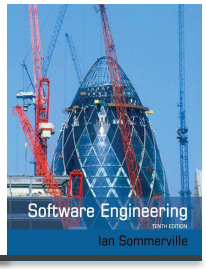
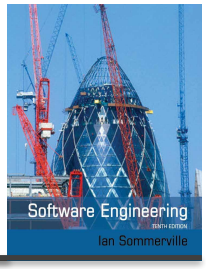


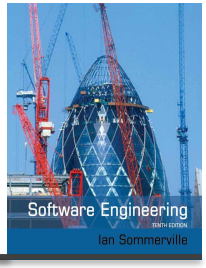
Chapter 7 – Design and Implementation

Topics covered



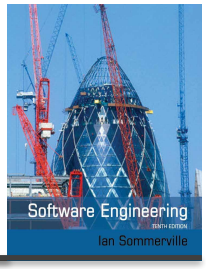
- Object-oriented design using the UML
- Design patterns
- Implementation issues
- Open source development

Design and implementation

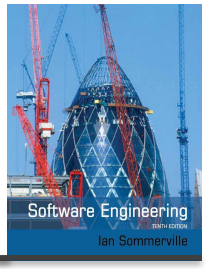


- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components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

Build or buy

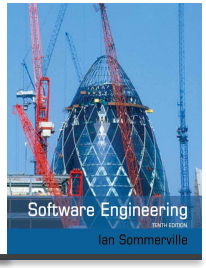


- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.



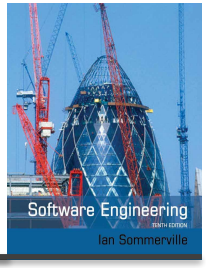
Object-oriented design using the UML

An object-oriented design process



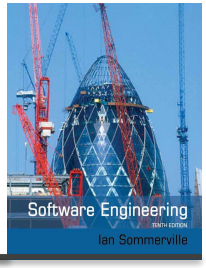
- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design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

Process stages



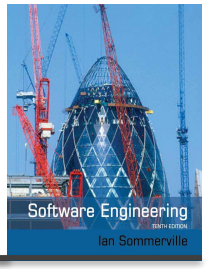
- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- Process illustrated here using a design for a wilderness weather station.

System context and interactions



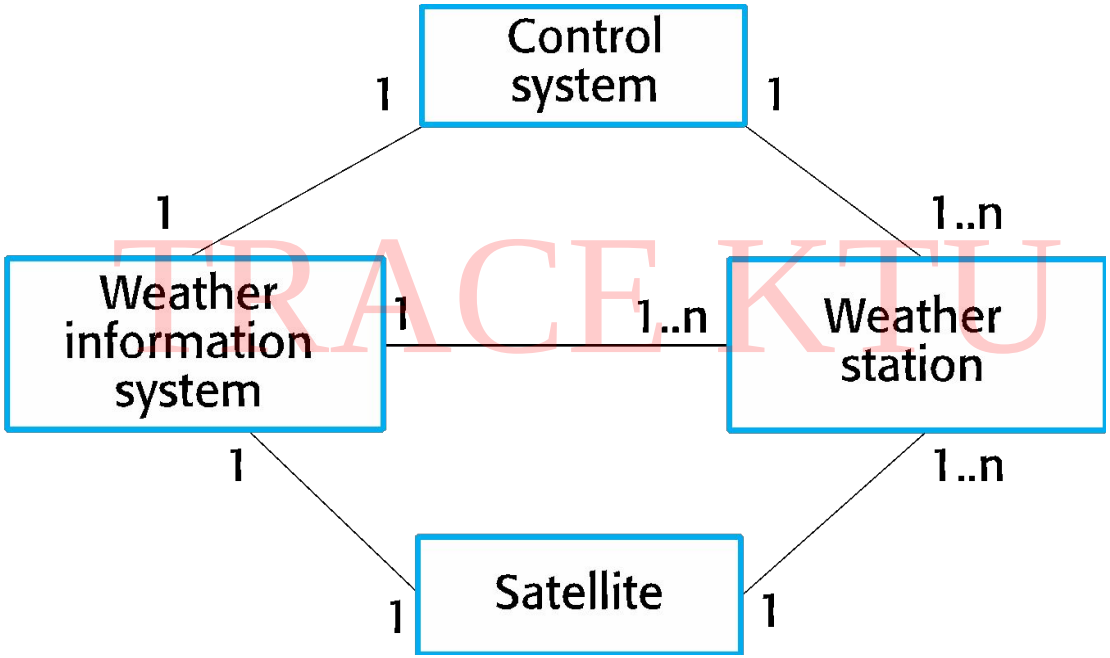
- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish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

Context and interaction models

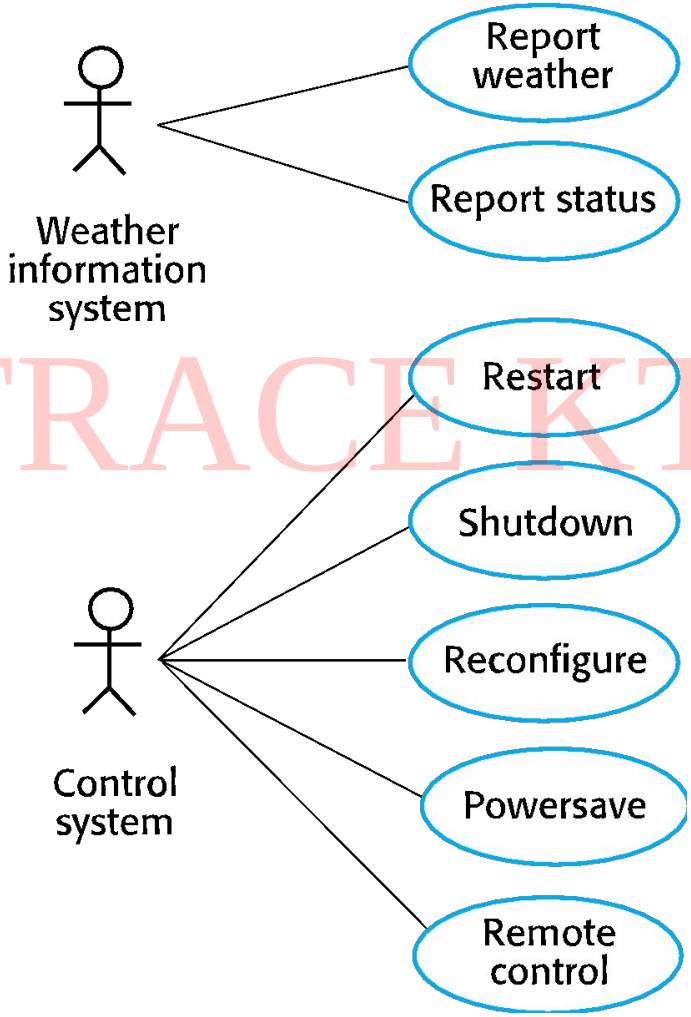


- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

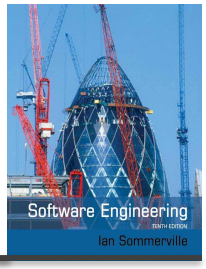
System context for the weather station



Weather station use cases

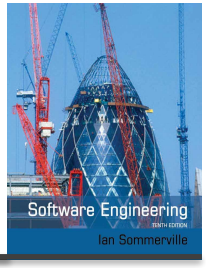


Use case description—Report weather



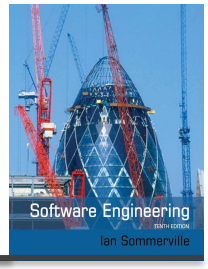
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum,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

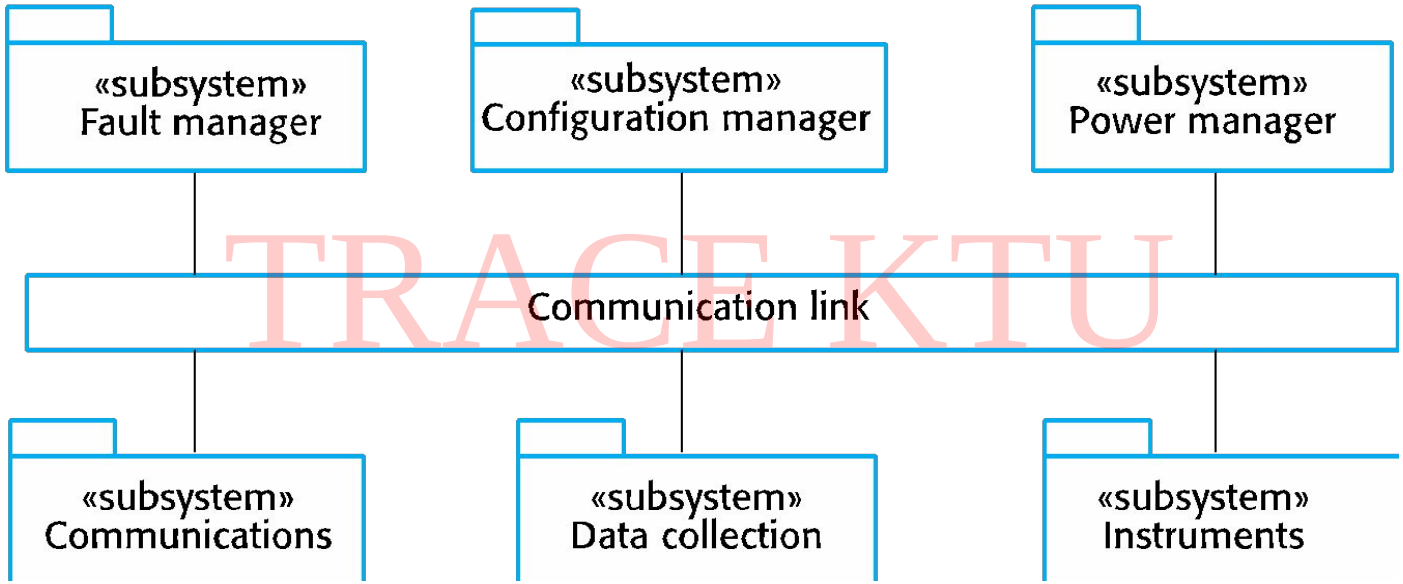
Architectural design



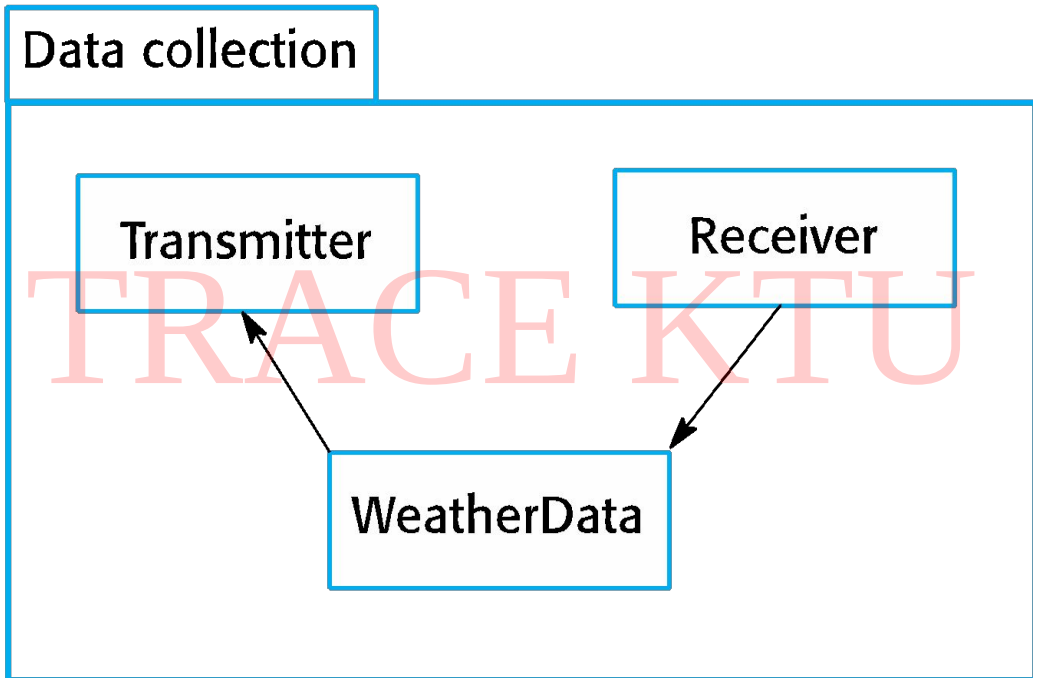
- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station

