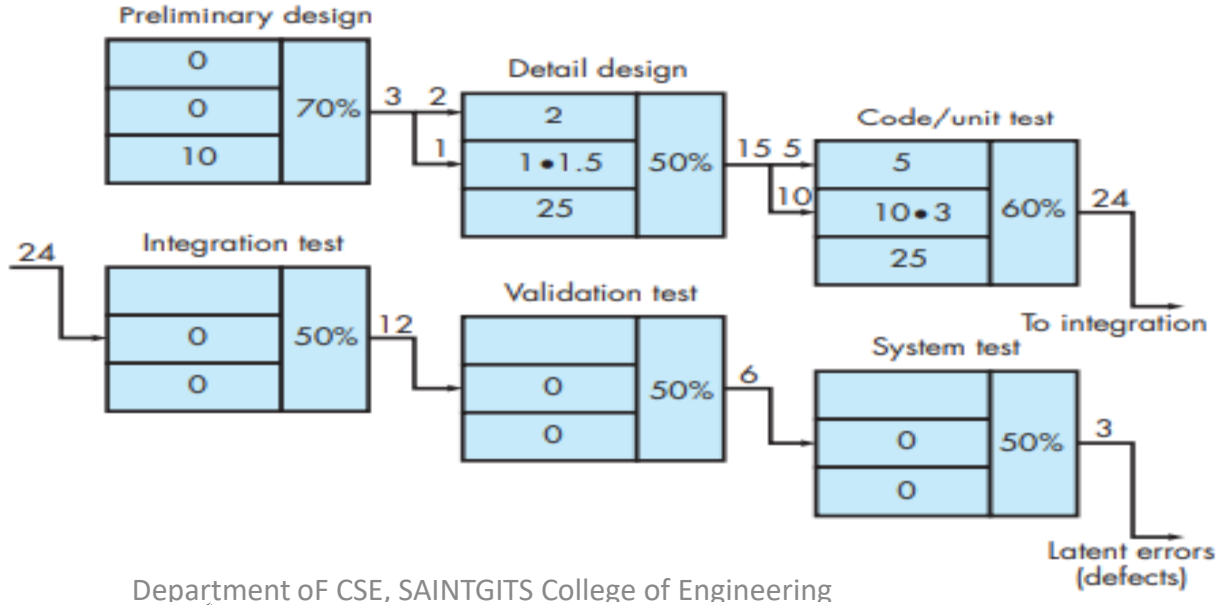


FIGURE 20.3

Defect amplification—reviews conducted



REVIEW METRICS AND THEIR USE

Analyzing Metrics

Cost-Effectiveness of Reviews

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.

REVIEW METRICS AND THEIR USE

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

- Preparation effort, Ep—the effort (in person-hours) required to review a work product prior to the actual review meeting
- Assessment effort, Ea— the effort (in person-hours) that is expended during the actual review
- Rework effort, Er — 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, Errminor—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- Major errors found, Errmajor—the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

These metrics can be further refi ned by associating the type of work product that was reviewed for the metrics collected.

Analyzing Metrics

- Before analysis can begin, a few simple computations must occur. The total review effort and the total number of errors discover.

$$E_{\text{review}} = E_p + E_a + E_r$$

$$\text{Err}_{\text{tot}} = \text{Err}_{\text{minor}} + \text{Err}_{\text{major}}$$

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

$$\text{Error density} = \frac{\text{Err}_{\text{tot}}}{\text{WPS}}$$

- If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases), the percentage of errors uncovered for each review can be computed against the total number of errors found for all reviews.
- In addition, the error density for each work product can be computed. Once data are collected for many reviews conducted across many projects, average values for error density enable you to estimate the number of errors to be found in a new w as yet unreviewed document.
- Once testing has been conducted it is possible to collect additional error data, including the effort required to fi nd and correct errors uncovered during testing and the error density of the software. The costs associated with finding and correcting an error during testing can be compared to those for reviews.

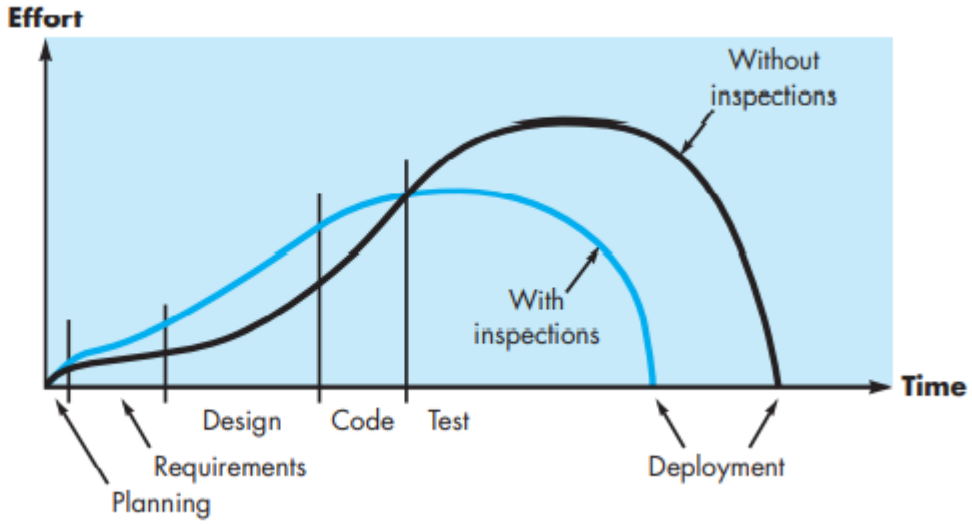
Cost-Effectiveness of Reviews

- 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 (via testing).
- More importantly, industry data for software reviews has been collected for more than two decades and is summarized qualitatively using the graphs illustrated in Figure.

FIGURE 20.4

Effort expended with and without reviews

Source: Adapted from [Fag86].



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.
- Figure below depicts a reference model for technical reviews that identifies four characteristics that contribute to the formality with which a review is conducted.

FIGURE 20.5

Reference model for technical reviews

REVIEWS : A FORMALITY SPECTRUM

- Each of the reference model characteristics helps to define the level of review formality.
- The formality of a review increases when

(1) distinct roles are explicitly defined for the reviewers,

(2) there is a sufficient amount of planning and preparation for the review,

(3) a distinct structure for the review (including tasks and internal work products) is defined, and

(4) follow-up by the reviewers occurs for any corrections that are made.

REVIEWS : A FORMALITY SPECTRUM

- two broad categories of technical reviews:
  - **informal reviews and**
  - **more formal technical reviews.**

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, or the review-oriented aspects of pair programming.
- A simple desk check or a casual meeting conducted with a colleague is a review.
- However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches.
- But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.
- 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.
- The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product.
- Any errors or issues noted by the reviewers are recorded by the designer for resolution at a later time.
- Desk checks may be scheduled in an ad hoc manner, or they may be mandated as part of good software engineering practice.
- In general, the amount of material to be reviewed is relatively small and the overall time spent on a desk check span little more than one or two hours.
- **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 benefi t is immediate discovery of errors and better work product quality as a consequence.

INFORMAL REVIEWS

Example: a desk check of the interface prototype for SafeHomeAssured.com.

Rather than simply playing with the prototype at the designer's workstation, the designer and a colleague examine the prototype using a 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?

And so on

FORMAL TECHNICAL REVIEWS

The Review Meeting

Review Reporting and Record Keeping

Review Guidelines

Sample-Driven Reviews

FORMAL TECHNICAL REVIEWS[FTR]

- A formal technical review (FTR) is a software quality control activity performed by software engineers (and others).
- The objectives of an FTR are:

  (1) to uncover errors in function, logic, or implementation for any representation of the software;

  (2) to verify that the software under review meets its requirements;

  (3) to ensure that the software has been represented according to predefined standards;

  (4) to achieve software that is developed in a uniform manner; and

  (5) to make projects more manageable. In addition, the 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

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.
- Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components.

The Review Meeting

- By narrowing the focus, the FTR has a higher likelihood of uncovering errors.
- 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 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.
- The project leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.
- 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 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 occurs, the recorder notes each.

The Review Meeting

- At the end of the review, all attendees of the FTR must decide whether to:

(1) accept the product without further modification,

(2) reject the product due to severe errors (once corrected, another review must be performed), or

(3) 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 fi ndings.

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:

1. What was reviewed?

2. Who reviewed it?

3. What were the findings and conclusions?

The review summary report is a single-page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The review issues list serves two purposes:

(1) to identify problem areas within the product and

(2) 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.

You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow-up to the review leader.

Review Guidelines

- Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse than no review at all.
- The following represents a minimum set of guidelines for formal technical reviews:

1. Review the product, not the producer. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.

2. Set an agenda and maintain it. One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.

3. Limit debate and rebuttal. When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.

4. Enunciate problem areas, but don't attempt to solve every problem noted. A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.

5. Take written notes. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded. Alternatively, notes may be entered directly into a notebook computer.

Review Guidelines

6. Limit the number of participants and insist upon advance preparation. Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).

7. Develop a checklist for each product that is likely to be reviewed. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even testing work products.

8. Allocate resources and schedule time for FTRs. For reviews to be effective, they should be scheduled as a task during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.

9. Conduct meaningful training for all reviewers. To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg estimate a one-month learning curve for every 20 people who are to participate effectively in reviews.

10. Review your early reviews. Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

Sample-Driven Reviews

- In an ideal setting, every software-engineering work product would undergo a formal technical review.
- In the real word of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized.
- Thelin and his colleagues suggest **a sample-driven review process** in which samples of all software engineering work products are inspected to determine which work products are most error prone.
- Full FTR resources are then focused only on those work products that are likely (based on data collected during sampling) to be error prone.
- To be effective, the sample-driven review process must attempt to quantify those work products that are primary targets for full FTRs.
- To accomplish this, the following steps are suggested:

1. Inspect a fraction a_i of each software work product i. Record the number of faults f_i found within a_i.
2. Develop a gross estimate of the number of faults within work product i by multiplying f_i by $1/a_i$.
3. Sort the work products in descending order according to the gross estimate of the number of faults in each.
4. Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must be representative of the work product as a whole and large enough to be meaningful to the reviewers who do the sampling. As ai increases, the likelihood that the sample is a valid representation of the work product also increases.

However, the resources required to do sampling also increase. A software engineering team must establish the best value for ai for the types of work products produced.

POST-MORTEM EVALUATIONS

- Many lessons can be learned if a software team takes the time to evaluate the results of a software project after the software has been delivered to end users.
- Baaz and his colleagues suggest the use of a **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 (that is, achievements and positive experiences) and challenges (problems and negative experiences)” .
- Often conducted in a workshop format, 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.

Module 3(part 2)

Object-oriented design using the UML, Design patterns, Implementation issues, Open-source development - Open-source licensing - GPL, LGPL, BSD. Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. Informal Review, Formal Technical Reviews, Post-mortem evaluations. **Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing, Debugging, White box testing, Path testing, Control Structure testing, Black box testing, Testing Documentation and Help facilities. Test automation, Test-driven development, Security testing. Overview of DevOps and Code Management - Code management, DevOps automation, Continuous Integration, Delivery, and Deployment (CI/CD/CD).** Software Evolution - Evolution processes, Software maintenance.

SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

SOFTWARE TESTING STRATEGIES

- A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.
- Therefore, any testing strategy must incorporate test planning, test-case design, test execution, and resultant data collection and evaluation. A software testing strategy should be flexible enough to promote a customized testing approach.
- At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses

A STRATEGIC APPROACH TO SOFTWARE TESTING

Verification and Validation

Organizing for Software Testing

Software Testing Strategy—The Big Picture

Criteria for Completion of Testing

A STRATEGIC APPROACH TO SOFTWARE TESTING

- Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specifi c test-case design techniques and testing methods—should be defi ned for the software process

A STRATEGIC APPROACH TO SOFTWARE TESTING

Generic 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.

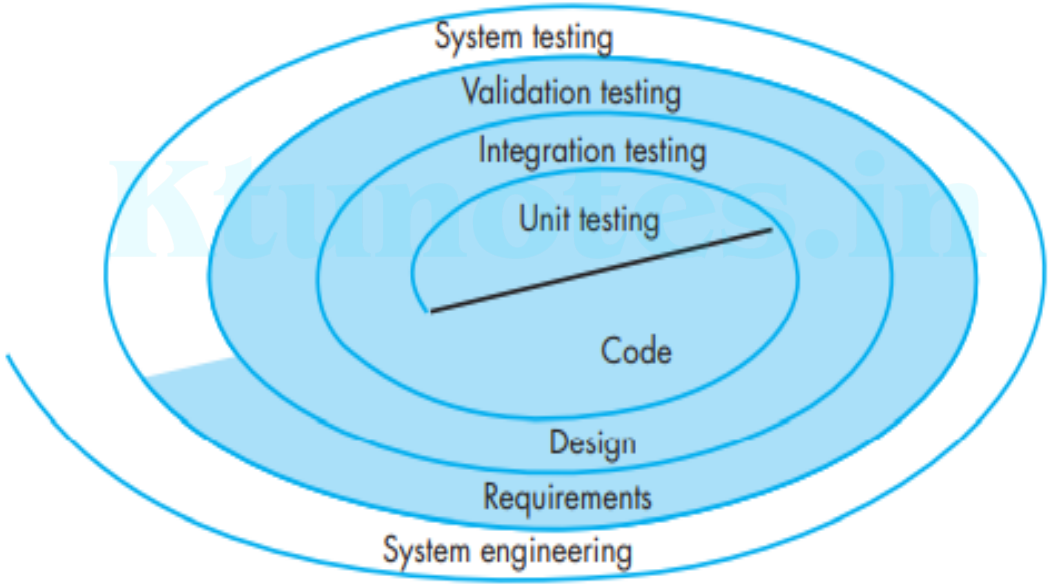
Verification and Validation

- Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).
- 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.
- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

Organizing for Software Testing

- The **software developer** is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed.
- In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture.
- Only after the software architecture is complete does an independent test group become involved.
- The role of an **independent test group (ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built.
- Independent testing removes the conflict of interest that may otherwise be present.
- The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.
- While testing is conducted, the developer must be available to correct errors that are uncovered.