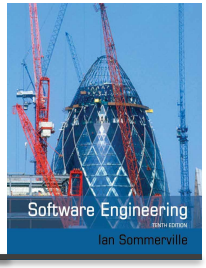




Architecture of data collection system

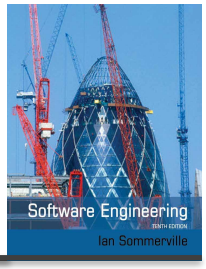


Object class identification



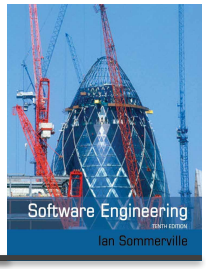
- Identifying object classes is often a difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first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

Approaches to identification



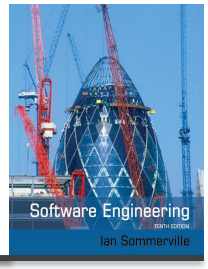
- Use a grammatical approach based on a natural language description of the system.
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes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

Weather station object classes



- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

Weather station object classes



reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

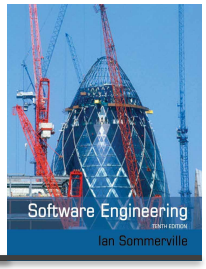
groundTemperatures windSpeeds windDirections pressures rainfall
collect ( ) summarize ( )

Ground thermometer
gt_Ident temperature

Anemometer
an_Ident windSpeed windDirection

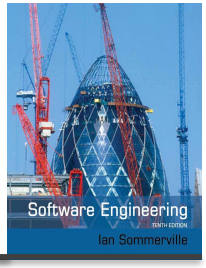
Barometer
bar_Ident pressure height

Design models



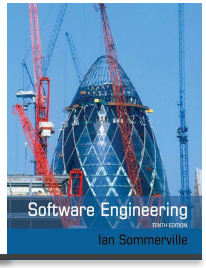
- Design models show the objects and object classes and relationships between these entities.
- There are two kinds of design model:
  - Structural models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

Examples of design models



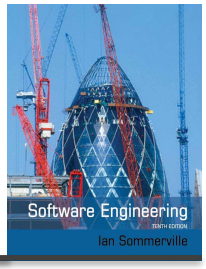
- **Subsystem models** that show logical groupings of objects into coherent subsystems.
- **Sequence models** that show the sequence of object interactions.
- **State machine models** that show how individual objects change their state in response to events.
- Other models include **use-case models aggregation models, generalisation models**, etc. ,

Subsystem models



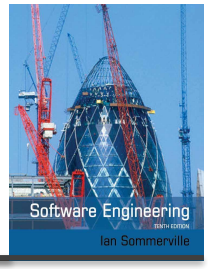
- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

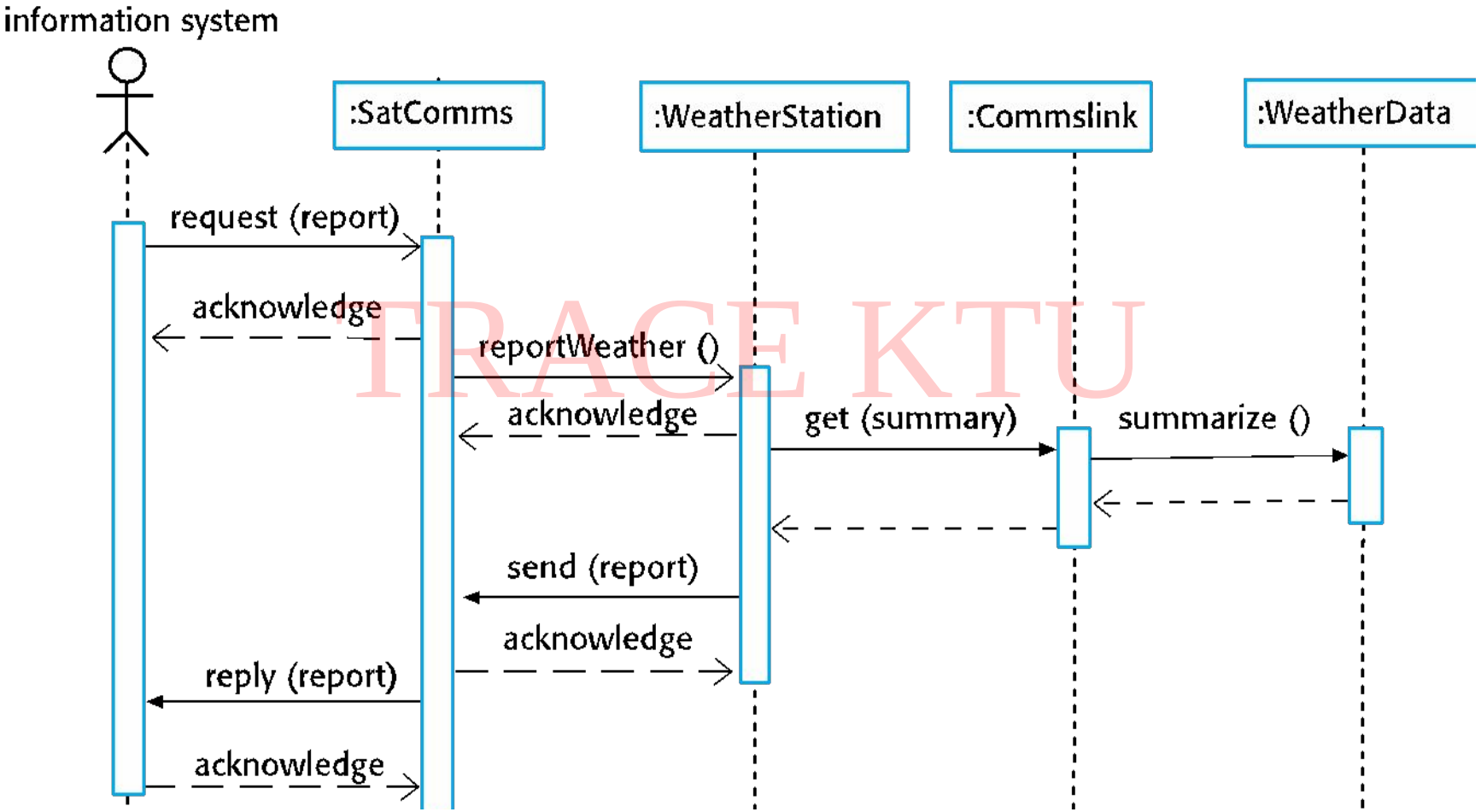
Sequence models



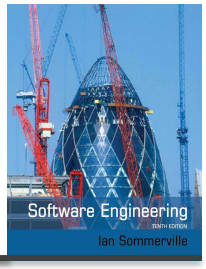
- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Sequence diagram describing data collection



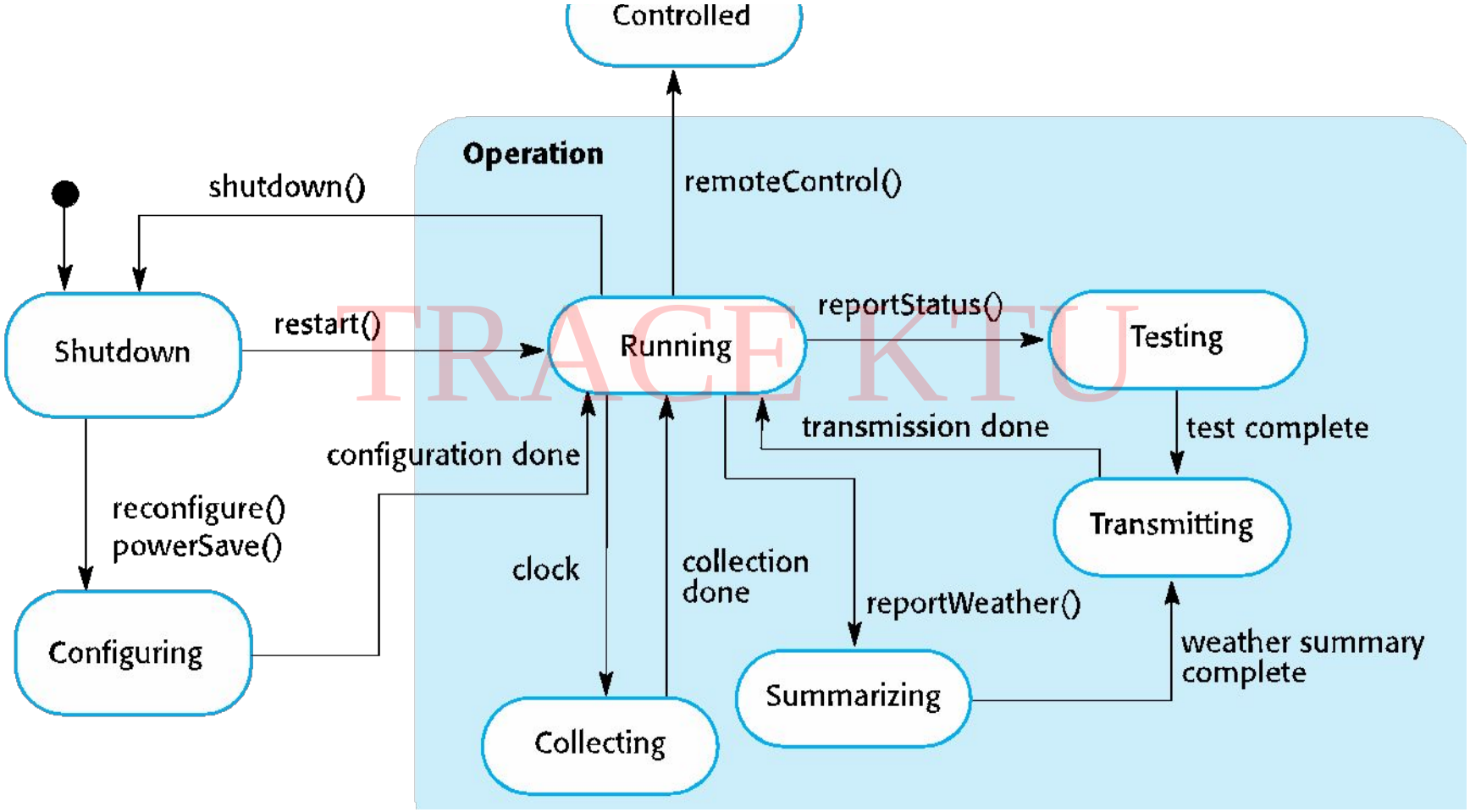


State diagrams

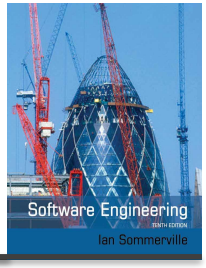


- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object's run-time behavior.
- You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram

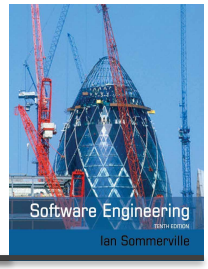


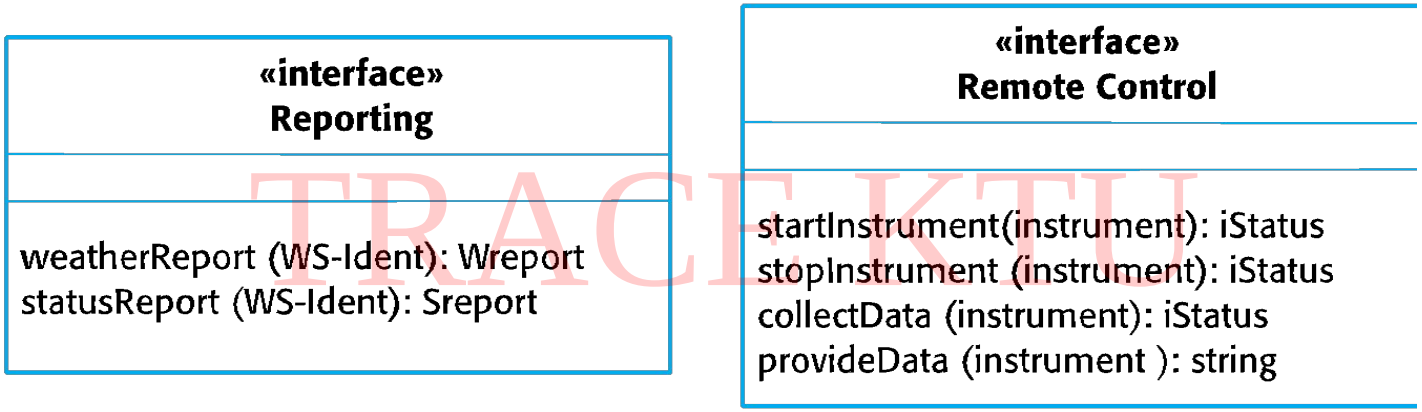
Interface specification

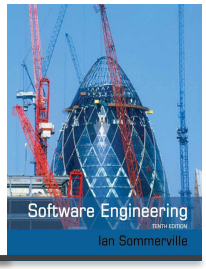


- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.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

Weather station interfaces

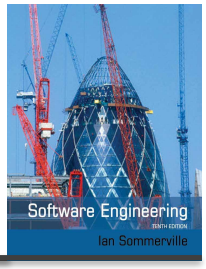






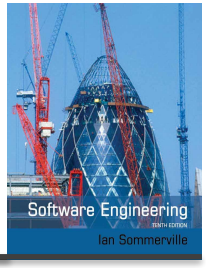
Design patterns

Design patterns



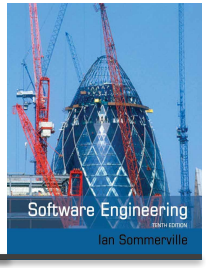
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented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

Patterns



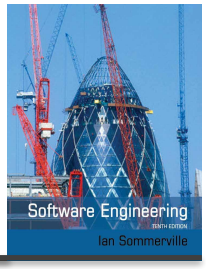
- *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

Pattern elements



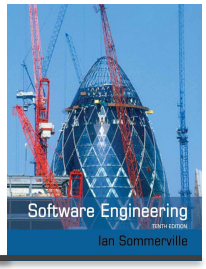
- Name
  - A meaningful pattern identifier.
- Problem description.
- Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
  - The results and trade-offs of applying the pattern.

The Observer pattern



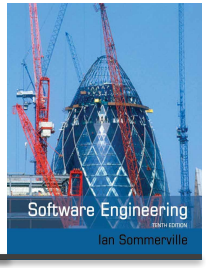
- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical.

The Observer pattern (1)



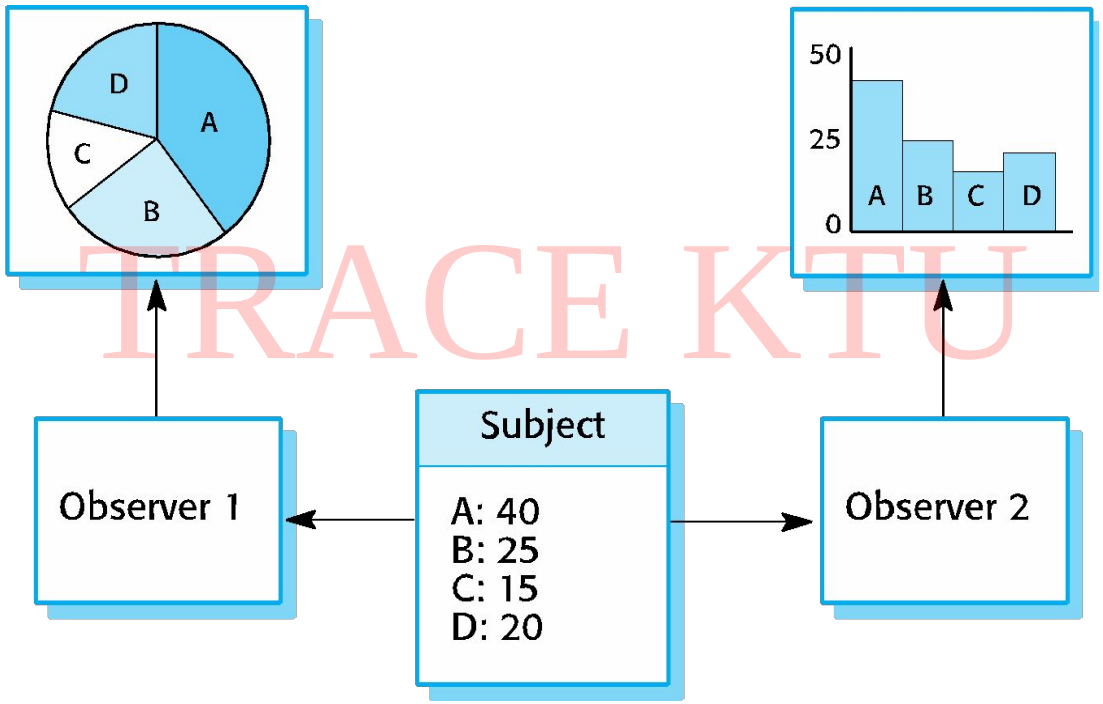
Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

The Observer pattern (2)

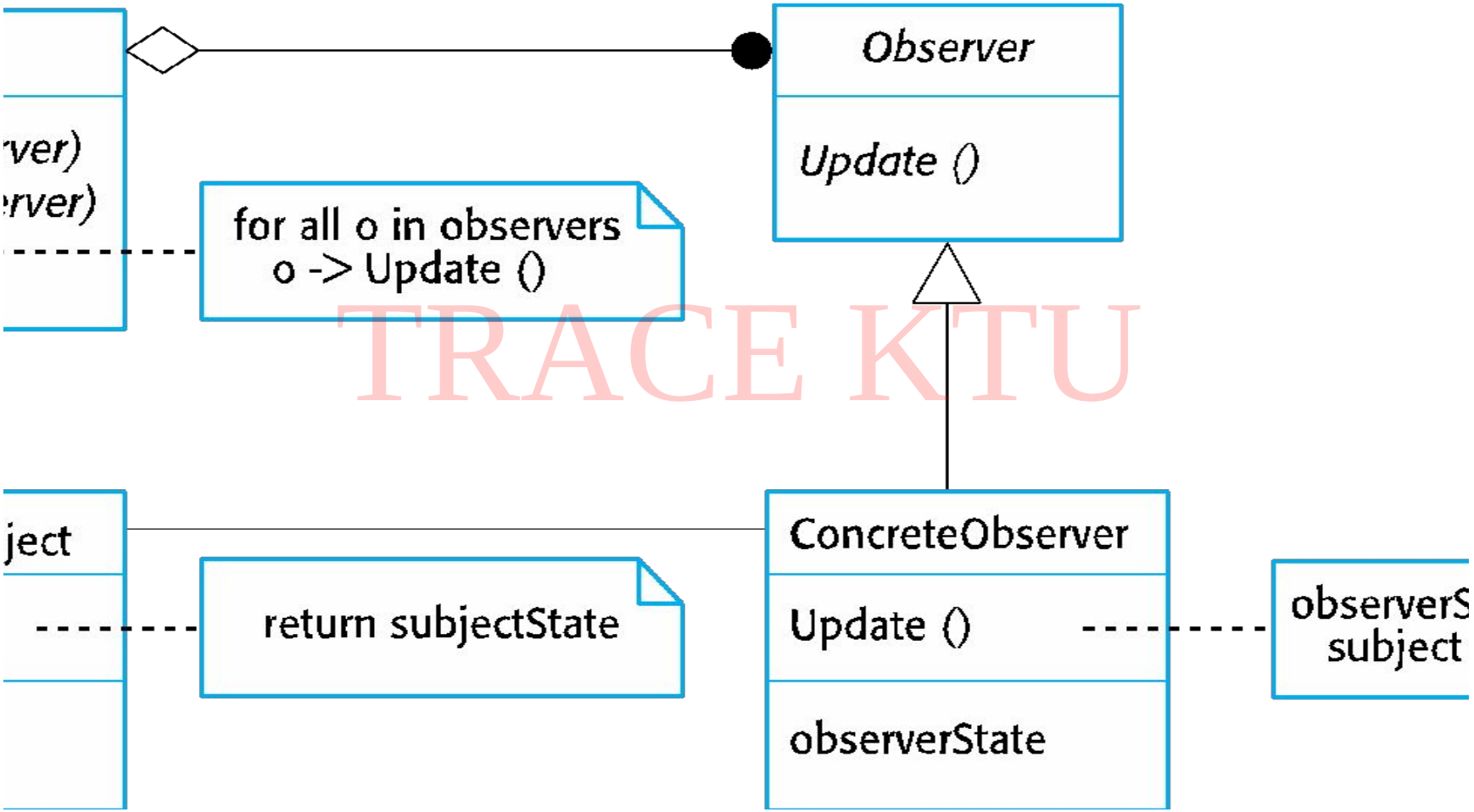


Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

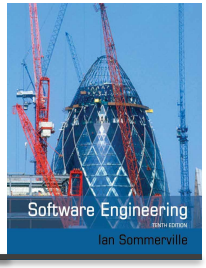
Multiple displays using the Observer pattern



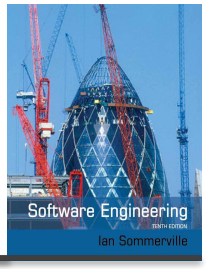
A UML model of the Observer pattern



Design problems

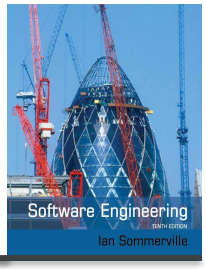


- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).