Software Testing Strategy—The Big Picture

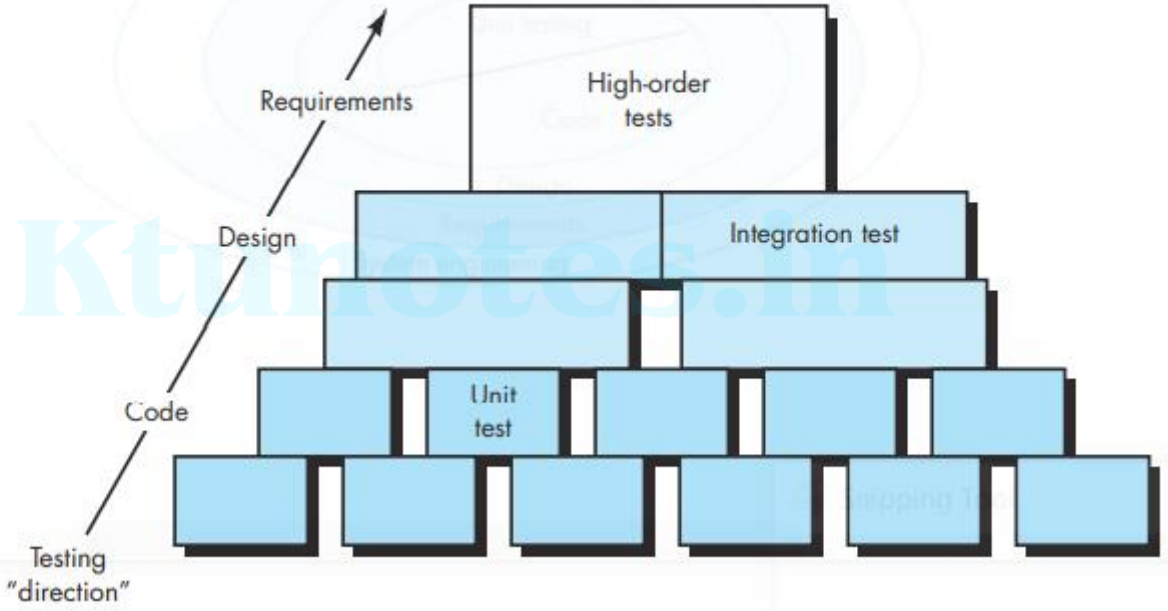


Software Testing Strategy—The Big Picture

- Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modeling are validated against the software that has been constructed.
- Finally, you arrive at system testing, where the software and other system elements are tested as a whole.
- To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.

FIGURE 22.2

Software testing steps



- “You're never done testing; the burden simply shifts from you (the software engineer) to the end user.”
- Every time the user executes a computer program, the program is being tested.
- “You’re done testing when you run out of time or you run out of money.”

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

Unit Testing

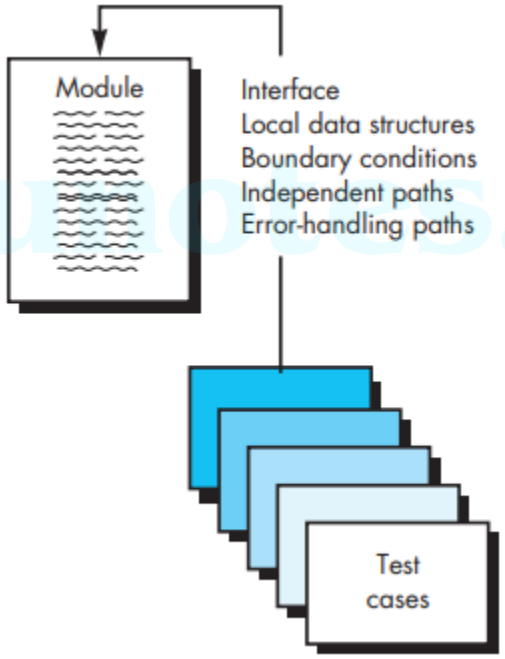
Unit Testing

- 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 Test Considerations.

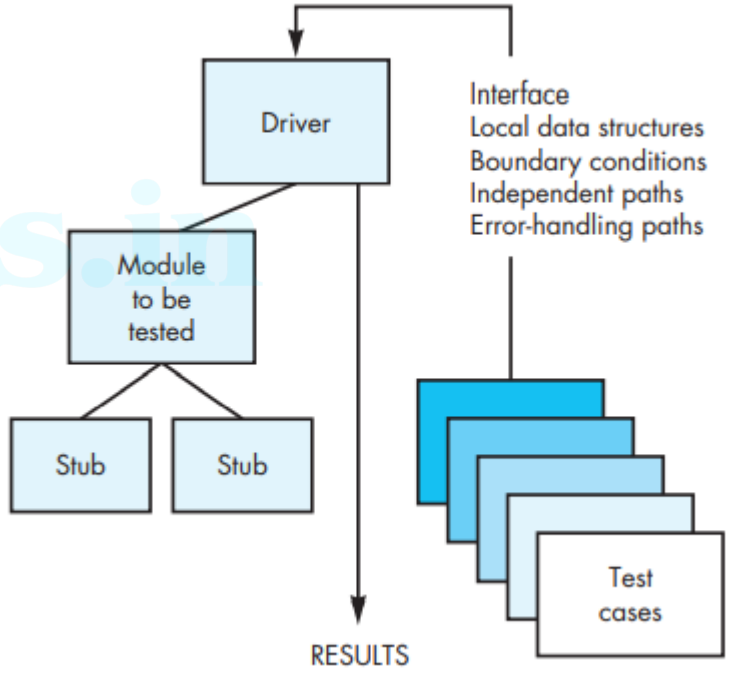
- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- 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.
- In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Unit Test Considerations.



Unit-Test Procedures.

- The design of unit tests can occur before coding begins or after source code has been generated.
- Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.
- The unit test environment is illustrated in Figure
- In most applications a driver is nothing more than a “main program” that accepts test-case data, passes such data to the component (to be tested), and prints relevant results.
- Stubs serve to replace modules that are subordinate (invoked by) the component to be tested.
- A stub or “dummy subprogram” uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent testing “overhead.” That is, both are software that must be coded (formal design is not commonly applied) but that is not delivered with the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).



Integration Testing

Top-Down Integration

Bottom-Up Integration

Regression Testing

Smoke Testing

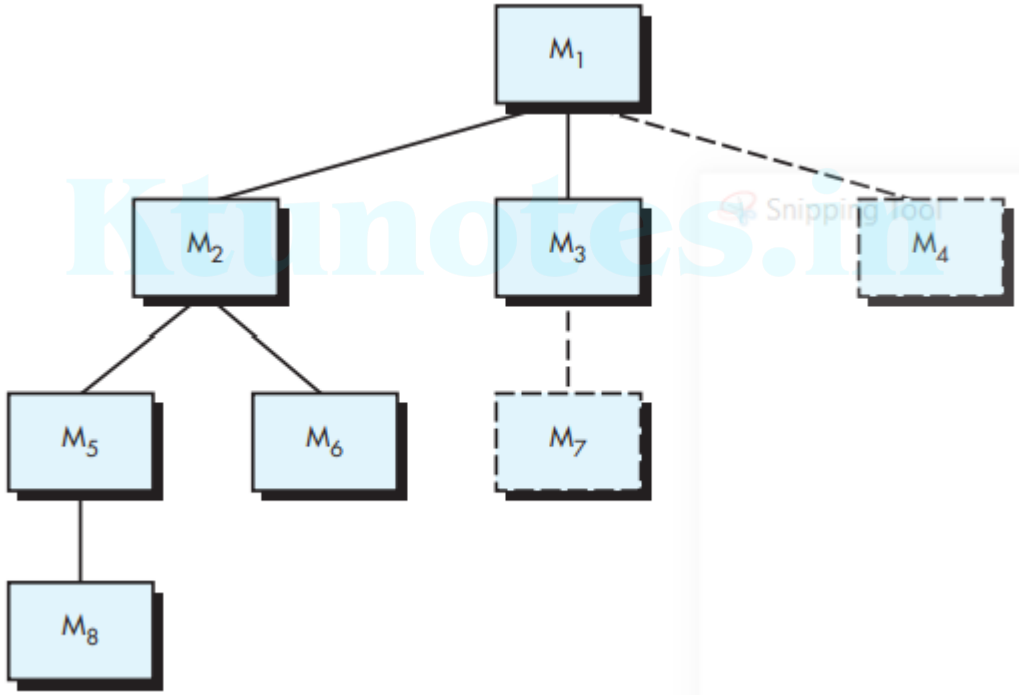
Integration Testing

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that has been dictated by design.
- Non incremental integration -to construct the program using a “big bang” approach. All components are combined in advance and the entire program is tested as a whole. Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- 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.

Top-Down Integration

- .Top-down integration testing is 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 (and ultimately subordinate) to the main control module are incorporated into the structure in either a depthfirst or breadth-first manner.

Top-Down Integration



Top-Down Integration

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

- The top-down integration strategy verifies major control or decision points early in the test process.

Bottom-Up Integration

- Bottom-up integration testing, as its name implies, 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.

A bottom-up integration strategy may be implemented with the following steps:

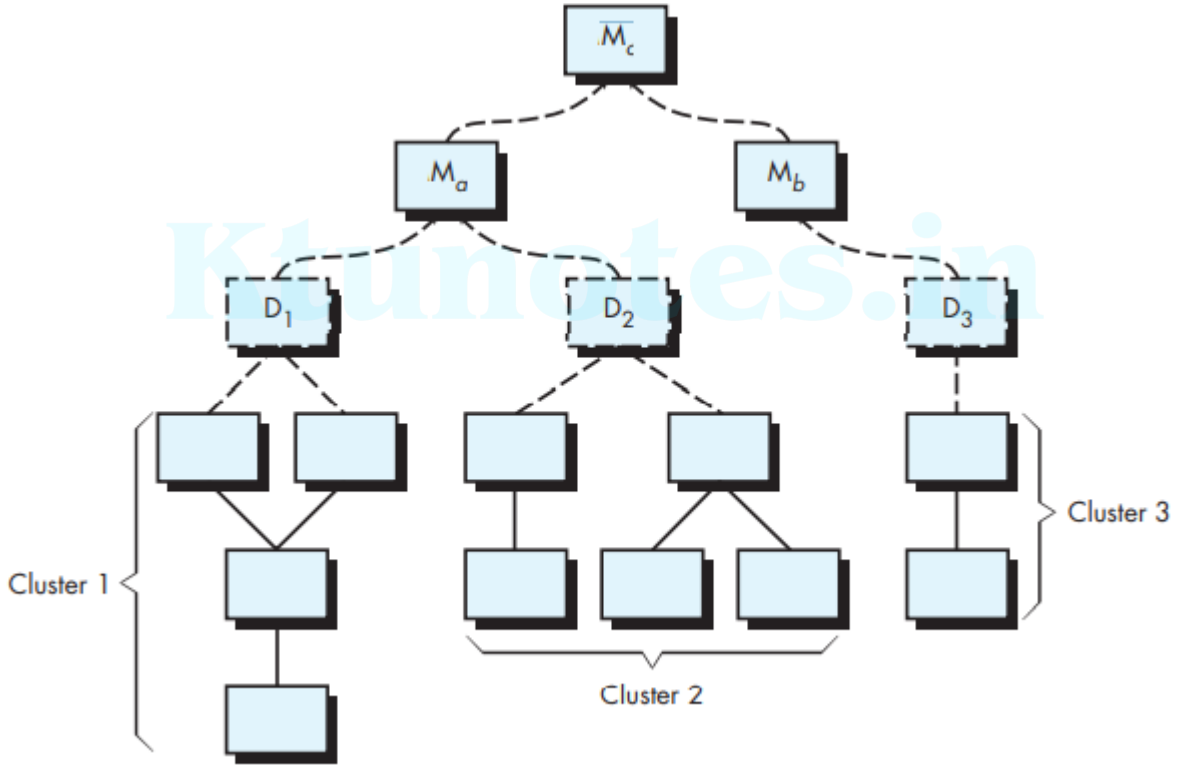
1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test-case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Bottom-Up Integration.



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 reexecution 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.

Regression Testing

- Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools.
- Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- ❑ A representative sample of tests that will exercise all software functions.
- ❑Additional tests that focus on software functions that are likely to be affected by the change.
- ❑Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Smoke 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.

In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily.

Smoke testing provides a number of benefi ts when it is applied on complex, time-critical software projects:

- Integration risk is minimized
- The quality of the end product is improved.
- Error diagnosis and correction are simplified.
- Progress is easier to assess.

Validation testing

- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.
- At the validation or system level, the distinction between different software categories disappears.
- Testing focuses on user-visible actions and user-recognizable output from the system.
- Validation succeeds when software functions in a manner that can be reasonably expected by the customer

Validation testing

Validation-Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
- If a deviation from specification is uncovered, a deficiency list is created.
- A method for resolving deficiencies (acceptable to stakeholders) must be established.

Configuration Review

- An important element of the validation process is a configuration review.
- The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities.
- The confi guration reviewis sometimes called an audit.

Validation testing

- Alpha and Beta Testing
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

Alpha Tests

- The alpha test is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
- Alpha tests are conducted in a controlled environment.

Beta Tests

- The beta test is conducted at one or more end-user sites
- Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

Customer Acceptance Testing

- A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract.
- The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SYSTEM TESTING

- A classic system-testing problem is “finger pointing.” This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem.
- Rather than indulging in such nonsense, you should anticipate potential interfacing problems and

(1) design error-handling paths that test all information coming from other elements of the system,

(2) conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) record the results of tests to use as “evidence” if finger pointing does occur, and

(4) participate in planning and design of system tests to ensure that software is adequately tested.

SYSTEM TESTING

Recovery testing

- Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security testing

- Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."
- Given enough time and resources, good security testing will ultimately penetrate a system.
- The role of the system designer is to make penetration cost more than the value of the information that will be obtained

SYSTEM TESTING

Stress Testing

- Stress tests are designed to confront programs with abnormal situations.
- In essence, the tester who performs stress testing asks: “How high can we crank this up before it fails?”
- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- For example, (1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created.
- Essentially, the tester attempts to break the program.
- A variation of stress testing is a technique called **sensitivity testing.** In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing

SYSTEM TESTING

Performance Testing

- Performance testing is designed to test the run-time performance of software within the context of an integrated system.
- Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.
- However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.
- That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.
- External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis.
- By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

SYSTEM TESTING

Deployment Testing

- Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Debugging

The Debugging Process
Psychological Considerations
Debugging Strategies
Correcting the Error

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

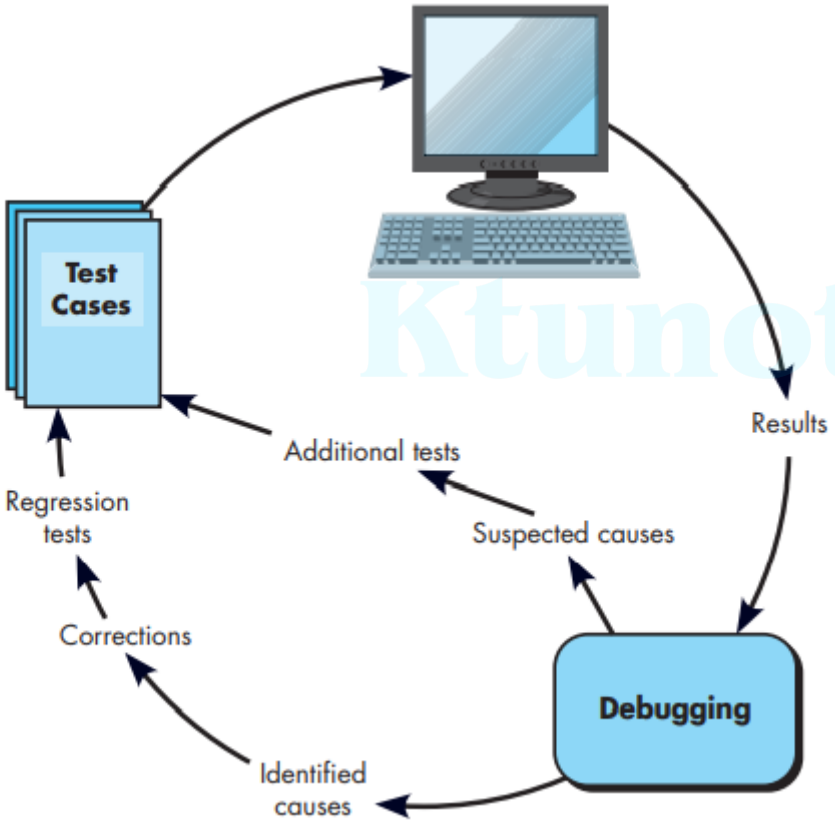
- The debugging process begins with the execution of a test case.
- Results are assessed and a lack of correspondence between expected and actual performance is encountered.
- In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden.
- The debugging process attempts to match symptom with cause, thereby leading to error correction.
- The debugging process will usually have one of two outcomes:

  (1) the cause will be found and corrected or

  (2) the cause will not be found.

In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

The Debugging Process



The Debugging Process

Why is debugging so difficult?

1. The symptom and the cause may be geographically remote.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. 8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to fi nd the cause also increases.

Psychological Considerations

- Debugging process is an innate human trait. Some people are good at it and others aren’t.
- Large variances in debugging ability have been reported for programmers with the same education and experience.

SafeHome

Debugging

The scene: Ed's cubical as code and unit testing is conducted.

The players: Ed and Shakira—members of the *SafeHome* software engineering team.

The conversation:

Shakira (looking in through the entrance to the cubical): Hey . . . where were you at lunchtime?

Ed: Right here . . . working.

Shakira: You look miserable . . . what's the matter?

Ed (sighing audibly): I've been working on this bug since I discovered it at 9:30 this morning and it's what, 2:45 . . . I'm clueless.

Shakira: I thought we all agreed to spend no more than one hour debugging stuff on our own; then we get help, right?

Ed: Yeah, but . . .

Shakira (walking into the cubical): So what's the problem?

Ed: It's complicated, and besides, I've been looking at this for, what, 5 hours. You're not going to see it in 5 minutes.

Shakira: Indulge me . . . what's the problem?

[Ed explains the problem to Shakira, who looks at it for about 30 seconds without speaking, then . . .]

Shakira (a smile is gathering on her face): Uh, right there, the variable named *setAlarmCondition*. Shouldn't it be set to "false" before the loop gets started?

[Ed stares at the screen in disbelief, bends forward, and begins to bang his head gently against the monitor. Shakira, smiling broadly now, stands and walks out.]

Debugging Strategies

Three debugging strategies have been proposed :

- brute force
- backtracking
- cause elimination.

Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging Strategies

Brute force

- The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- You apply brute force debugging methods when all else fails.
- Using a “let the computer find the error” philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you’ll find a clue that can lead us to the cause of an error.
- Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time.

Backtracking

- Backtracking is a fairly common debugging approach that can be used successfully in small programs.
- Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found.
- Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

Cause elimination

- The third approach to debugging— cause elimination—is manifested by induction or deduction and introduces the concept of binary partitioning.
- Data related to the error occurrence are organized to isolate potential causes.
- A “cause hypothesis” is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Debugging Strategies

Automated Debugging

- Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.
- Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation."
- A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available.
- However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

Debugging Strategies

- **The People Factor.**

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people!

Correcting the Error

- Once a bug has been found, it must be corrected.
- Three simple questions that you should ask before making the “correction” that removes the cause of a bug.
  - Is the cause of the bug reproduced in another part of the program?
  - What “next bug” might be introduced by the fi x I'm about to make?
  - What could we have done to prevent this bug in the first place?

Testing

- Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer

SOFTWARE TESTING FUNDAMENTALS

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or a product with “testability” in mind.
- The **tests** themselves **must exhibit a set of characteristics** that achieve the goal of finding the most errors with a minimum of effort.
  - Testability.
  - Operability.
  - Observability
  - Controllability.
  - Decomposability.
  - Simplicity.
  - Stability.
  - Understandability.

Test Characteristics

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex.

INTERNAL AND EXTERNAL VIEWS OF TESTING

- Any engineered product can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

- The first test approach takes an external view and is called **black-box testing**. The second requires an internal view and is termed **white-box testing.**

- **Black-box testing** alludes to tests that are conducted at the **software interface**. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
- White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.
- At first glance it would seem that very thorough **white-box testing** would lead to "100 percent correct programs." All we need do is **define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively.**
- Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large.
- White-box testing should not, however, be dismissed as impractical. **A limited number of important logical paths can be selected and exercised**. Important data structures can be probed for validity.

WHITE-BOX TESTING

BASIS PATH TESTING

CONTROL STRUCTURE TESTING

WHITE-BOX TESTING

- White-box testing, sometimes called glass-box testing or structural testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- Using white-box testing methods, you can derive test cases that

  (1) guarantee that all independent paths within a module have been exercised at least once,

  (2) exercise all logical decisions on their true and false sides,

  (3) execute all loops at their boundaries and within their operational bounds, and

  (4) exercise internal data structures to ensure their validity

BASIS PATH TESTING

Flow Graph Notation

Independent Program Paths

Deriving Test Cases

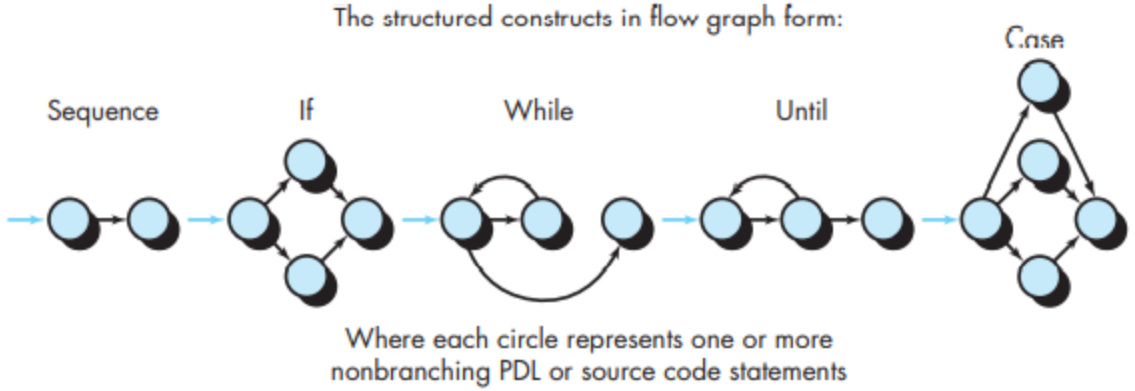
Graph Matrices

BASIS PATH TESTING

- Basis path testing is a white-box testing technique first proposed by Tom McCabe .
- The basis path method enables the test-case designer to derive a **logical complexity measure of a procedural design** and use this measure as a guide for **defining a basis set of execution paths**.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation

- a simple notation for the representation of control flow graph
- The flow graph depicts logical control flow using the notation illustrated in Figure. Each structured construct has a corresponding flow graph symbol.



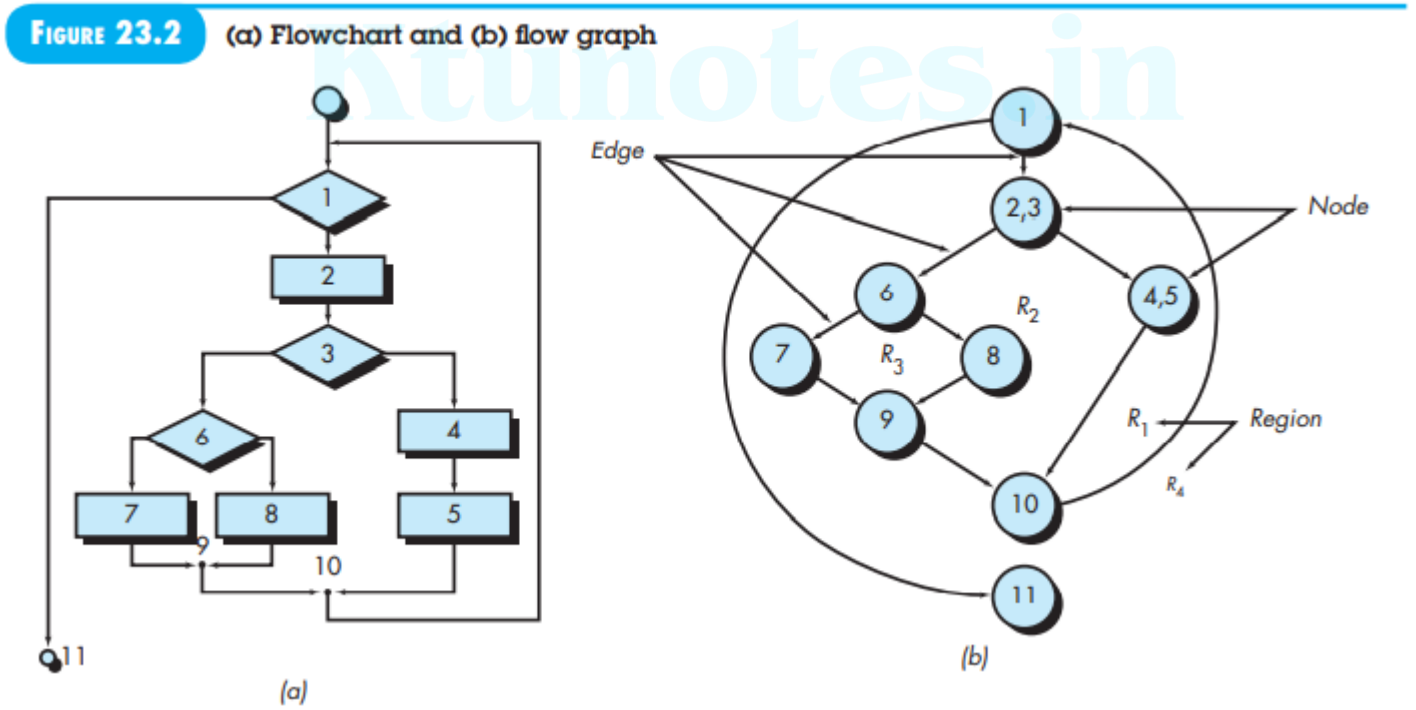
- Figure: flow graph notations

Flow Graph Notation

- Each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the fl ow graph symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions.
- When counting regions, we include the area outside the graph as a region.
- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.
- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.,.

Flow Graph Notation

- The program design language (PDL) segment translates into the flow graph shown.

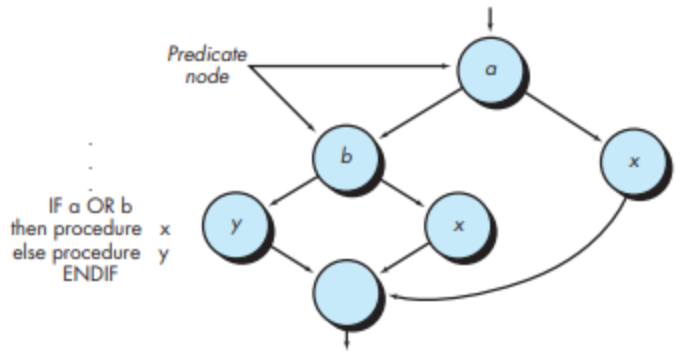


Flow Graph Notation

- A separate node is created for each of the conditions a and b in the statement ***IF a OR b***. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it

FIGURE 23.3

Compound logic



Independent Program Paths

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.
- For example, a set of independent paths for the flow graph illustrated in the Figure 23.2b is
- Path 1: 1-11
- Path 2: 1-2-3-4-5-10-1-11
- Path 3: 1-2-3-6-8-9-10-1-11
- Path 4: 1-2-3-6-7-9-10-1-11
- Note that each new path introduces a new edge.
- **The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specifi ed paths and does not traverse any new edges**
- Paths 1 through 4 constitute a basis set for the flow graph in Figure 23.2b . That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.
- It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

Independent Program Paths

- **Cyclomatic complexity** is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
- Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric.
- Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

   $V(G) = E - N + 2$

   where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

   where P is the number of predicate nodes contained in the flow graph G.