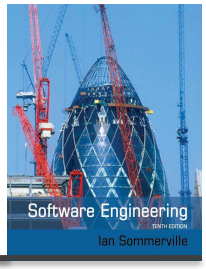
Implementation issues

Implementation issues



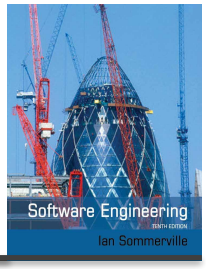
- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - 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).

Reuse



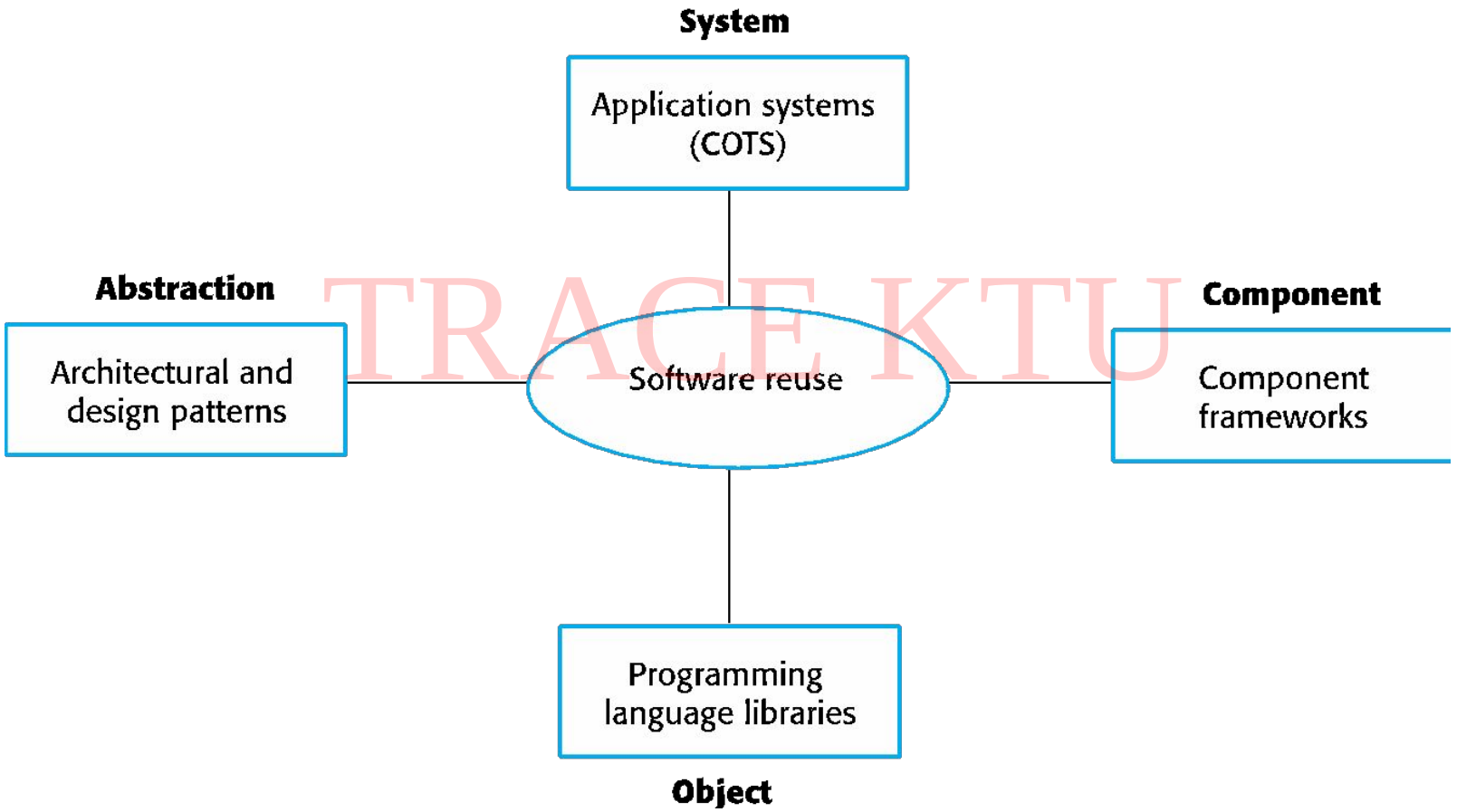
- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach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

Reuse levels

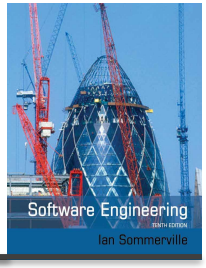


- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Software reuse

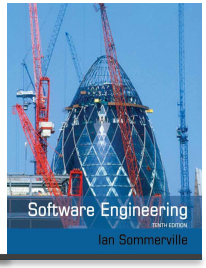


Reuse costs



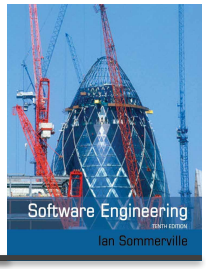
- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software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

Configuration management



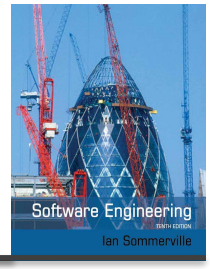
- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- See also Chapter 25.

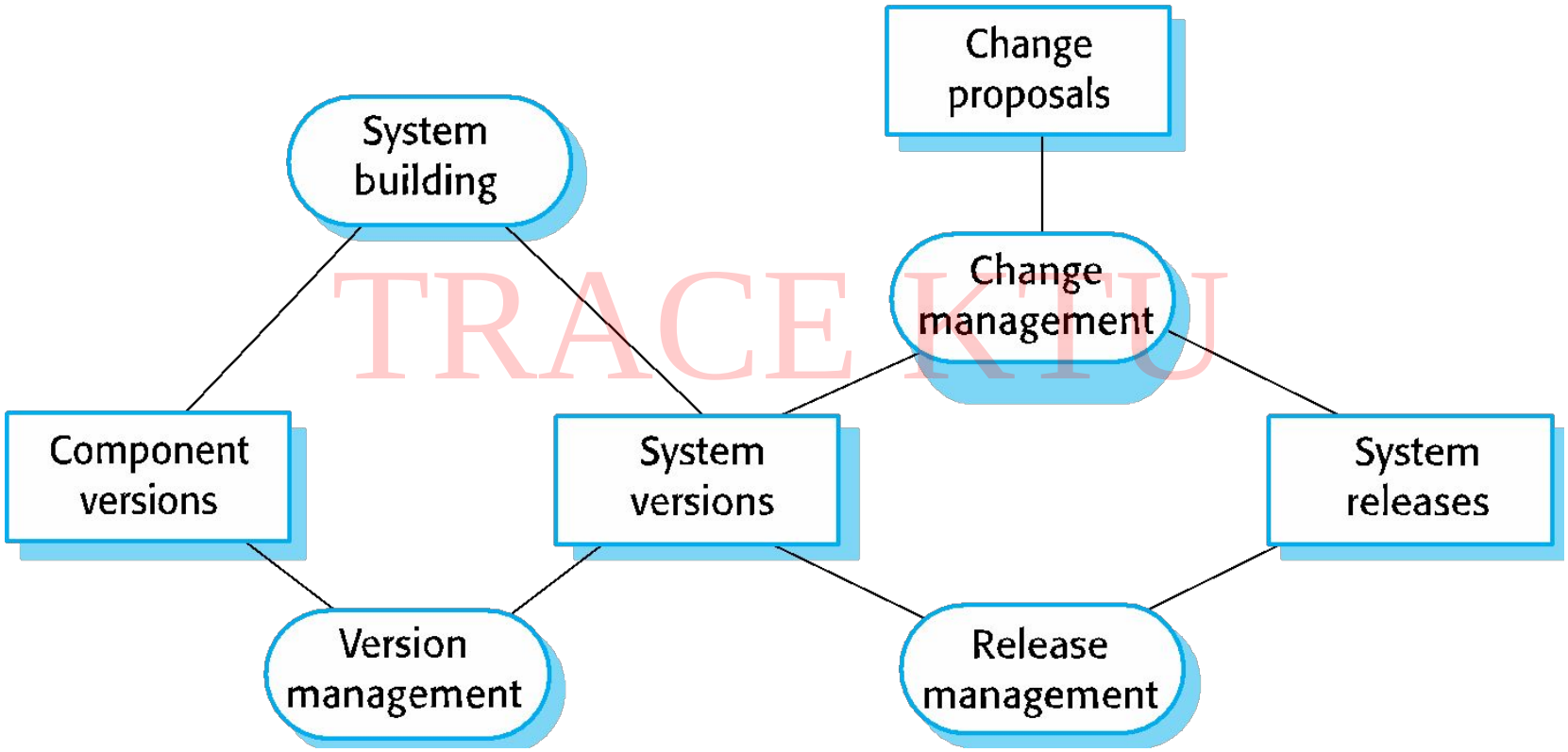
Configuration management activities



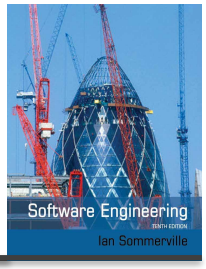
- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components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