Independent Program Paths

- For flow graph in Figure 23.2b , the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. $V(G)$ = 11 edges − 9 nodes + 2 = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

- Therefore, the cyclomatic complexity of the flow graph in Figure 23.2bis 4.
- the value for V(G) provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

- The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

Deriving Test Cases- Example

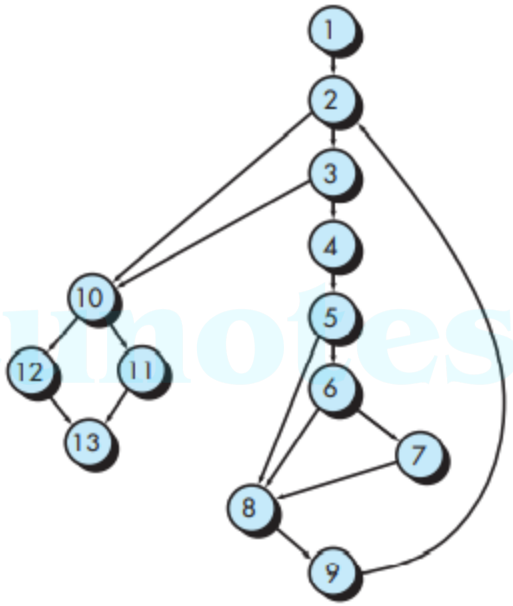
```
PROCEDURE average;

*  This procedure computes the average of 100 or fewer
   numbers that lie between bounding values; it also computes the
   sum and the total number valid.

   INTERFACE RETURNS average, total.input, total.valid;
   INTERFACE ACCEPTS value, minimum, maximum;

   TYPE value[1:100] IS SCALAR ARRAY;
   TYPE average, total.input, total.valid;
      minimum, maximum, sum IS SCALAR;
   TYPE i IS INTEGER;
1  i = 1;
   total.input = total.valid = 0;                    2
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
4     increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7                sum = s sum + value[i]
         ELSE skip
8     ENDIF
      increment i by 1;
9  ENDDO
   IF total.valid > 0                                10
11    THEN average = sum / total.valid;
12    ELSE average = -999;
13 ENDIF
END average
```

PDL with nodes identified



Step 1: Flow graph for the procedure average

$V(G)$ = 6 regions
$V(G)$ = 17 edges − 13 nodes + 2 = 6
$V(G)$ = 5 predicate nodes + 1 = 6

Step 2: Determine V(G)

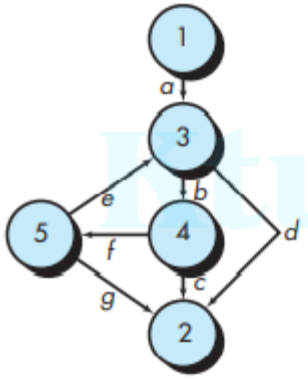
Path 1: 1-2-10-11-13
Path 2: 1-2-10-12-13
Path 3: 1-2-3-10-11-13
Path 4: 1-2-3-4-5-8-9-2-. . .
Path 5: 1-2-3-4-5-6-8-9-2-. . .
Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

Step 3: determine linearly independent paths

Graph Matrices

- The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization.
- A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

Graph Matrices



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices

- the graph matrix is nothing more than a tabular representation of a flow graph.
- However, by adding a **link weight to each matrix entry**, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow.
- In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:
  - The probability that a link (edge) will be executed.
  - The processing time expended during traversal of a link
  - The memory required during traversal of a link
  - The resources required during traversal of a link.

CONTROL STRUCTURE TESTING

- The basis path testing technique is one of a number of techniques for control structure testing.
- variations on control structure testing are there. These broaden testing coverage and improve the quality of white-box testing.
  - **Condition testing** is a test-case design method that exercises the logical conditions contained in a program module.
  - **Data flow testing** selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - **Loop testing** is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined:
    - simple loops
    - concatenated loops
    - nested loops
    - unstructured loops.

CONTROL STRUCTURE TESTING

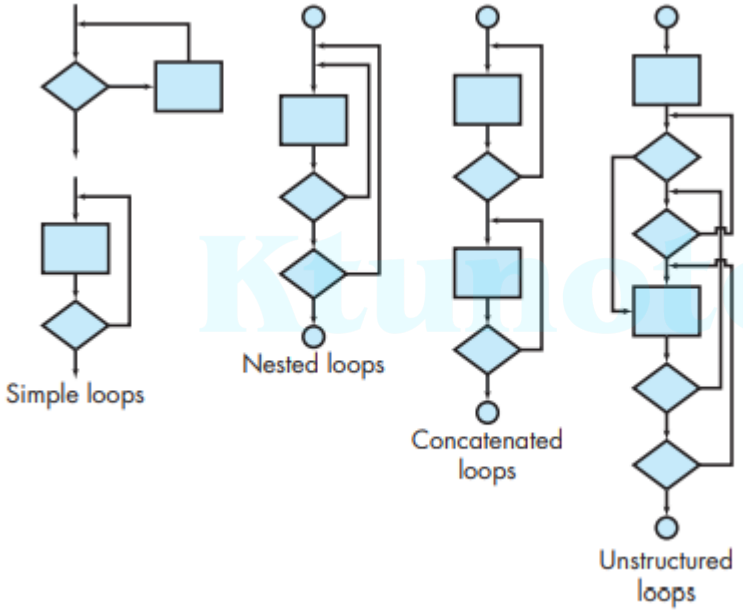


Figure: Classes of loops

CONTROL STRUCTURE TESTING

Simple Loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where $m < n$.
5. $n - 1, n, n + 1$ passes through the loop.

Nested Loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated Loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured Loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 14).

BLACK-BOX TESTING

Graph-Based Testing Methods
Equivalence Partitioning
Boundary Value Analysis
Orthogonal Array Testing

BLACK-BOX TESTING

- Black-box testing, also called behavioral testing or functional testing, focuses on the functional requirements of the software.
- 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 is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.
- Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions,

(2) interface errors,

(3) errors in data structures or external database access,

(4) behavior or performance errors, and

(5) initialization and termination errors.

BLACK-BOX TESTING

- Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing .
- Because black-box testing purposely disregards control structure, attention is focused on the information domain.
- Tests are designed to answer the following questions:
  - How is functional validity tested?
  - How are system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?
- By applying black-box techniques, you derive a set of test cases that satisfy the following criteria :

test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods

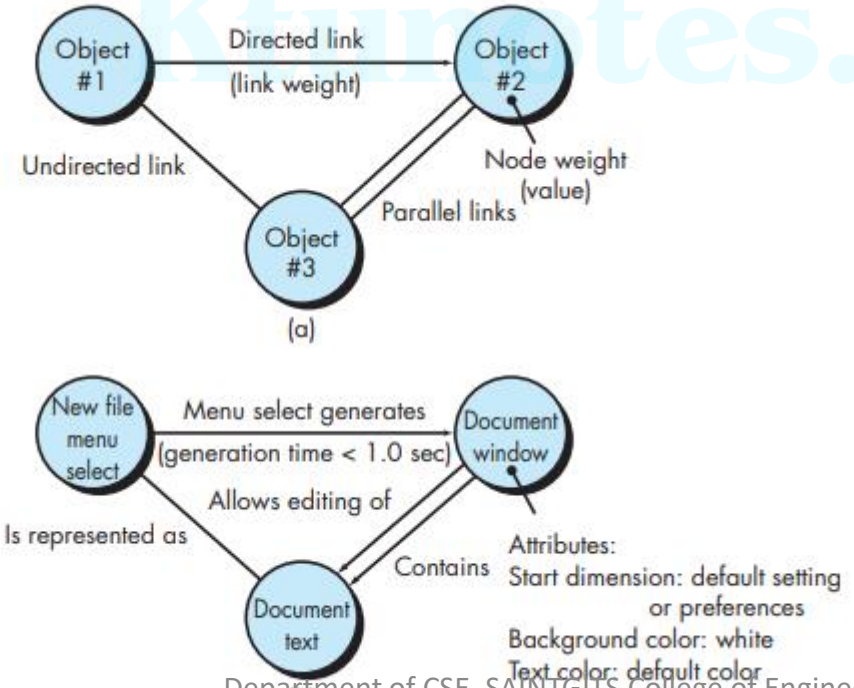
- The first step in black-box testing is to **understand the objects that are modeled in software and the relationships that connect these objects.**
- Once this has been accomplished, the next step is to define a series of tests that verify “all objects have the expected relationship to one another”
- Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- To accomplish these steps, you begin by **creating a graph—a** collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link.

Graph-Based Testing Methods

- The symbolic representation of a graph is shown in Figure.
- Nodes are represented as circles connected by links that take a number of different forms.
- A directed link (represented by an arrow) indicates that a relationship moves in only one direction.
- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions.
- Parallel links are used when a number of different relationships are established between graph nodes.

FIGURE 23.8

(a) Graph notation; (b) simple example



Graph-Based Testing Methods

- As a simple example, consider a portion of a graph for a word-processing application (Figure 23.8b) where

Object #1 = **newFile** (menu selection)
Object #2 = **documentWindow**
Object #3 = **documentText**

Graph-Based Testing Methods

Behavioral testing methods that can make use of graphs:

- **Transaction flow modeling**. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps.
- **Finite state modeling**. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state . The state diagram can be used to assist in creating graphs of this type.
- **Data flow modeling**. The nodes are data objects, and the links are the transformations that occur to translate one data object into another.
- **Timing modeling**. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.
- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present .
- An equivalence class represents a set of valid or invalid states for input conditions.
- Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.
- Equivalence classes may be defined according to the following guidelines:
    1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
    2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
    3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
    4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Boundary Value Analysis

- A greater number of errors occurs at the boundaries of the input domain rather than in the “center.”
- It is for this reason that boundary value analysis (BVA) has been developed as a testing technique.
- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- Boundary value analysis is a test-case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the “edges” of the class.
- Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Boundary Value Analysis

Guidelines for BVA

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Orthogonal Array Testing

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.

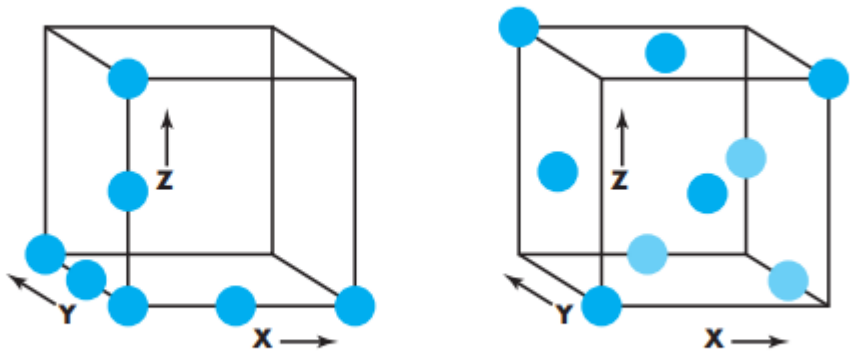
Orthogonal Array Testing

- consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are 3^3=27 possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 23.9 . Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).
- When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a “balancing property” .
- That is, test cases (represented by dark dots in the figure) are “dispersed uniformly throughout the test domain,” as illustrated in the right-hand cube in Figure 23.9 . Test coverage across the input domain is more complete.

FIGURE 23.9

A geometric view of test cases

Source: [Pha97].



One input item at a time

L9 orthogonal array

Orthogonal Array Testing

- To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:
- P1 = 1, send it now
- P1 = 2, send it one hour later
- P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

If a “one input item at a time” testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specifi ed: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).

But these would uncover only single mode faults , that is, faults that are triggered by a single parameter. Given the relatively small number of input parameters and discrete values, exhaustive testing is possible.

The number of tests required is 3 ^4 = 81, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

Orthogonal Array Testing

- The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure 23.10 .

FIGURE 23.10

An L9 orthogonal array

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Orthogonal Array Testing

- Phadke assesses the result of tests using the L9 orthogonal array in the following manner:
- **Detect and isolate all single mode faults**. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 23.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with “send it now” (P1 = 1)] as the source of the error. Such an isolation of fault is important to fi x the fault.
- **Detect all double mode faults**. If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
- **Multimode faults**. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

MODEL -BASED TESTING

MODEL -BASED TESTING

- Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test. The MBT technique requires five steps:
  1. Analyze an existing behavioral model for the software or create one.
  2. Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state. The inputs will trigger events that will cause the transition to occur.
  3. Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
  4. Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.
  5. Compare actual and expected results and take corrective action as required.

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

TESTING DOCUMENTATION AND HELP FACILITIES

TESTING DOCUMENTATION AND HELP FACILITIES

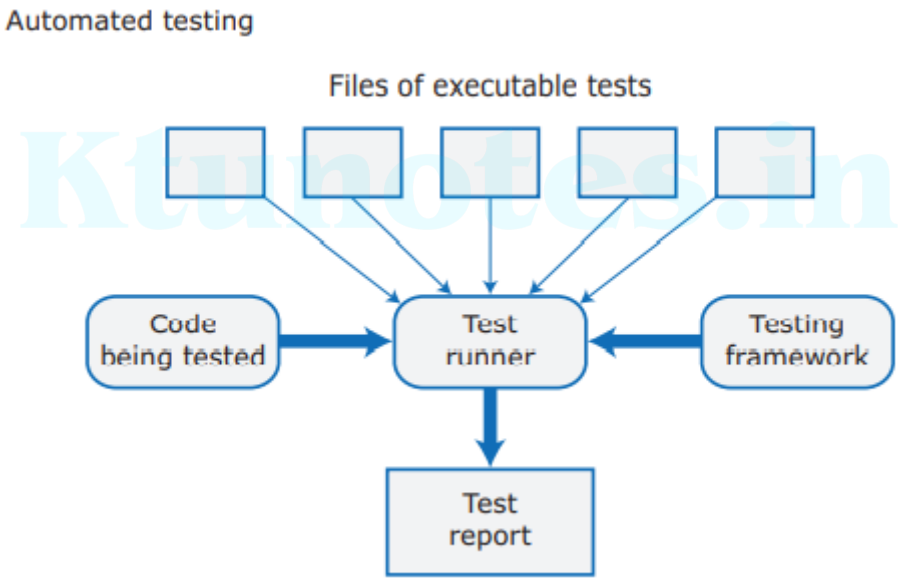
- documentation testing should be a meaningful part of every software test plan.
- Documentation testing can be approached in **two phases**. The first phase, **technical review** ,examines the document for editorial clarity. The second phase, **live test**, uses the documentation in conjunction with the actual program.
- A live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods.
- Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

Test automation

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.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product.
- The development of automated testing frameworks, such as JUnit for Java in the 1990s, reduced the effort involved in developing executable tests.
- Testing frameworks are now available for all widely used programming languages. A suite of hundreds of unit tests, developed using a framework, can be run on a desktop computer in a few seconds.
- A test report shows the tests that have passed and failed.
- Testing frameworks provide a base class, called something like “TestCase” that is used by the testing framework.
- To create an automated test, you define your own test class as a subclass of this TestCase class.
- Testing frameworks include a way of running all of the tests defined in the classes that are based on TestCase and reporting the results of the tests.

Test automation



Test automation

- It is good practice to structure automated tests in three parts:

1. Arrange :You 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.
2. Action: You call the unit that is being tested with the test parameters.
3. Assert: You make an assertion about what should hold if the unit being tested has executed successfully.

Once you set these up for one test, it is usually straightforward to reuse the setup code in other tests of the same unit. Ideally, you should have only one assertion in each test. If you have multiple assertions, you may not be able to tell which of them has failed.

Test automation

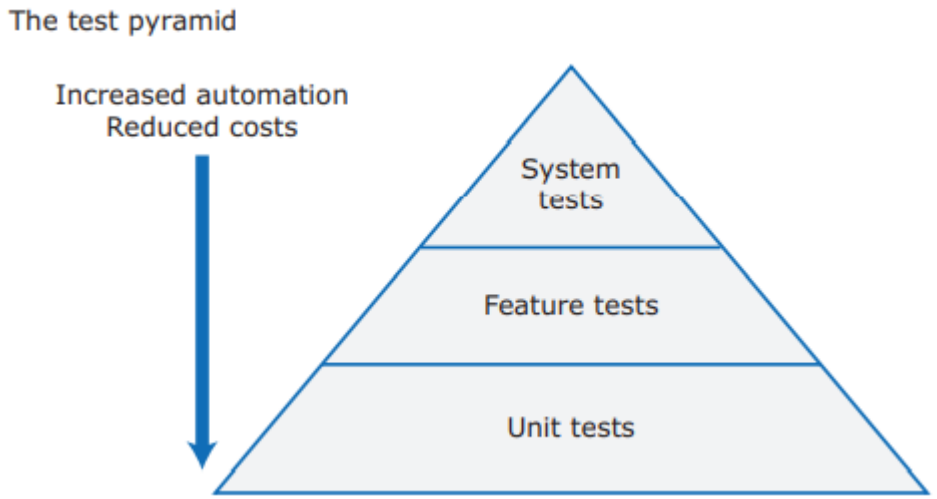
Two approaches to reduce the chances of test errors:

1. Make tests as simple as possible. The more complex the test, the more likely that it will be buggy. The test condition should be immediately obvious when reading the code.

2. Review all tests along with the code that they test. As part of the review process, someone apart from the test programmer should check the tests for correctness.

Test automation

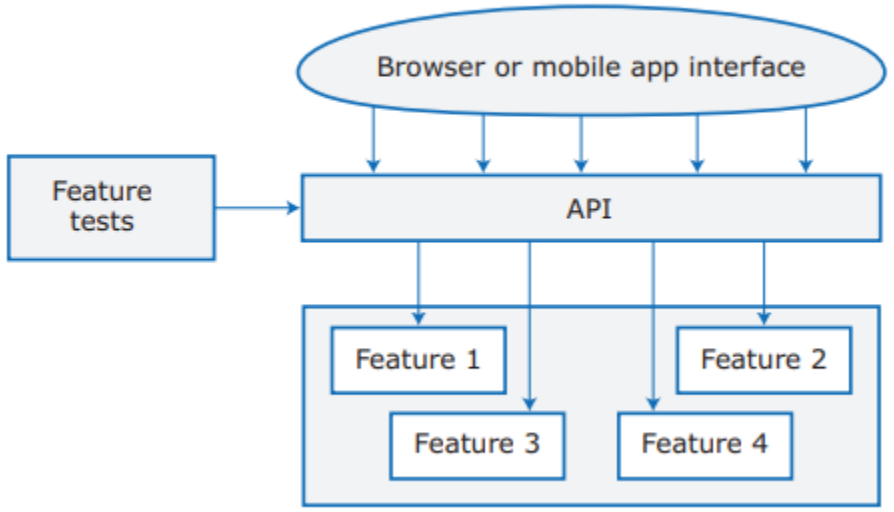
- Unit tests are the easiest to automate, so the majority of your tests should be unit tests. Mike Cohn, who first proposed the **test pyramid** suggests that 70% of automated tests should be unit tests, 20% feature tests (he called these service tests), and 10% system tests (UI tests)



Test automation

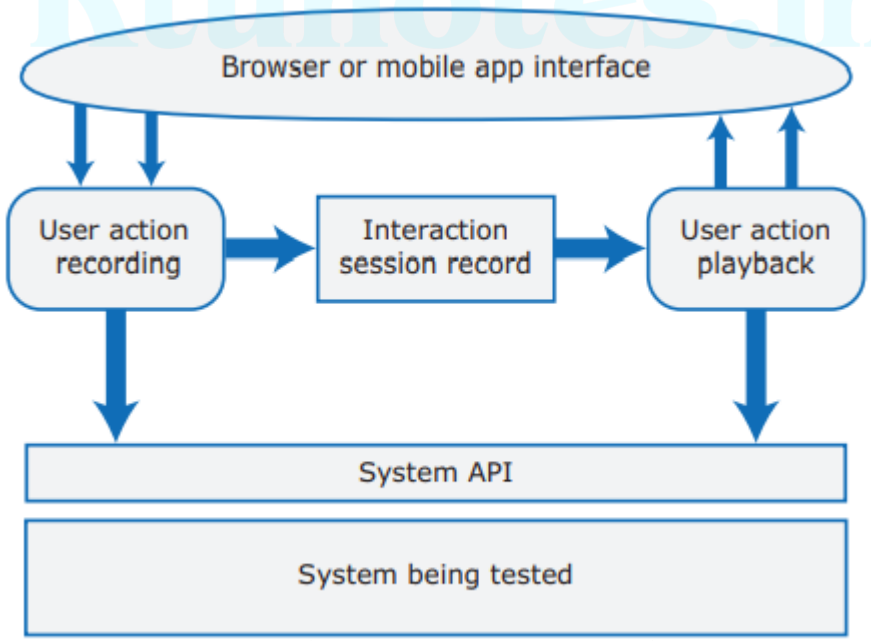
- Generally, users access features through the product's graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to use an alternative feature testing strategy.
- This involves designing your product so that its **features can be directly accessed through an API**, 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.

Feature testing through an API



Test automation

- **Interaction recording tools** record mouse movements and clicks, menu selections, keyboard inputs, and so on.
- They save the interaction session and can replay it, sending commands to the application and replicating them in the user's browser interface.
- These tools also provide scripting support so that you can write and execute scenarios expressed as test scripts.
- This is particularly useful for cross-browser testing, where you need to check that your software works in the same way with different browsers.

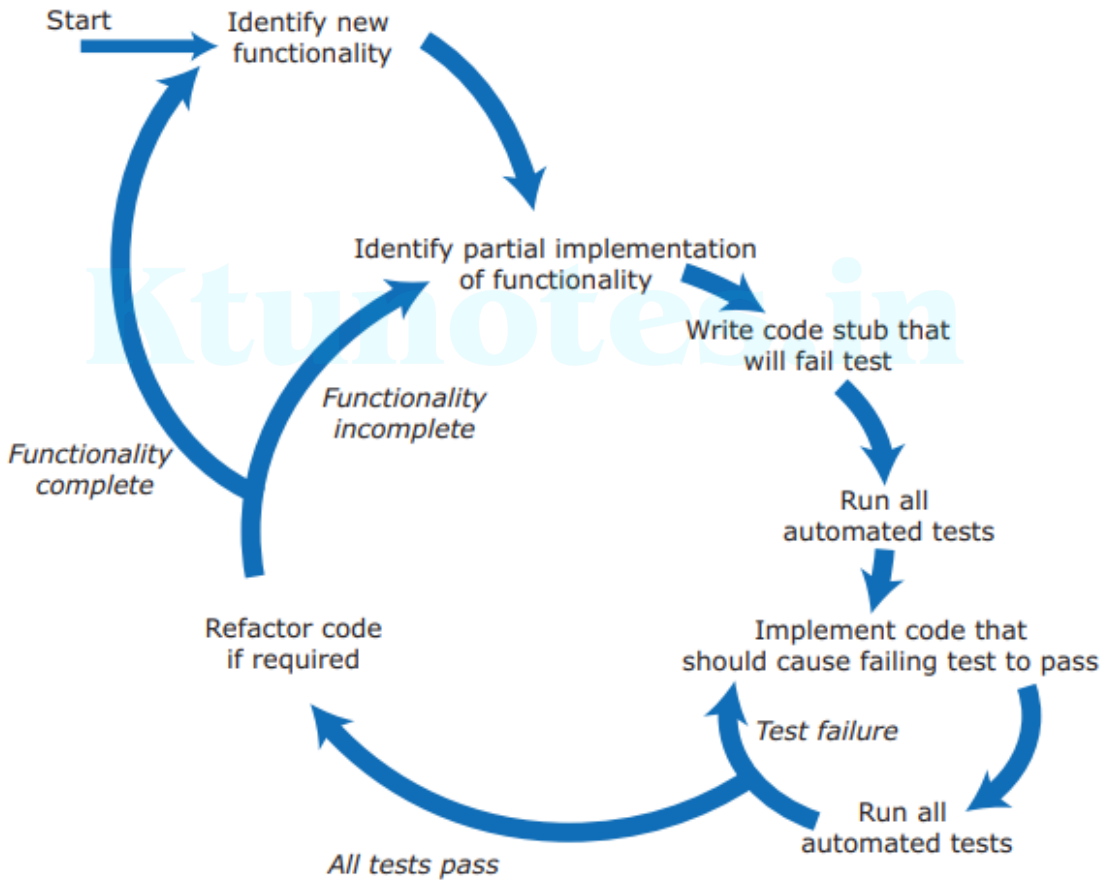


Test-driven development

Test-driven development

- Test-driven development (TDD) is an approach to program development that is based on the general idea that you should write an executable test or tests for code that you are writing before you write the code.
- TDD was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- 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



Test-driven development - stages

Table 9.9 Stages of 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.

Test-driven development

The benefits of test-driven development are:

1. It is a systematic approach to testing in which tests are clearly linked to sections of the program code. This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code.
2. 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
3. Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
4. It is argued that TDD leads to simpler code, as programmers only write code that's necessary to pass tests. They don't overengineer their code with complex features that aren't needed

Test-driven development

Disadvantages

- Test-driven development works best for the development of individual program units; it is more difficult to apply to system testing.
- it is challenging to use this approach when you are developing and testing systems with graphical user interfaces

Security testing

Security testing

- It aims to find vulnerabilities that an attacker may exploit 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.
- Discovering vulnerabilities is much harder than finding bugs.
- Three challenges in vulnerability testing:
    1. When you are testing for vulnerabilities, you are testing for something that the software should not do, so there are an infinite number of possible tests.
    2. Vulnerabilities are often obscure and lurk in rarely used code, so they may not be revealed by normal functional tests.
    3. Software products depend on a software stack that includes operating systems, libraries, databases, browsers, and so on. These may contain vulnerabilities that affect your software. These vulnerabilities may change as new versions of software in the stack are released.

Security testing

- One practical way to organize security testing is to adopt a **risk-based approach**, where you identify the common risks and then develop tests to demonstrate that the system protects itself from these risks.
- You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.
- In a risk-based approach, you start by identifying the main security risks to your product. To identify these risks, you use knowledge of possible attacks, known vulnerabilities, and security problems.
- Based on the risks that have been identified, you 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 behavior and its files.
- Once you have identified security risks, you then analyze them to assess how they might arise.

Security testing

Table 9.11 Examples of security risks

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

- For the first risk in Table 9.11 (unauthorized attacker) there are several possibilities:

1. The user has set weak passwords that an attacker can guess.
2. The system's password file has been stolen and an attacker has discovered the passwords.
3. The user has not set up two-factor authentication.
4. An attacker has discovered the credentials of a legitimate user through social engineering techniques.

Develop tests to check some of these possibilities. For example, you might run a test to check that the code that allows users to set their passwords always checks the strength of the passwords. It should not allow users to set passwords that are easy to crack.

Security testing

- The reliable programming techniques provide some protection against these risks.. Developers might have made mistakes.
- As well as adopting a risk-based approach to security testing, you can use **basic tests** that check whether or not common programming mistakes have occurred.
- These might test that sessions are properly closed or that inputs have been validated.
- An example of a basic test that checks for incorrect session management is a simple login test:
    1. Log into a web application.
    2. Navigate to a different website.
    3. Click the BACK button of the browser

When you are testing the features of a system, it is sensible to focus on those features that are most used and to test the “normal” usage of these features.

However, when you are testing the security of a system, you need to think like an attacker rather than a normal end-user.

DevOps

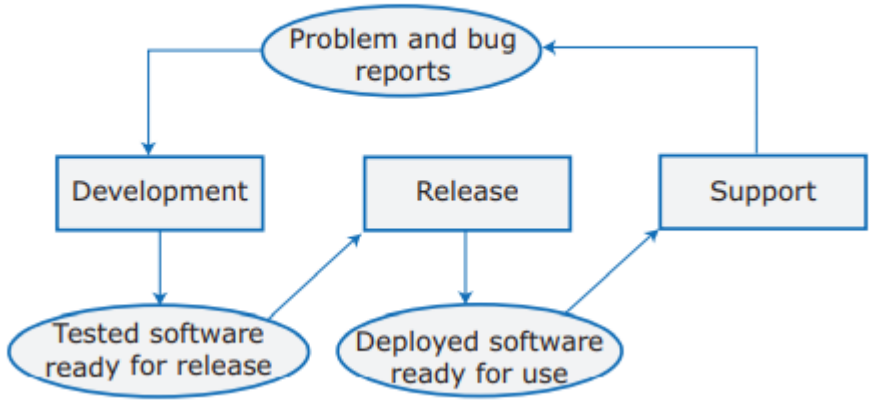
Code management

DevOps automation

DevOps

- The ultimate goal of software product development is to release a product to customers.
- Traditionally, separate teams were responsible for software development, software release, and software support

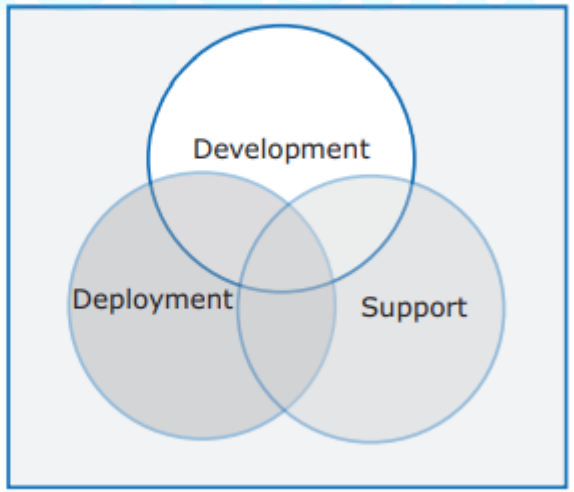
Figure 10.1 Development, release, and support



DevOps

- more and more companies are using an alternative approach called DevOps. DevOps (development + operations) integrates development, deployment, and support, with a single team responsible for all of these activities

Figure 10.2 DevOps



DevOps

- Three factors led to the development and widespread adoption of DevOps:

1. Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment. Agile enthusiasts started looking for a way around this problem.
2. Amazon re-engineered their software around services and introduced an approach in which a service was both developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.
3. 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- priniples

Table 10.1 DevOps principles

Principle	Explanation
Everyone is responsible for everything.	All team members have joint responsibility for developing, delivering, and supporting the software.
Everything that can be automated should be automated.	All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.
Measure first, change later.	DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