Configuration management tool interaction



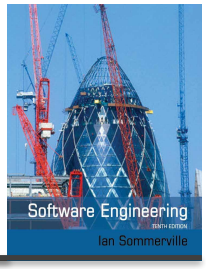


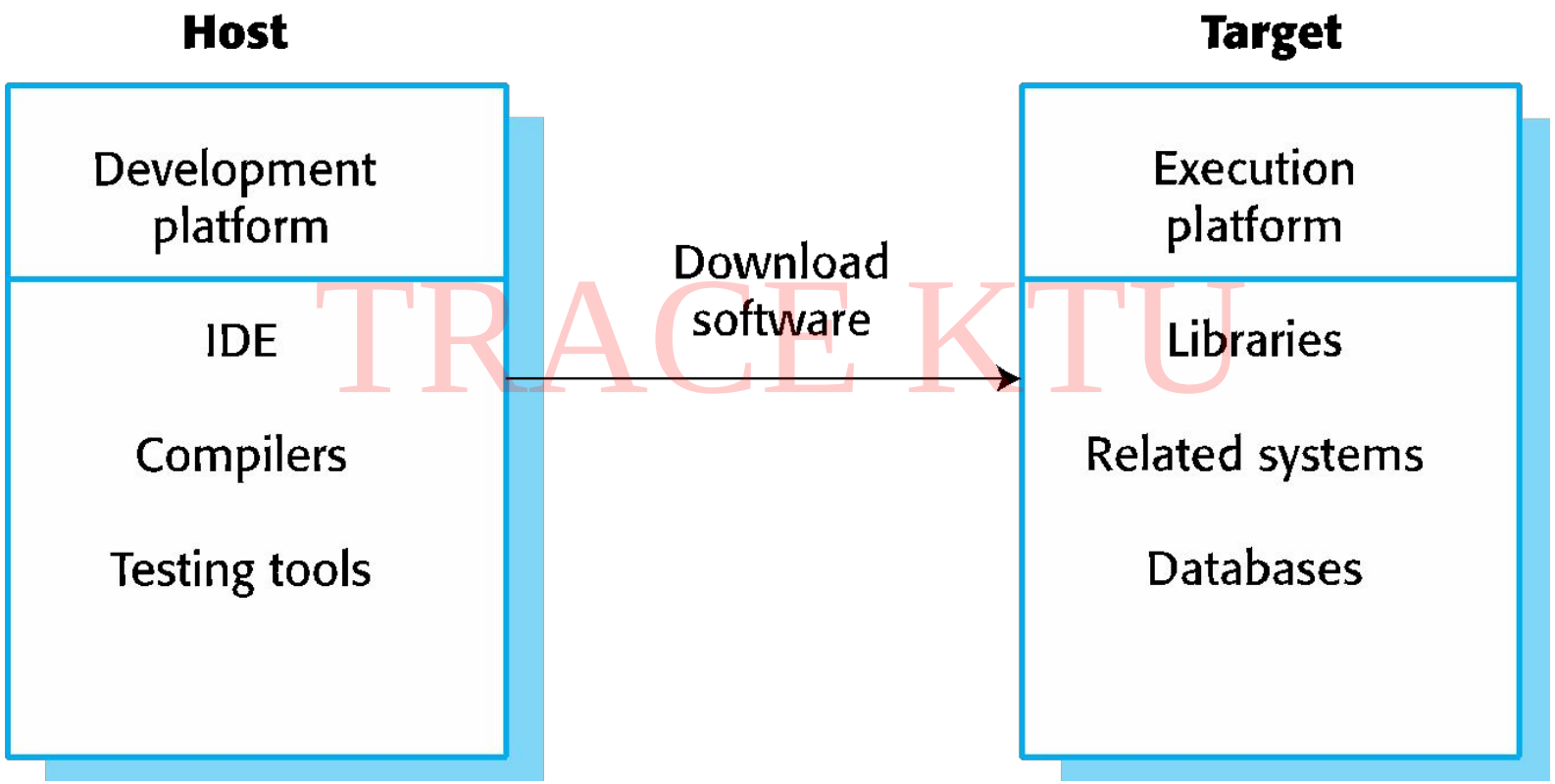
Host-target development



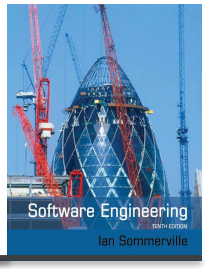
- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Host-target development



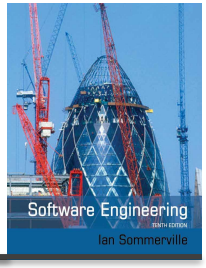


Development platform tools



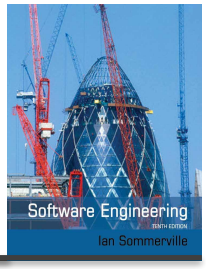
- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.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

Integrated development environments (IDEs)

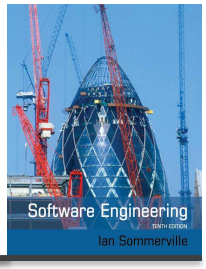


- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Component/system deployment factors

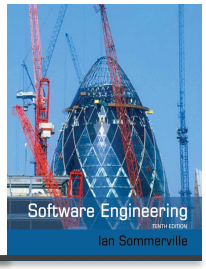


- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.



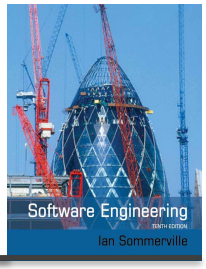
Open source development

Open source development



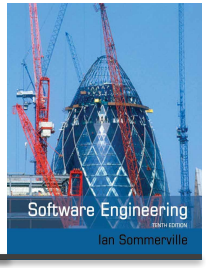
- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source systems



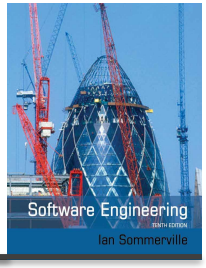
- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues



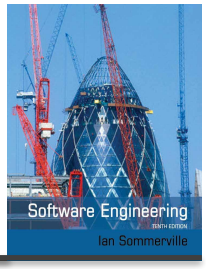
- Should the product that is being developed make use of open source components?
- Should an open source approach be used for the software's development?

Open source business



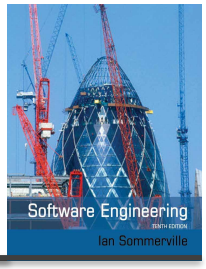
- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open source licensing



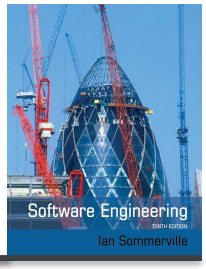
- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it 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.

License models



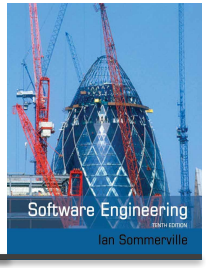
- The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you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

License management



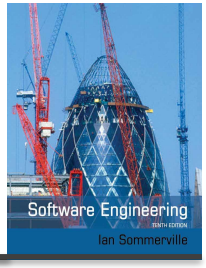
- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community.

Key points



- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

Key points



- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

Module 3
Review Techniques

Anit Thomas M
Assistant Professor
CSE, VJEC

Software Review

- 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 activity in the software process).
- The primary objective of formal 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 formal 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 during the software process. However, formal 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 steps in the development and support phases.
- To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects.

- Assume that an error uncovered during design will cost 1.0 monetary unit to correct.
- Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units; during testing, 15 units; and after release, between 60 and 100 units.

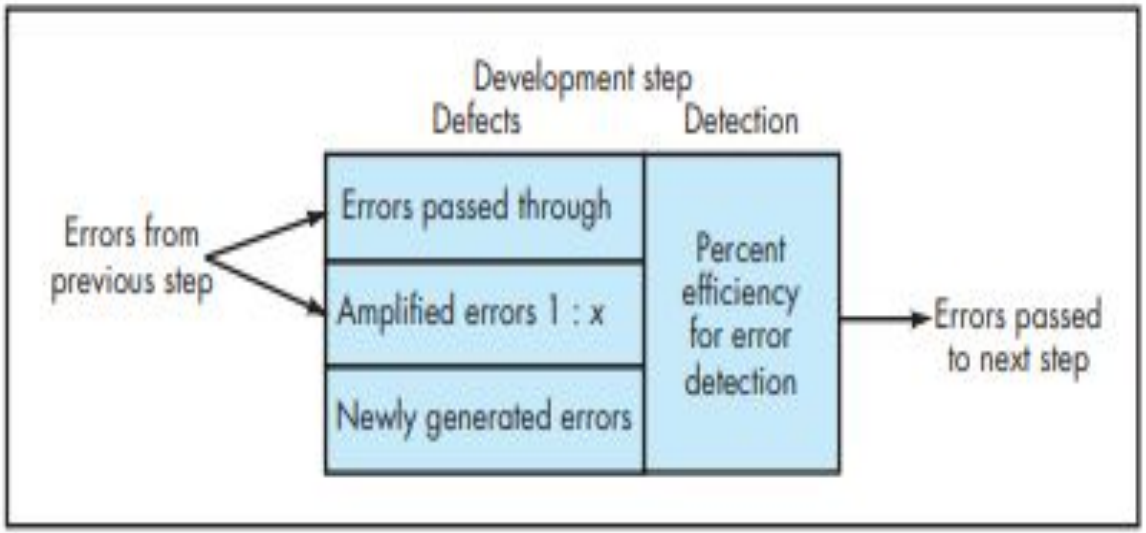
DEFECT AMPLIFICATION AND REMOVAL

- A defect amplification model [IBM81] 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 20.1 . 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

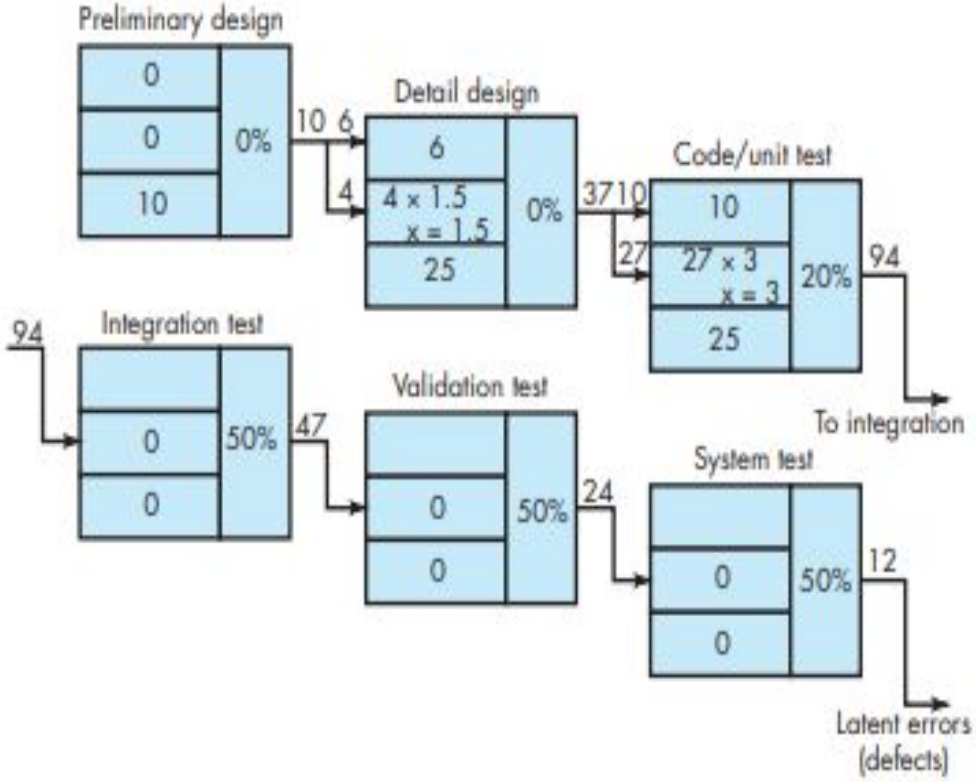
Defect amplification model



- Figure 20.2 illustrates a hypothetical example of defect amplification for a software process in which no reviews are conducted.
- Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption).
- Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field.