DevOps - Benefits

Table 10.2 Benefits of DevOps

Benefit	Explanation
Faster deployment	Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
Reduced risk	The increment of functionality in each release is small so there is less chance of feature interactions and other changes that cause system failures and outages.
Faster repair	DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team was responsible for the problem and to wait for them to fix it.
More productive teams	DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

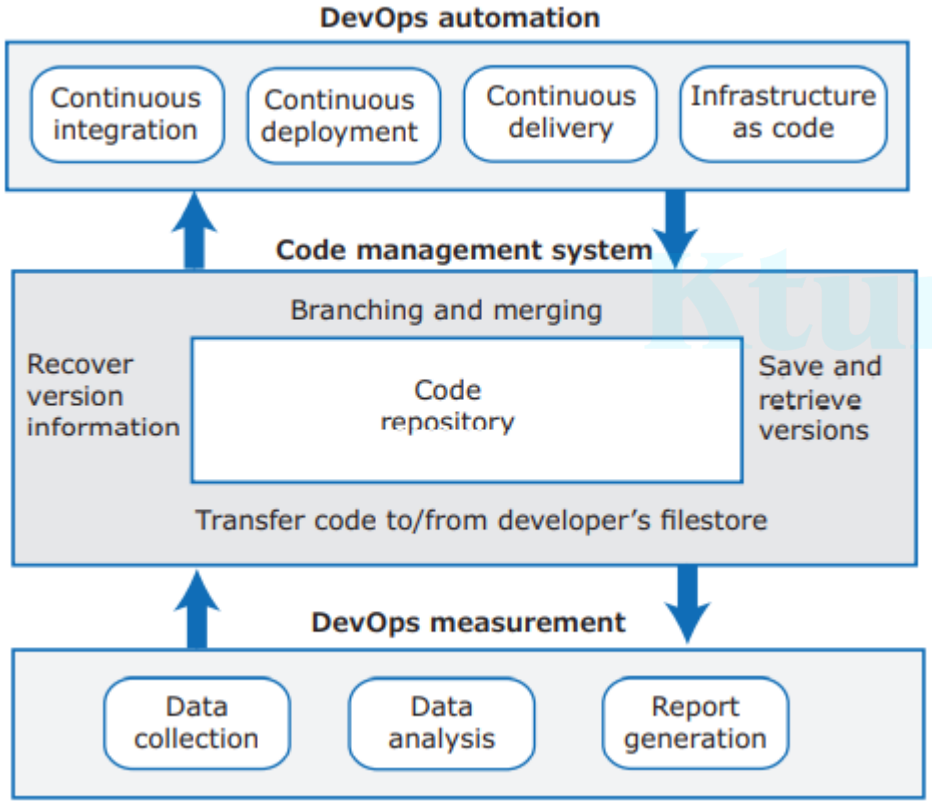
code management

Fundamentals of source code management

Code management

- Code management is a set of software-supported practices used to manage an evolving codebase.
- You need code management 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.
- Source code management, combined with automated system building, is critical 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, but it also stores all other information that is used in DevOps processes. DevOps automation and measurement tools all interact with the code management system

Code management



Fundamentals of source code management

- Source code management systems are designed to manage an evolving project codebase to allow different versions of components and entire systems to be stored and retrieved.
- Developers can work in parallel without interfering with each other and they can integrate their work with that from other developers.
- The code management system provides a set of features that support four general areas:

1. Code transfer Developers take code into their personal file store to work on it; then they return it to the shared code management system.
2. Version storage and retrieval Files may be stored in several different versions, and specific versions of these files can be retrieved.
3. Merging and branching Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.
4. Version information Information about the different versions maintained in the system may be stored and retrieved

Fundamentals of source code management

1. All source code files and file versions are stored in the repository, as are other artifacts such as configuration files, build scripts, shared libraries, and versions of tools used. The repository includes a database of information about the stored files, such as version information, information about who has changed the files, what changes were made at what times, and so on.

2. The source code management features transfer files to and from the repository and update the information about the different versions of files and their relationships. Specific versions of files and information about these versions can always be retrieved from the repository.

- Several open-source and proprietary-source code management systems are currently used.
- When files are added, the source code management system assigns a unique identifier to each file.
- This is used to name stored files; the unique name means that managed files can never be overwritten.
- Other identifying attributes may be added to the controlled file so that it can be retrieved by name or by using these attributes.
- Any version of a file can be retrieved from the system.
- When a change is made to a file and it is submitted to the system, the submitter must add an identifying string that explains the changes made.
- This helps developers understand why the new version of the file was created.

Fundamentals of source code management

Table 10.4 Features of source code management systems

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.

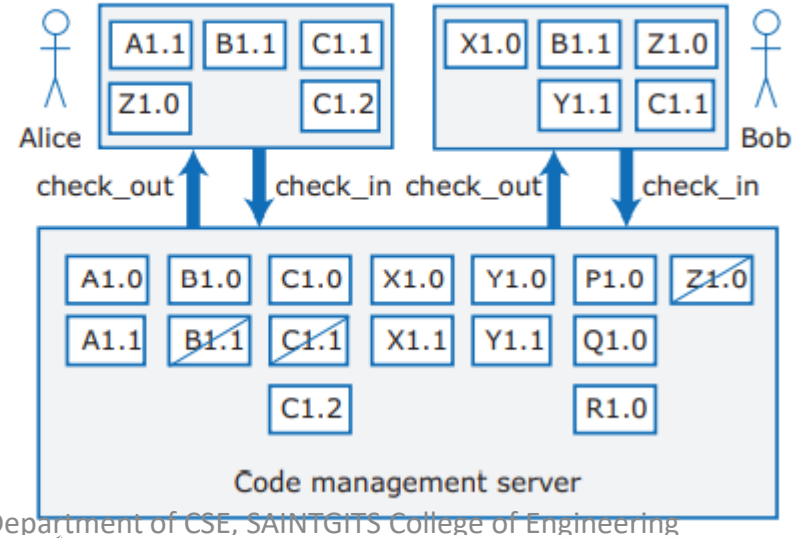
Fundamentals of source code management

- Code management systems support independent development when several developers work on the same file simultaneously.
- They submit changes to the code management system, which creates a new version of the file for each submission. This avoids the file overwriting problem.
- Rather than storing every version of a file, storage compaction reduced the space required for the set of files being managed.
- These systems stored a version as a list of changes from a master version of a file.
- The version file could be recreated by applying these changes to the master file.
- If several versions of a file had been created, it was relatively slow to recreate a specific version, as this involved retrieving and applying several sets of code edits.
- As storage is now cheap and plentiful, modern code management systems are less concerned with optimizing storage. They use faster mechanisms for version storage and retrieval

Fundamentals of source code management

- Early source code management systems had a centralized repository architecture that requires users to check in and check out files (Figure 10.4).
- If a user checks out a file, anyone else who tries to check out that file is warned that it is already in use. When the edited file is checked in, a new version of that file is created.

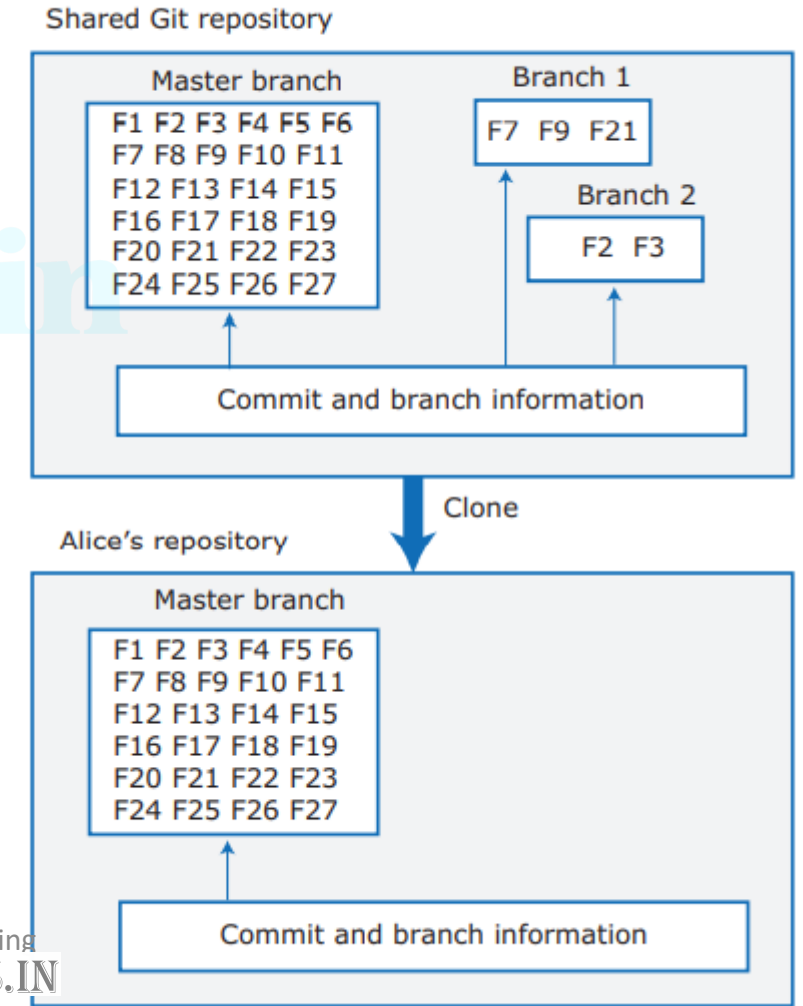
Figure 10.4 Centralized source code management



Fundamentals of source code management

- In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a distributed version control system (DVCS) called Git to manage the code of the Linux kernel.
- Git was geared to supporting large-scale open-source development.
- It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.
- 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 (Figure 10.5).
- A fundamental concept in Git is the "master branch," which is the current master version of the software that the team is working on. You create new versions by creating a new branch.
- Two branches have been created in addition to the master branch. When users request a repository clone, they get a copy of the master branch that they can work on independently

Figure 10.5 Repository cloning in Git



Fundamentals of source code management

- Git and other distributed code management systems have several advantages over centralized systems:

1. 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.
2. Speed - Committing changes to the repository is a fast, local operation and does not need data to be transferred over the network.
3. Flexibility - Local experimentation is much simpler. Developers can safely try different approaches without exposing their experiments to other project members. With a centralized system, this may only be possible by working outside the code management system

Fundamentals of source code management

- 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 (Figure 10.6).
- 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.

DevOps automation

Continuous integration

Continuous delivery and deployment

Infrastructure as code

DevOps automation

- By using DevOps with automated support you 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.
- In addition to reducing the costs and time required for integration, deployment, and delivery, automation 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.
- Deployment does not depend on a system manager who knows the server configurations. A specific server configuration can be quickly and reliably reproduced using the system model.

DevOps automation

Table 10.5 Aspects of 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.

DevOps automation

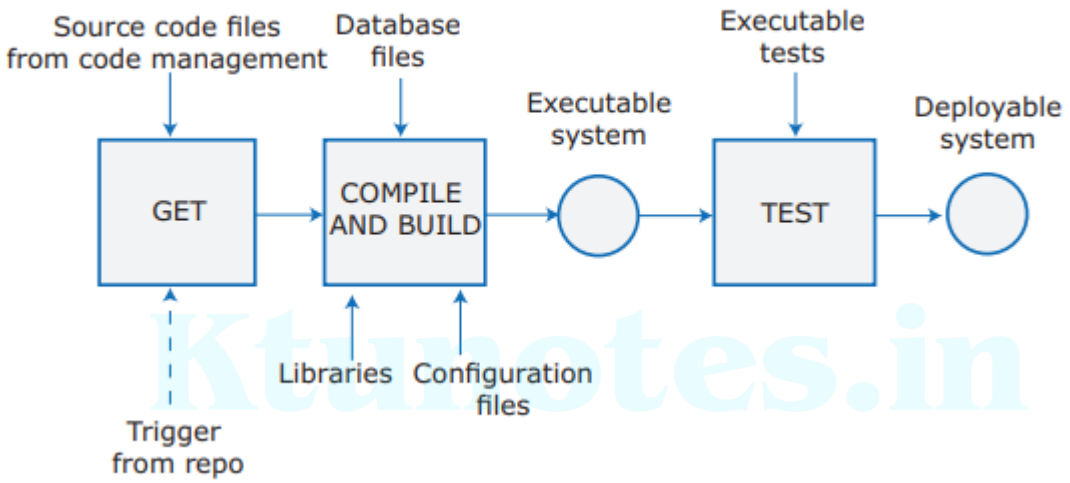
automation of issue tracking.

- Issue and bug tracking involves recording observed problems and the development team's responses to these problems. If you use Scrum or a Scrum-like process, these issues should be automatically added to the product backlog.
- Several open-source and proprietary issue-tracking tools are widely used, such as Bugzilla, FogBugz, and JIRA.
- These systems include the following features:

1. Issue reporting: Users and testers can report an issue or a bug and provide further information about the context where the problem was discovered. These reports can be sent automatically to developers. Developers can comment on the report and indicate if and when the issue has been resolved. Reports are stored in an issue database.
2. Searching and querying :The issue database may be searched and queried. This is important to discover whether issues have already been raised, to discover unresolved issues, and to find out if related issues have been reported.
3. Data analysis: The issue database can be analyzed and information extracted, such as the number of unresolved issues, the rate of issue resolution, and so on. This is often presented graphically in a system dashboard.
4. Integration with source code management: Issue reports can be linked to versions of software components stored in the code management system

Continuous integration

Figure 10.9 Continuous integration

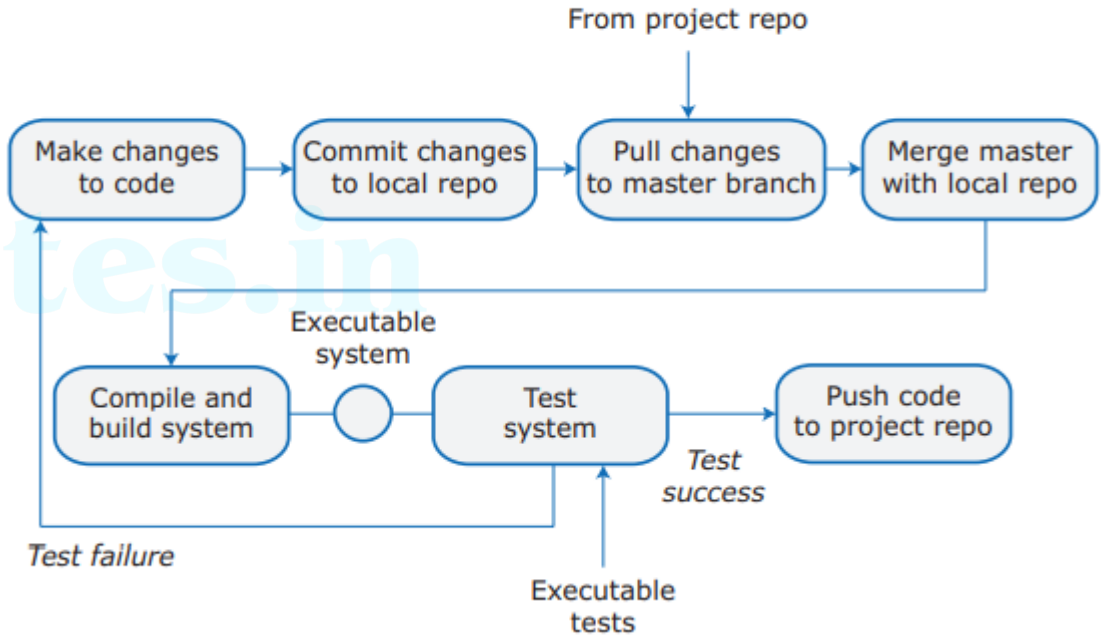


- Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared code repository.
- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product

Continuous integration

- In a continuous integration environment, developers have to make sure that they don't "break the build."
- Breaking the build means pushing code to the project repository, which when integrated, causes some of the system tests to fail. This holds up other developers.
- If this happens to you, your priority is to discover and fix the problem so that normal development can continue.
- To avoid breaking the build, you should always adopt an "integrate twice" approach to system integration.
- You should integrate and test on your own computer before pushing code to the project repository to trigger the integration server (Figure 10.10).