FIGURE 20.2

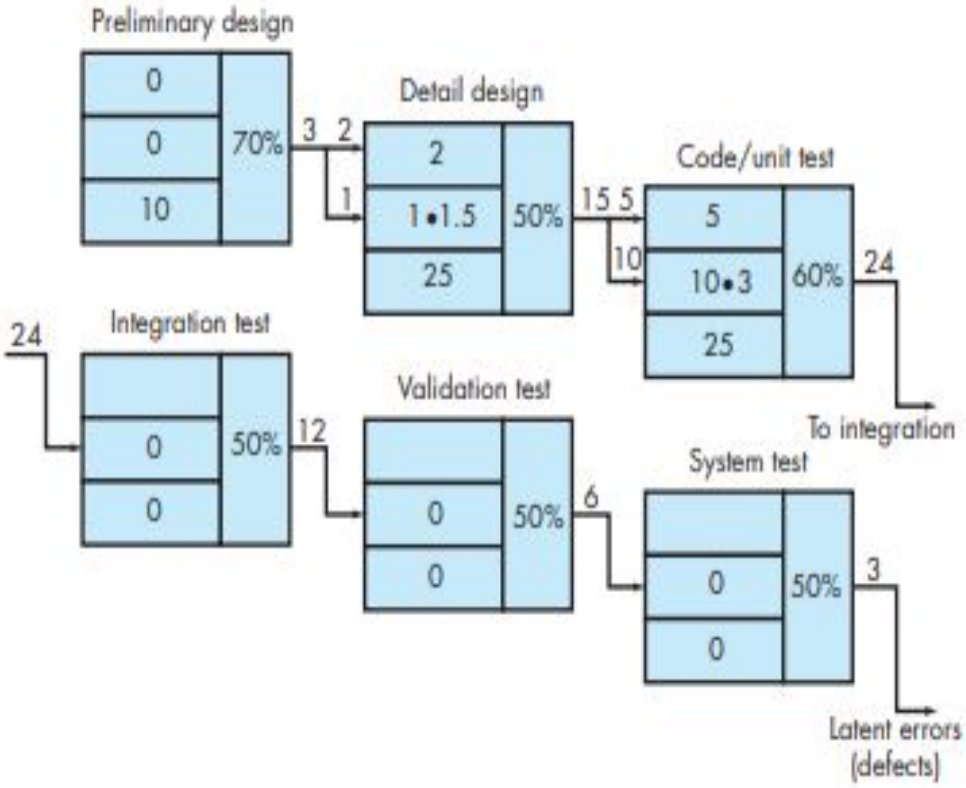
Defect amplification—no reviews



- Figure 20.3 considers the same conditions except that design and code reviews are conducted as part of each software engineering action.
- In this case, 10 initial preliminary (architectural) design errors are amplified to 24 errors before testing commences. Only three latent errors exist.
- The relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established.

FIGURE 20.3

Defect amplification—reviews conducted



- The number of errors uncovered during each of the steps noted in Figures 20.2 and 20.3 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).
- Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted, total cost is 2177 units—nearly three times more costly.
- To conduct reviews, you must expend time and effort, and your development organization must spend money.

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 that can be used to assess their efficacy.
- Although many metrics can be defined for technical reviews, a relatively small subset can provide useful insight.
- The following review metrics can be collected for each review that is conducted:

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

Analyzing Metrics

- Before analysis can begin, a few simple computations must occur.
- The total review effort and the total number of errors discovered are defined as:

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

$$Err_{tot} = Err_{minor} + Err_{major}$$

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

$$\text{Error density} = \frac{Err_{tot}}{WPS}$$

- For example, if a requirements model is reviewed to uncover errors, inconsistencies, and omissions, it would be possible to compute the error density in a number of different ways.
- The requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials.
- The review uncovers 18 minor errors and 4 major errors.
- Therefore, Err tot = 22. Error density is 1.2 errors per UML diagram or 0.68 errors per requirements model page.

- 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 (as yet unreviewed document).

- For example, if the average error density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long, a rough estimate suggests that your software team will find about 19 or 20 errors during the review of the document.
- If you find only 6 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.
- Once testing has been conducted, it is possible to collect additional error data, including the effort required to find 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).

- Returning to the example presented in the above section, the average error density for requirements models was determined to be 0.6 per page.
- The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours.
- The effort required for a major requirement error was found to be 18 person-hours.
- Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors.
- Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.

- Requirements-related errors uncovered during testing require an average of 45 person-hours to find and correct (no data are available on the relative severity of the error).
- Using the averages noted, we get:

$$\text{Effort saved per error} = E_{\text{testing}} - E_{\text{reviews}}$$

$$45 - 6 = 30 \text{ person-hours/error}$$

- Since 22 errors were found during the review of the requirements model, a saving of about 660 person-hours of testing effort would be achieved.
- And that's just for requirements-related errors.
- Errors associated with design and code would add to the overall benefit.
- The bottom line—effort saved leads to shorter delivery cycles and improved time to market.

FORMAL TECHNICAL REVIEWS

- A formal technical review (FTR) is a software quality assurance activity performed by software engineers (and others).
- The objectives of the 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, inspections, round-robin reviews and other small group technical assessments of software.
- 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:
- 1. Between three and five people (typically) should be involved in the review.
- 2. Advance preparation should occur but should require no more than two hours of work for each person.
- 3. 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.
- By narrowing 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 specification, a detailed component design, a source code listing 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 the *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 are discovered, the recorder notes each. 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).

- The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

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. It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

Review Guidelines

- Guidelines for the conduct of 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 that 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.

- **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 test documents.

- **8.** ***Allocate resources and schedule time for FTRs.*** For reviews to be effective, they should be scheduled as a task during the software engineering 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 real world of s/w projects, resources are limited and time is short.
- In such situations, reviews are often skipped.
- In this, samples of all s/w work products are inspected to determine which work products are more error prone.
- Full FTR resources are then focused only to those work products that are likely to be error-prone .
- Sample driven review must attempt to quantify those work products that are primary targets for full FTRs.

- Following are the suggested steps:
  - Inspect a fraction a_i of each software work product, i. Record the number of faults, f_i found within a_i
  - Develop a gross estimate of the no. of faults within the work product i by multiplying f_i by $1/ a_i$
  - Sort the work products in descending order according to the gross estimate of the number of faults in each.
  - Focus available review resources on those work products that have the highest estimated number of faults

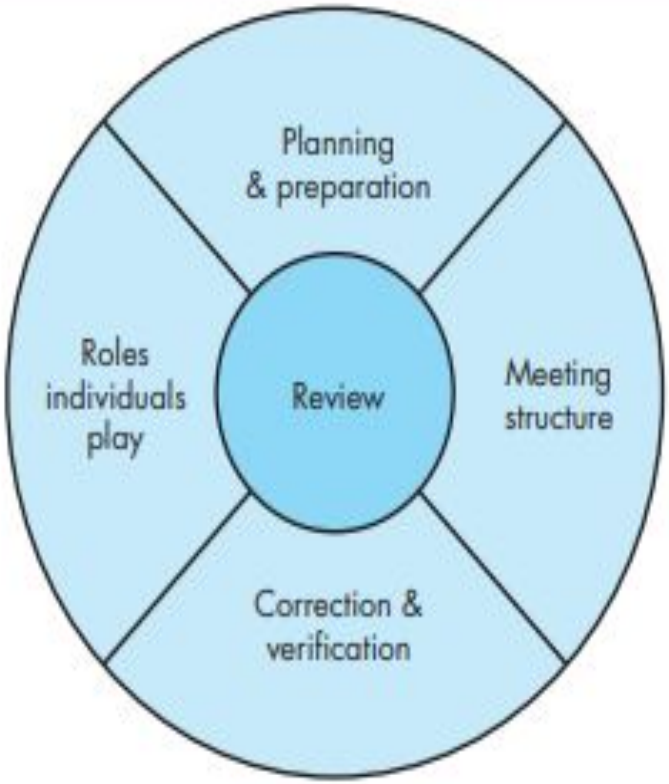
- The fraction of work product that is sampled must be
  - representative of the work product as a whole and
  - Large enough to be meaningful to the reviewer(s) who does the sampling.

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

FIGURE 20.5

Reference model for 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.
- 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 labelled?

and so on.

- 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.

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
Testing

Anit Thomas M
Assistant Professor
CSE, VJEC

Software 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.
- Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture.
- These techniques provide systematic guidance for designing tests that (1) exercise the internal logic and interfaces of every software component and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

- During early stages of testing, a software engineer performs all tests.
- However, as the testing process progresses, testing specialists may become involved.

Why is it important?

- Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors.
- To find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps?

- For conventional applications, software is tested from two different perspectives:
- (1) internal program logic is exercised using “white box” test-case design techniques and
- (2) software requirements are exercised using “black box” test-case design techniques.
- Use cases assist in the design of tests to uncover errors at the software validation level.
- In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.

What is the work product?

- A set of test cases designed to exercise internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

How do I ensure that I've done it right?

- When you begin testing, change your point of view.
- Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness.
- In addition, you can evaluate test coverage and track error detection activities.

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.
- At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.
- **Testability:** James Bach provides the following definition for testability: “ Software testability is simply how easily [a computer program] can be tested.”

The following characteristics lead to testable software:

- **Operability.** "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.
- **Observability.** "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

- **Controllability.** "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.
- **Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

- **Simplicity.** "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).
- **Stability.** "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.
- **Understandability.** "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

Test Characteristics.

- A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
- A good test should be “best of breed”. In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.
- A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

INTERNAL AND EXTERNAL VIEWS OF TESTING

- Any engineered product (and most other things) 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

- 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

- 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: Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 23.1

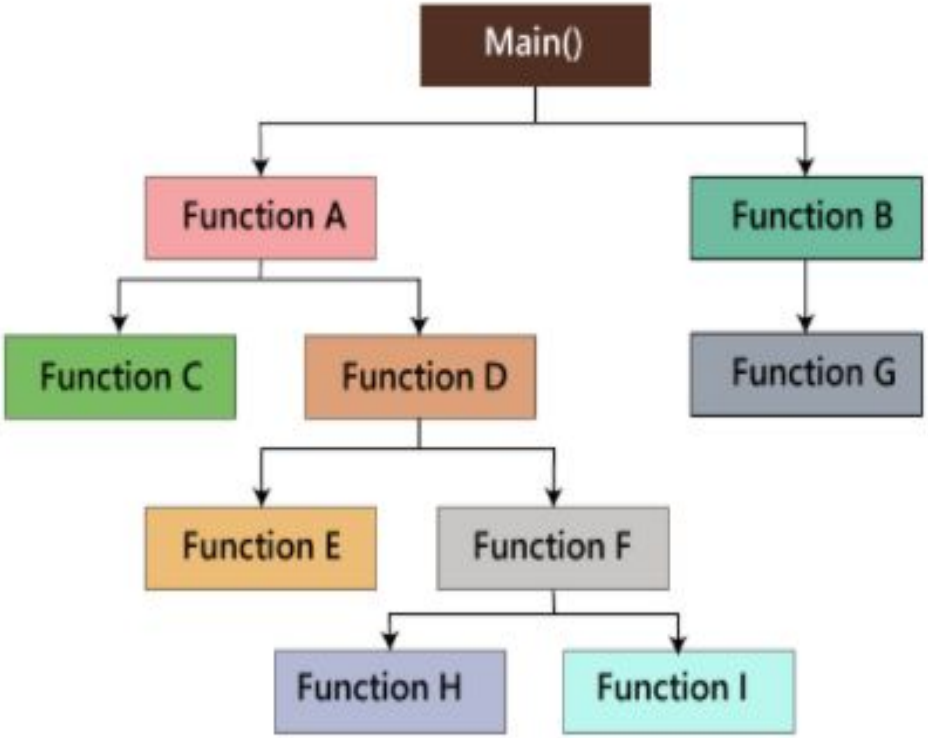

Main()
Function A
Function B
Function C
Function D
Function G
Function E
Function F
Function H
Function I

FIGURE 23.1

Flow graph notation

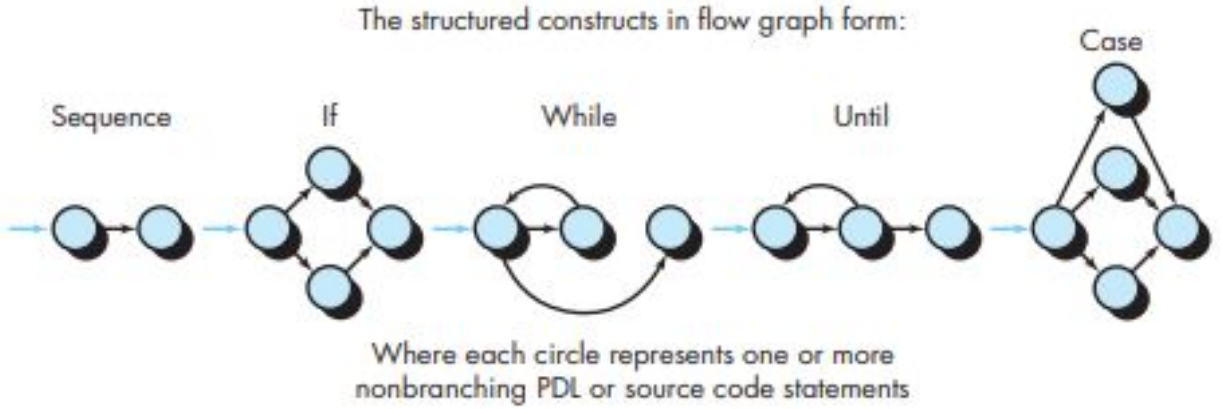
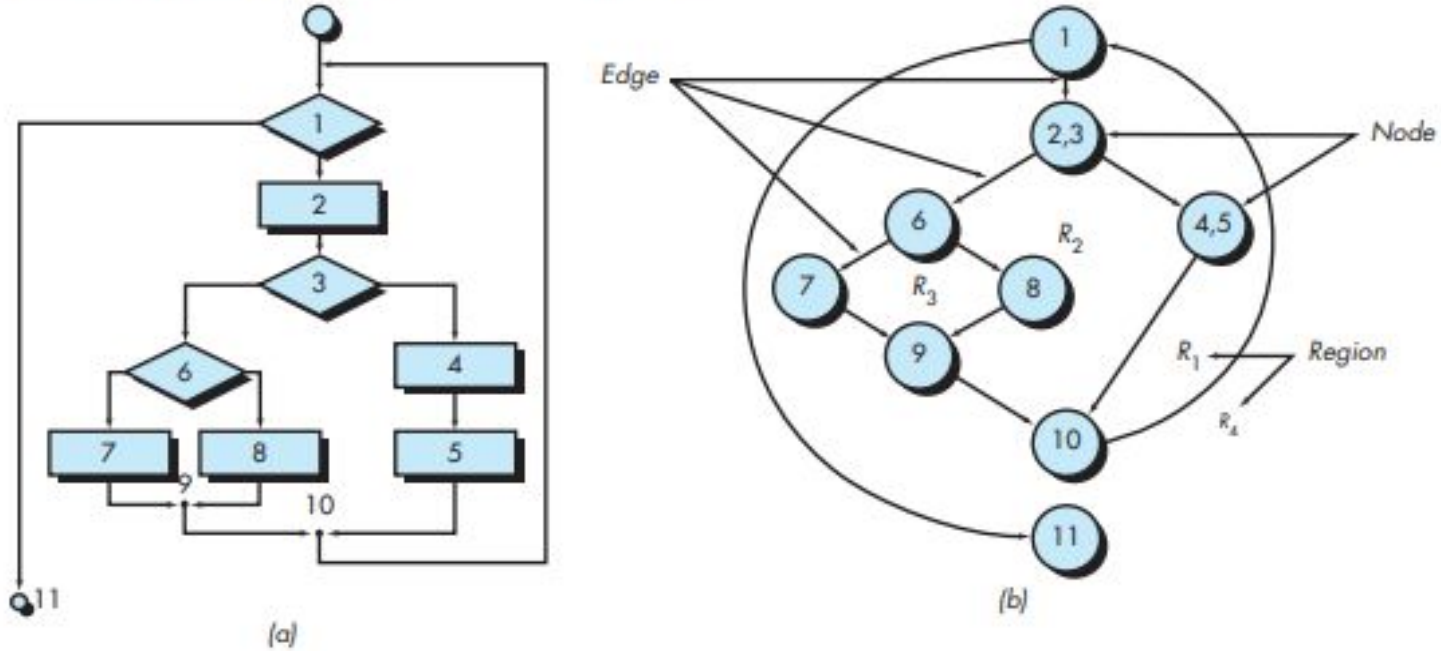


FIGURE 23.2 (a) Flowchart and (b) flow graph



- To illustrate the use of a flow graph, consider the procedural design representation in Figure 23.2a .
- Here, a flowchart is used to depict program control structure. Figure 23.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
- Referring to Figure 23.2b , 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 flow 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.

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 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

- The path given above is not considered to be an independent path because it is simply a combination of already specified 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.

- How do you know how many paths to look for?
- The computation of cyclomatic complexity provides the answer. 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.

- A predicate node is **a node with more than one edge emanating from it**.
- Referring once more to the 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 23.2b is 4.
- More important, 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

FIGURE 23.4

PDL with nodes identified

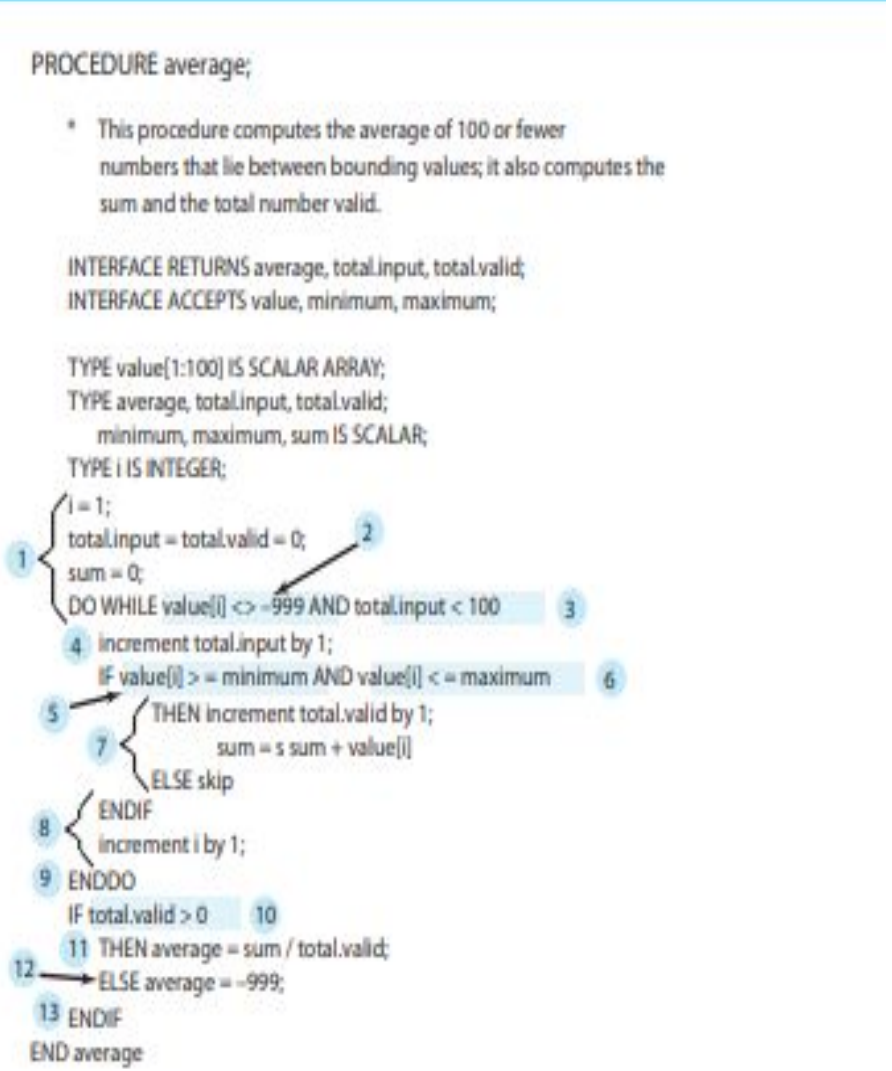
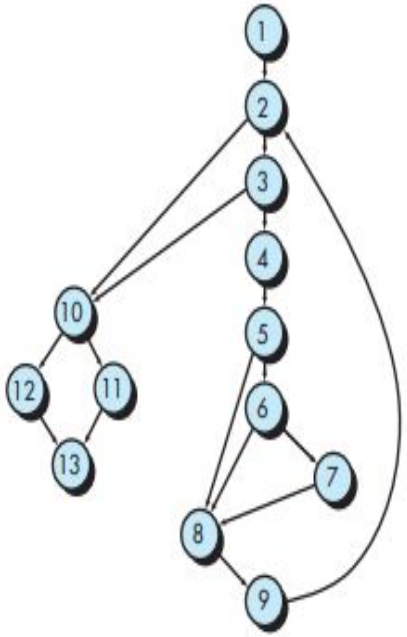


FIGURE 23.5

Flow graph for the procedure average



without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure 23.5 ,

V(G) = 6 regions

V(G) = 17 edges -n13 nodes + 2 = 6

V(G) = 5 predicate nodes + 1 = 6

- 3. Determine a basis set of linearly independent paths.
- The value of V(G) provides the number of linearly independent paths through the program control structure.
- In the case of procedure average, we expect to specify six paths:
- 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-. . .