Figure 10.10 Local integration



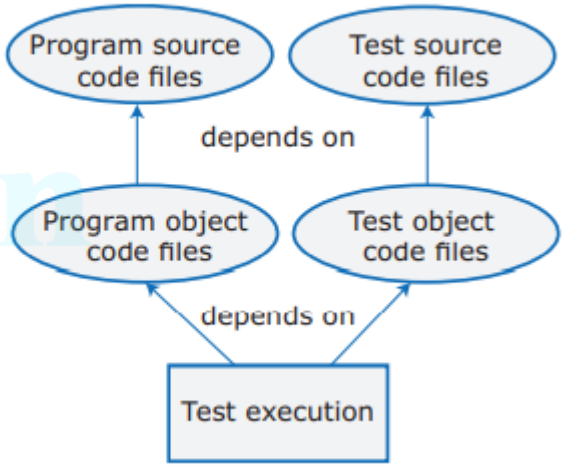
Continuous integration

- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.
- If you continuously integrate, then a working system is always available to the whole team.
- This can be used to test ideas and to demonstrate the features of the system to management and customers. Furthermore, continuous integration creates a “quality culture” in a development team.
- Team members want to avoid the stigma and disruption of breaking the build. They are likely to check their work carefully before pushing it to the project repo.

Continuous integration

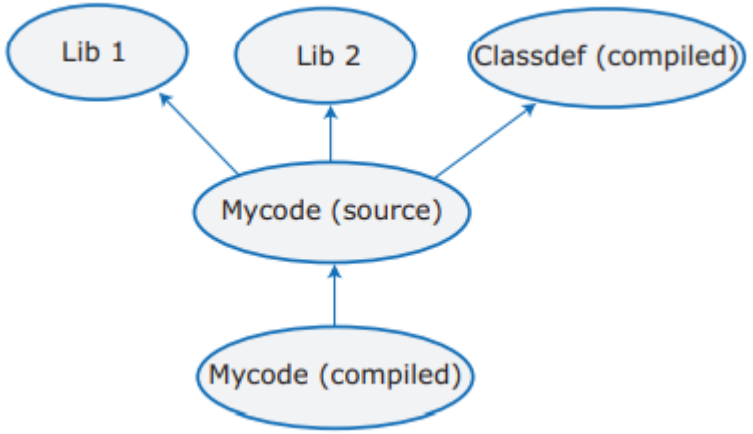
- Continuous integration is effective only if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- Fast automated building is possible because, in a continuous integration system, the changes made to the system between one integration and another are usually relatively small.
- Code integration tools use an incremental build process so that they only have to repeat actions, such as compilation, if the dependent files have been changed.
- To understand incremental system building, you need to understand the concept of dependencies.
- Figure shows a dependency model that shows the dependencies for test execution. An upward-pointing arrow means “depends on” and shows the information required to complete the task shown in the rectangle at the base of the model.
- Running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests. In turn, these depend on the source code for the system and the tests that are compiled to create the object code.

A dependency model



Continuous integration

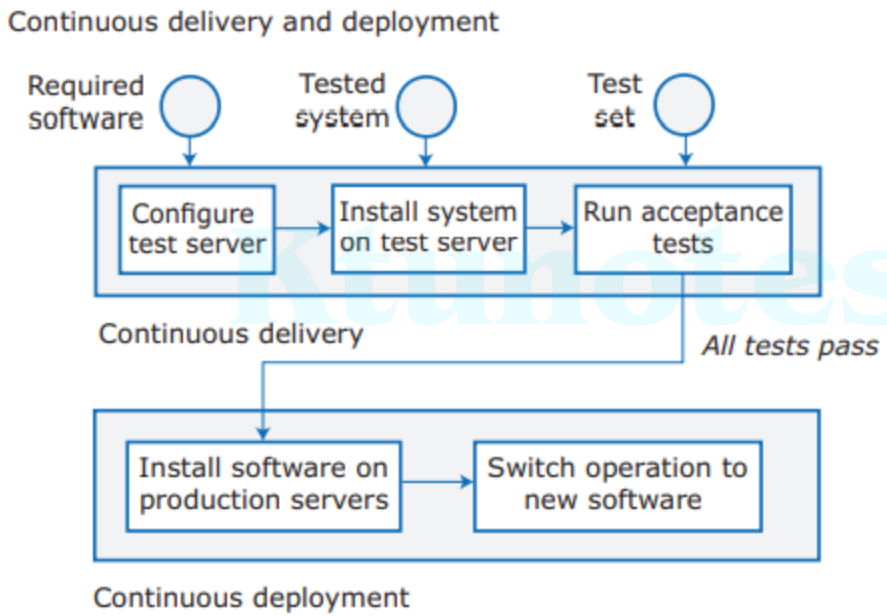
- Figure below shows is a lower-level dependency model that shows the dependencies involved in creating the object code for a source code file called Mycode.
- Source code files are rarely independent but rely on other information such as libraries. Mycode depends on two libraries (Lib 1 and Lib 2) and an externally defined class definition



Continuous delivery and deployment

- 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.
- As well as feature tests, you should run load tests that show how the software behaves as the number of users increases.
- You may also run tests to check the throughput of transactions and your system's response time
- . The simplest way to create a replica of a production environment is to run your software in a container.
- Your production environment is defined as a container, so to create a test environment, you simply create another container using the same image.
- This ensures that changes made to the production environment are always reflected in the test environment.

Continuous delivery and deployment



Continuous delivery and deployment

- After initial integration testing, a staged test environment is created.
- This is a replica of the actual production environment in which the system will run.
- The system acceptance tests, which include functionality, load, and performance tests, are then run to check that the software works as expected.
- If all of these tests pass, the changed software is installed on the production servers.
- To deploy the system, you transfer the software and required data to the production servers.
- You then momentarily stop all new requests for service and leave the older version to process the outstanding transactions.
- Once these have been completed, you switch to the new version of the system and restart processing.
- Continuous deployment is obviously only practical for cloud-based systems.
- If your product is sold through an app store or downloaded from your website, continuous integration and delivery make sense. A working version is always available for release.

Continuous delivery and deployment

Table 10.6 Benefits of continuous deployment

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.

Continuous delivery and deployment

- There are three business reasons you may not want to deploy every software change to customers:

1. You may have incomplete features available that could be deployed, but you want to avoid giving competitors information about these features until their implementation is complete.
2. Customers may be irritated by software that is continually changing, especially if this affects the user interface. They don't want to spend time continually learning about new features. Rather, they prefer to have a number of new features available before learning about them.
3. You may wish to synchronize releases of your software with known business cycles.

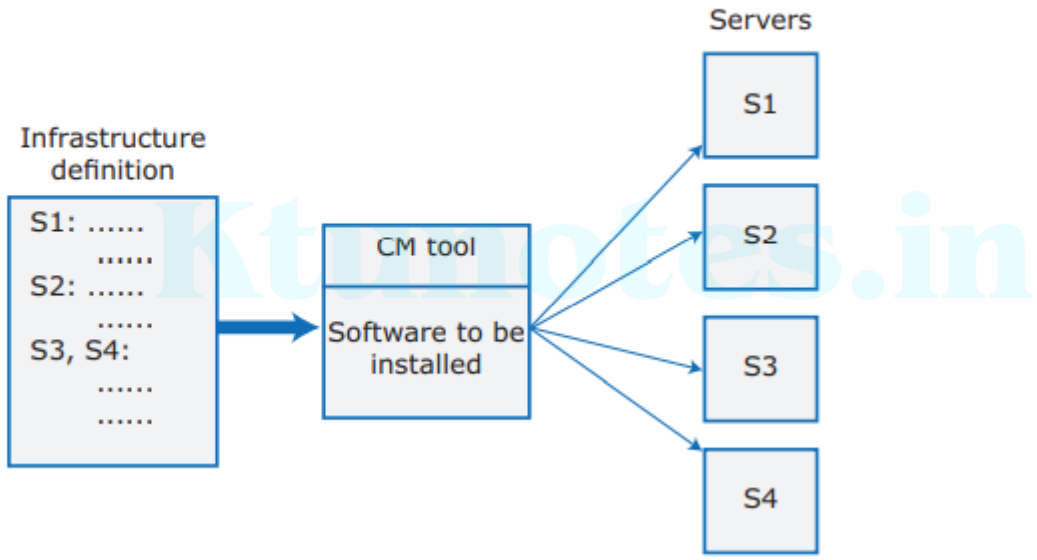
CI tools such as Jenkins and Travis may also be used to support continuous delivery and deployment

Infrastructure as code

- Manually maintaining a computing infrastructure with tens or hundreds of servers is expensive and error-prone.
- 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.
- The CM tool accesses a master copy of the software to be installed and pushes this to the servers being provisioned.
- When changes have to be made, the infrastructure model is updated and the CM tool makes the change to all servers.
- Defining your software infrastructure as code is obviously relevant to products that are delivered as services.
- The product provider has to manage the infrastructure of their services on the cloud. However, it is also relevant if software is delivered through downloads.

Infrastructure as code

Figure 10.14 Infrastructure as code



Infrastructure as code

Defining your infrastructure as code and using a configuration management system solve two key problems of continuous deployment:

1. Your testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.
2. When you change a service, you have to be able to roll that change out to all of your servers quickly and reliably. If there is a bug in your changed code that affects the system's reliability, you have to be able to seamlessly roll back to the older system.

Infrastructure as code

- The business benefits of defining your infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented.
- Characteristics of infrastructure as a code

Table 10.7 Characteristics of infrastructure as code

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.

DevOps measurement

- Measurements about software development and use fall into four categories:

1. Process measurements You collect and analyze data about your development, testing, and deployment processes.
2. Service measurements You collect and analyze data about the software's performance, reliability, and acceptability to customers.
3. Usage measurements You collect and analyze data about how customers use your product.
4. Business success measurements You collect and analyze data about how your product contributes to the overall success of the business.

Module 3(part 3)

Object-oriented design using the UML, Design patterns, Implementation issues, Open-source development - Open-source licensing - GPL, LGPL, BSD. Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. Informal Review, Formal Technical Reviews, Post-mortem evaluations. Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing, Debugging, White box testing, Path testing, Control Structure testing, Black box testing, Testing Documentation and Help facilities. Test automation, Test-driven development, Security testing. Overview of DevOps and Code Management - Code management, DevOps automation, Continuous Integration, Delivery, and Deployment (CI/CD/CD). **Software Evolution - Evolution processes, Software maintenance**

Software evolution

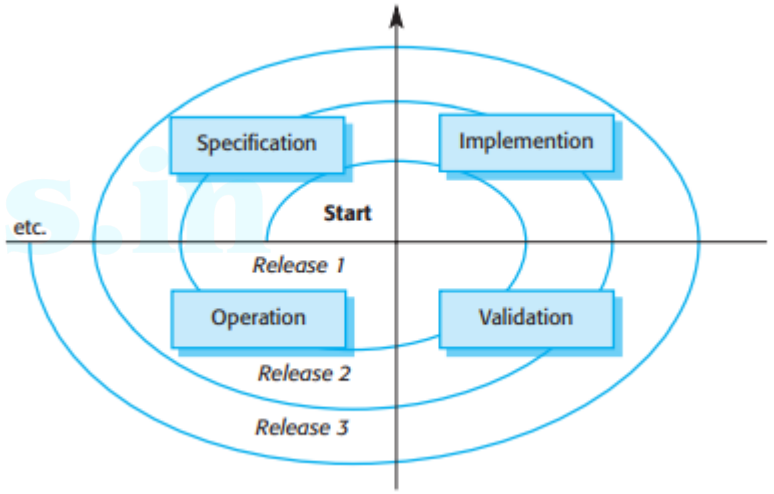
- Large software systems usually have a long lifetime.
- Successful software products and apps may have been introduced many years ago with new versions released every few years.
- Eg: the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.
- During their lifetime, operational software systems have to change if they are to remain useful.
- Business changes and changes to user expectations generate new requirements for the software.
- Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics.
- Software products and apps have to evolve to cope with platform changes and new features introduced by their competitors.
- Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Software evolution

- Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader “system of systems.”
- In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems.
- Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.
- **brownfield software development** -situations in which software systems have to be developed and managed in an environment where they are dependent on other software systems.

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 release 1 of the system.
- Once delivered, changes are proposed, and the development of release 2 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.

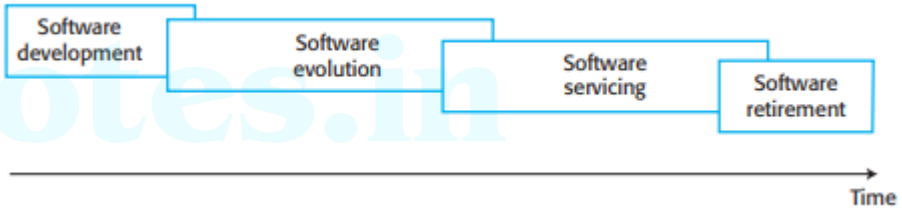


Spiral model of development and evolution

Software evolution

alternative view of the software evolution life cycle for business systems

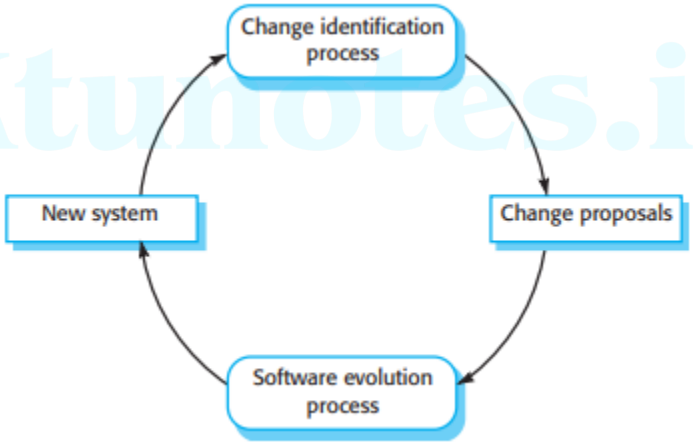
- In this model, they distinguish between evolution and servicing.
- Evolution is the phase in which significant changes to the software architecture and functionality are made.
- During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each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



Evolution Processes

- As with all software processes, there is no such thing as a standard software change or evolution process.
- The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved.
- For some types of system, such as **mobile apps**, evolution may be an informal process, where change requests mostly come from conversations between system users and developers.
- For other types of systems, such as **embedded critical systems**, software evolution may be formalized, with structured documentation produced at each stage in the process.
- Formal or informal system change proposals are the driver for system evolution in all organizations.
- In a change proposal, an individual or group suggests changes and updates to an existing software system.
- These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team.
- The processes of **change identification and system evolution** are cyclical and continue throughout the lifetime of a system(figure in the next slide)

Evolution Processes



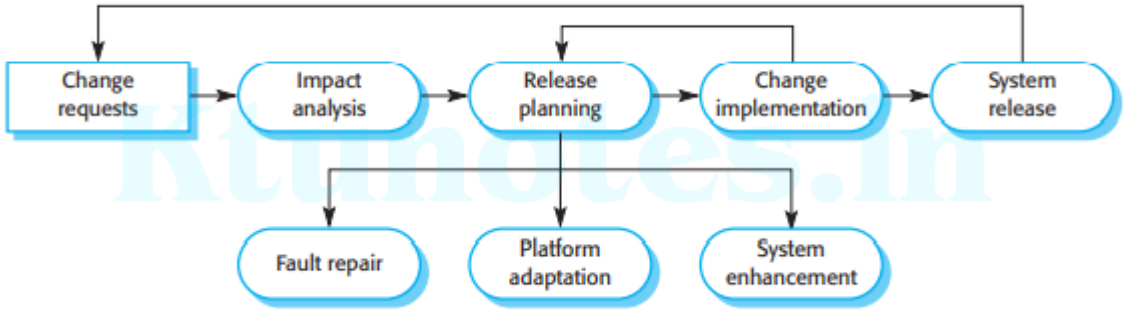
Change identification and evolution process

Evolution Processes

Activities involved in software evolution. (figure in the next slide)

- 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.
- In situations where development and evolution are integrated, change implementation is simply an iteration of the development process.
- Revisions to the system are designed, implemented, and tested.
- The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.
- Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.
- During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

Evolution Processes



Evolution Processes

- If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required .
- New software requirements should be written, and these should be analyzed and validated.
- If the design has been documented using UML models, these models should be updated.
- The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

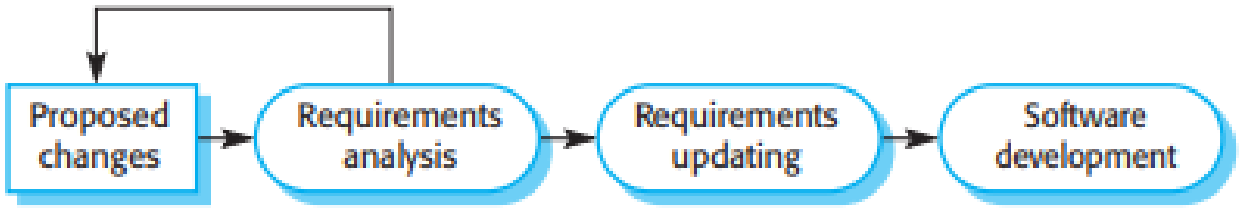


Figure:Change implementation

Evolution Processes

- change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent.

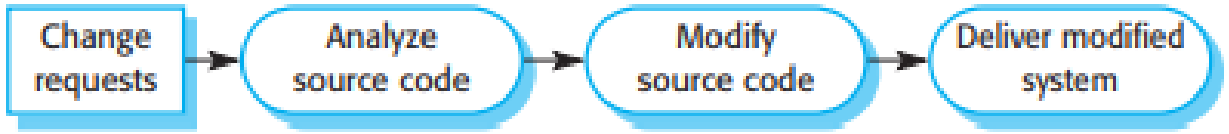


Figure 9.6 The emergency repair process

Evolution Processes

- Agile methods and processes, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.
- However, problems may arise during the handover from a development team to a separate team responsible for system evolution.
- There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes.
2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests.

Software maintenance

Maintenance prediction

Software reengineering

Refactoring

Software maintenance

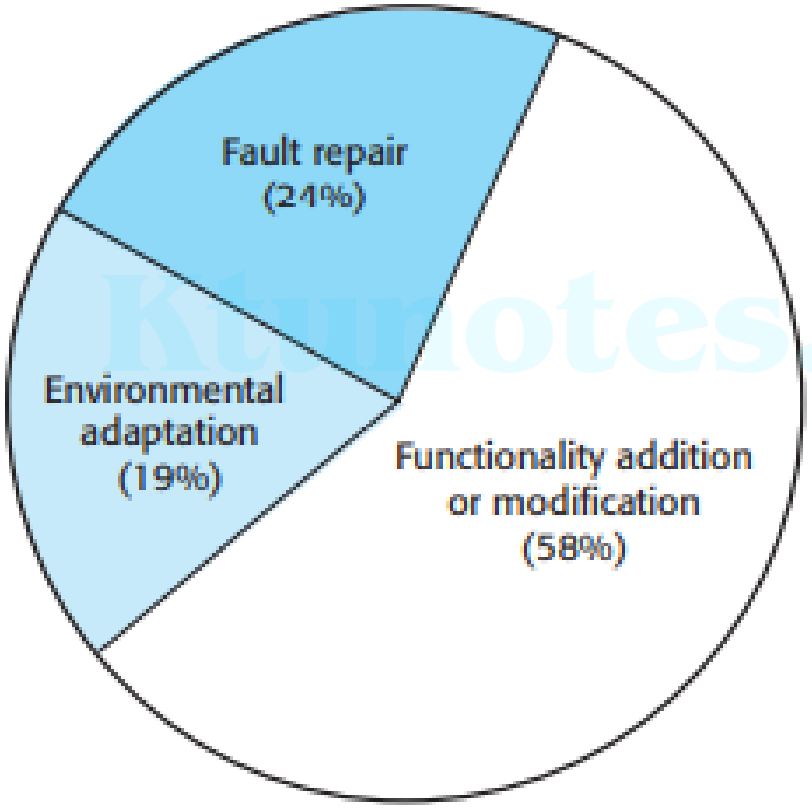
- Software maintenance is the general process of changing a system after it has been delivered.
- The term is usually applied to custom software, where separate development groups are involved before and after delivery.
- The changes made to the software may be simple changes to correct coding errors,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

Software maintenance

- There are three different types of software maintenance:

1. **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.
2. **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.
3. **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



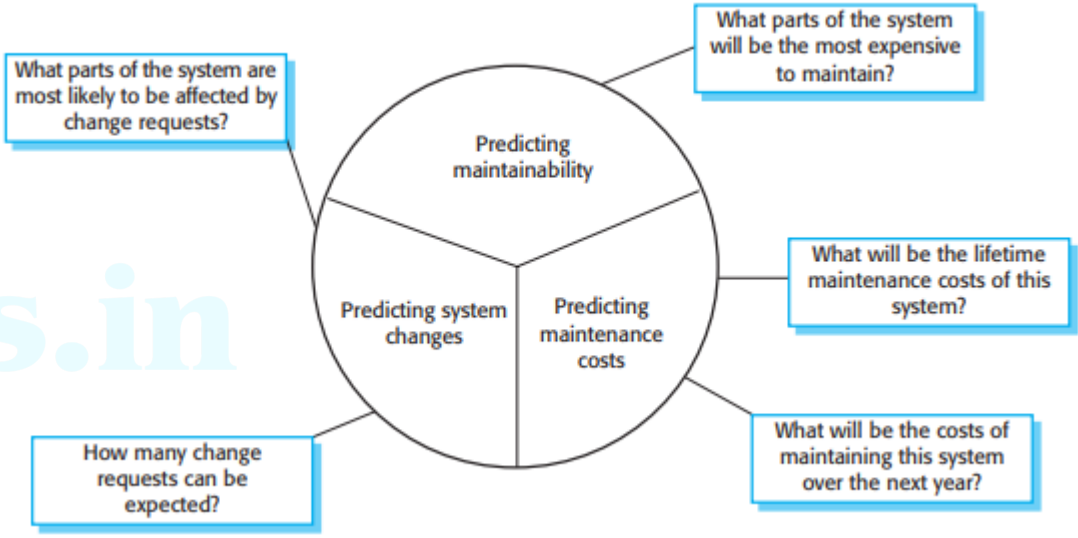
Software maintenance

- It is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. A new team has to understand the program being maintained.
2. Separating maintenance and development means there is no incentive for the development team to write maintainable software.
3. Program maintenance work is unpopular. Maintenance has a poor image among software engineers.
4. As programs age, their structure degrades and they become harder to change.

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.
- If you understand this, you can design the software components that are most likely to change to make them more adaptable.
- Figure shows possible predictions and the questions that these predictions may answer.
- 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.



Maintenance prediction

- To evaluate the relationships between a system and its environment, you should look at:

1. The number and complexity of system interfaces
2. The number of inherently volatile system requirements
3. The business processes in which the system is used

Maintenance prediction

- After a system has been put into service, you may be able to use process data to help predict maintainability.
- Examples of process metrics that can be used for assessing maintainability are:

1. Number of requests for corrective maintenance
2. Average time required for impact analysis
3. Average time taken to implement a change request
4. Number of outstanding change requests

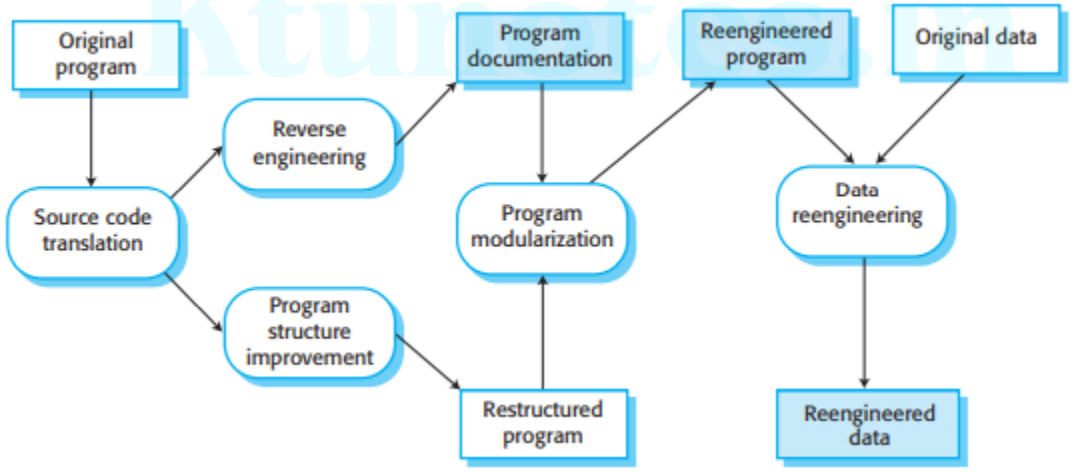
Software reengineering

- To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability.
- 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.
- Reengineering has two important advantages over replacement:

1. Reduced risk
2. Reduced cost

Software reengineering

- a general model of the reengineering process.
- The input to the process is a legacy program, and the output is an improved and restructured version of the same program.



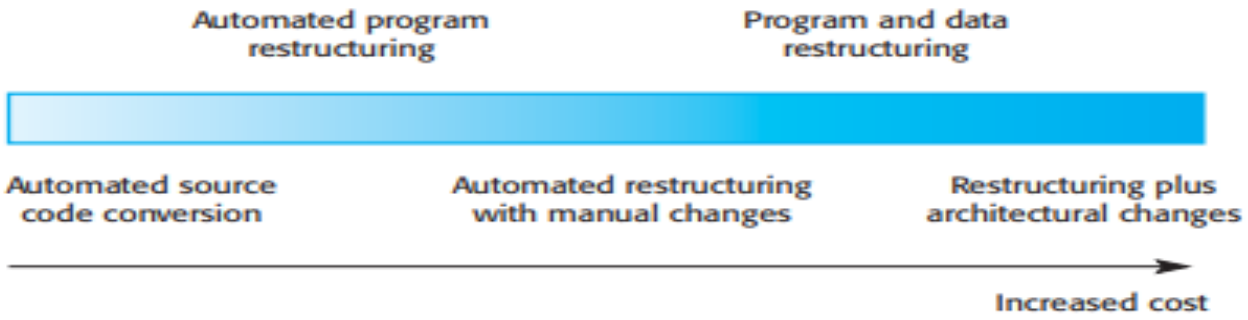
Software reengineering

- The activities in this reengineering process are:

1. **Source code translation**: Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.
2. **Reverse engineering**: The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. **Program structure improvement**: The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.
4. **Program modularization**: Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. **Data reengineering**: The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Software reengineering

- The costs of reengineering obviously depend on the extent of the work that is carried out.
- There is a spectrum of possible approaches to reengineering, as shown in Figure.
- Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.



Software reengineering

- The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering.
- It isn’t possible, for example, to convert a system written using a functional approach to an object-oriented system.
- Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive.
- Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

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 an inherent part of agile methods because these methods are based on change

Refactoring

- Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing.
- Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable.
- 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.
- Examples of bad smells that can be improved through refactoring include:

1. Duplicate code
2. Long methods - If a method is too long, it should be redesigned as a number of shorter methods
3. Switch (case) statements - These often involve duplication, where the switch depends on the type of a value.
4. Data clumping- Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. Speculative generality - This occurs when developers include generality in a program in case it is required in the future