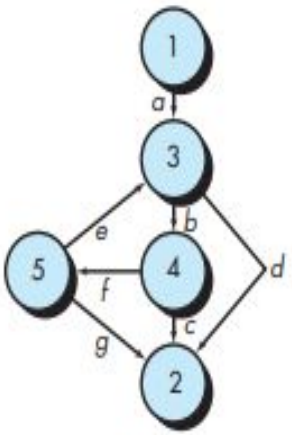
- 4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

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.

- A simple example of a flow graph and its corresponding graph matrix is shown in Figure 23.6 .

FIGURE 23.6
Graph matrix



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

- Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

- To this point, 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:

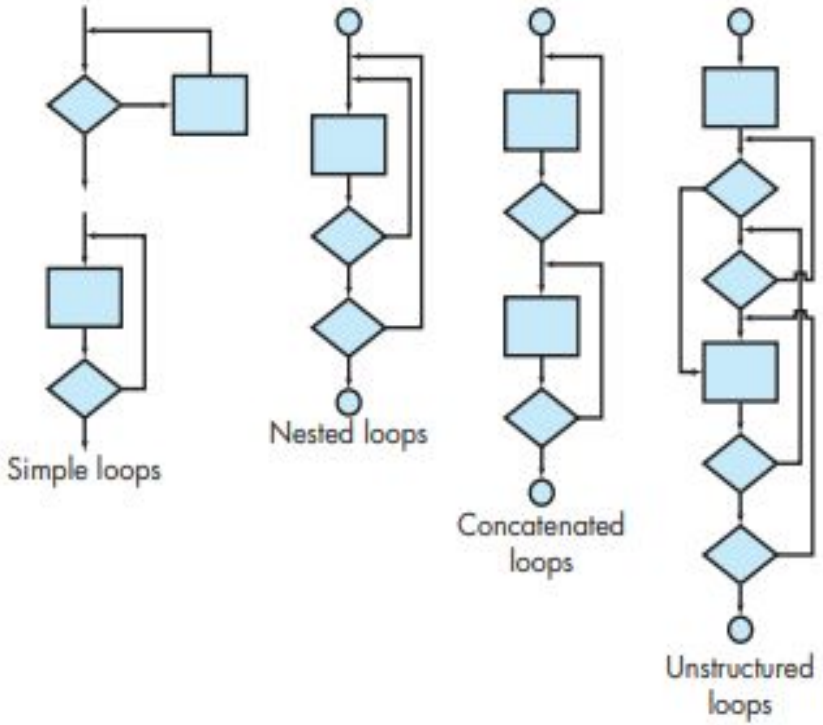
- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

CONTROL STRUCTURE TESTING

- The basis path testing technique described above is one of a number of techniques for control structure testing.
- Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. 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, and unstructured loops (Figure 23.7).

FIGURE 23.7

Classes of Loops



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 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.

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.

- 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 specifi c test at hand.

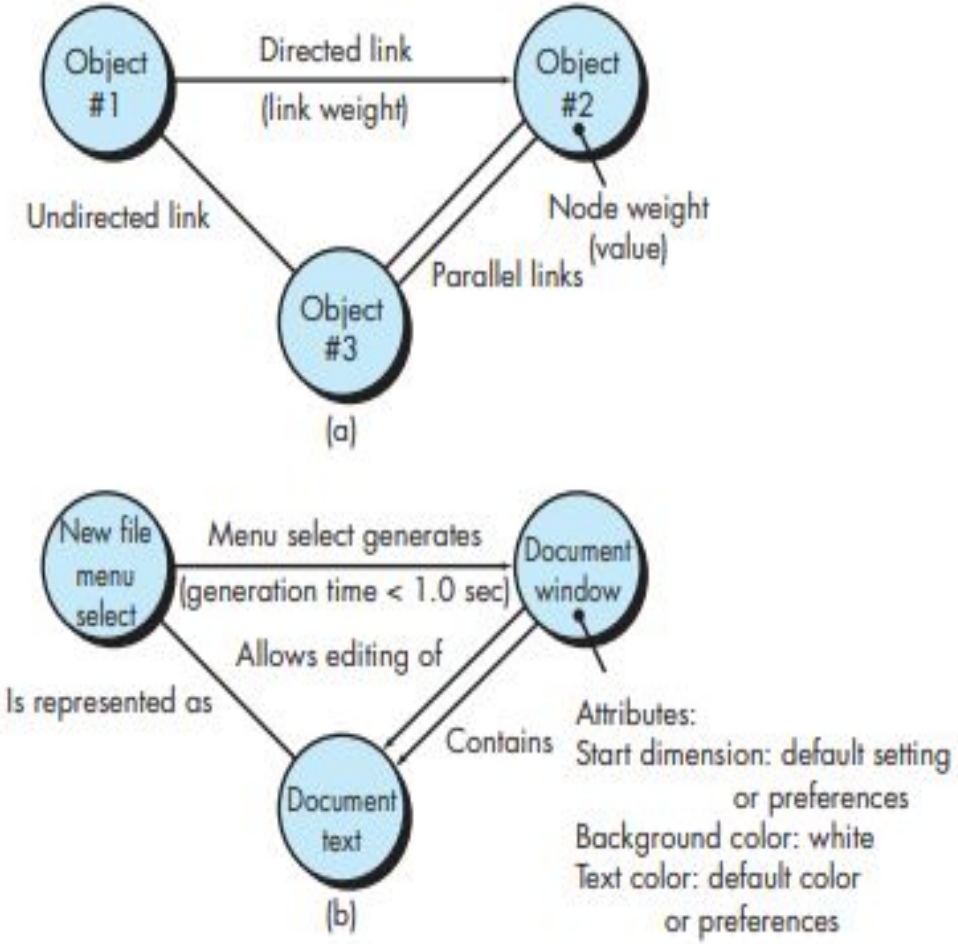
Graph-Based Testing Methods

- 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” .

- Stated in another way, 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.
- The symbolic representation of a graph is shown in Figure 23.8a . 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



- 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 = document Window

  Object #3 = document Text

- Referring to the figure, a menu select on newFile generates a document window. The node weight of documentWindow provides a list of the window attributes that are to be expected when the window is generated.
- The link weight indicates that the window must be generated in less than 1.0 second.
- An undirected link establishes a symmetric relationship between the new File menu selection and documentText, and parallel links indicate relationships between document Window and documentText.

- In reality, a far more detailed graph would have to be generated as a precursor to test-case design.
- You can then derive test cases by traversing the graph and covering each of the relationships shown.
- These test cases are designed in an attempt to fi nd errors in any of the relationships.
- Beizer describes a number of 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. For example, a data object flight InformationInput is followed by the operation validation AvailabilityProcessing().

- **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 (e.g., orderInformation is verifi ed during inventoryAvailabilityLook-up() and is followed by customerBillingInformation input). 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. For example, the node FICATaxWithheld (FTW) is computed from gross wages (GW) using the relationship, FTW = 0.62 * GW.

- **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. Using concepts introduced in the preceding section, 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.

- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:
- 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. For example, assume that a temperature versus pressure table is required as output from an engineering analys 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 analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

- 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

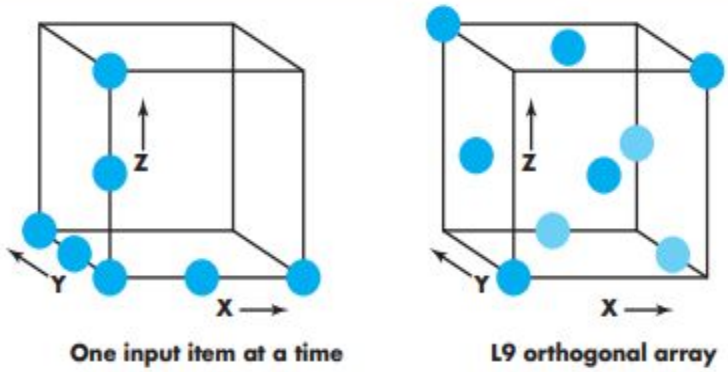
- Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.
- There are many applications in which the input domain is relatively limited.
- That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded.
- When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain.
- However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

- 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.
- To illustrate the difference between orthogonal array testing and more conventional “one input item at a time” approaches, 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).

FIGURE 23.9

A geometric view of test cases

Source: [Pha97].



- 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.

- 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.
- 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

- 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 specifi c 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.

TESTING DOCUMENTATION AND HELP FACILITIES

- The term software testing conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate.
- But errors in help facilities or program documentation can be as devastating to the acceptance of the program as errors in data or source code.
- Nothing is more frustrating than following a user guide or an online help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation.
- It is for this reason that 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.
- Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier.
- 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.

Security Testing

What is Security Testing?

- **Security Testing** is a type of Software Testing that uncovers vulnerabilities, threats, risks in a software application and prevents malicious attacks from intruders. The purpose of Security Tests is to identify all possible loopholes and weaknesses of the software system which might result in a loss of information, revenue, repute at the hands of the employees or outsiders of the Organization.

Why Security Testing is Important?

- The main goal of **Security Testing** is to identify the threats in the system and measure its potential vulnerabilities, so the threats can be encountered and the system does not stop functioning or can not be exploited. It also helps in detecting all possible security risks in the system and helps developers to fix the problems through coding.

Types of Security Testing:

- There are seven main types of security testing as per Open Source Security Testing methodology manual. They are explained as follows:
- **Vulnerability Scanning**: This is done through automated software to scan a system against known vulnerability signatures.
- **Security Scanning:** It involves identifying network and system weaknesses, and later provides solutions for reducing these risks. This scanning can be performed for both Manual and Automated scanning.
- **Penetration testing**: This kind of testing simulates an attack from a malicious hacker. This testing involves analysis of a particular system to check for potential vulnerabilities to an external hacking attempt.
- **Risk Assessment:** This testing involves analysis of security risks observed in the organization. Risks are classified as Low, Medium and High. This testing recommends controls and measures to reduce the risk.

- **Security Auditing:** This is an internal inspection of Applications and Operating systems for security flaws. An audit can also be done via line by line inspection of code
- **Ethical hacking:** It's hacking an Organization Software systems. Unlike malicious hackers, who steal for their own gains, the intent is to expose security flaws in the system.
- **Posture Assessment:** This combines Security scanning, Ethical Hacking and Risk Assessments to show an overall security posture of an organization.

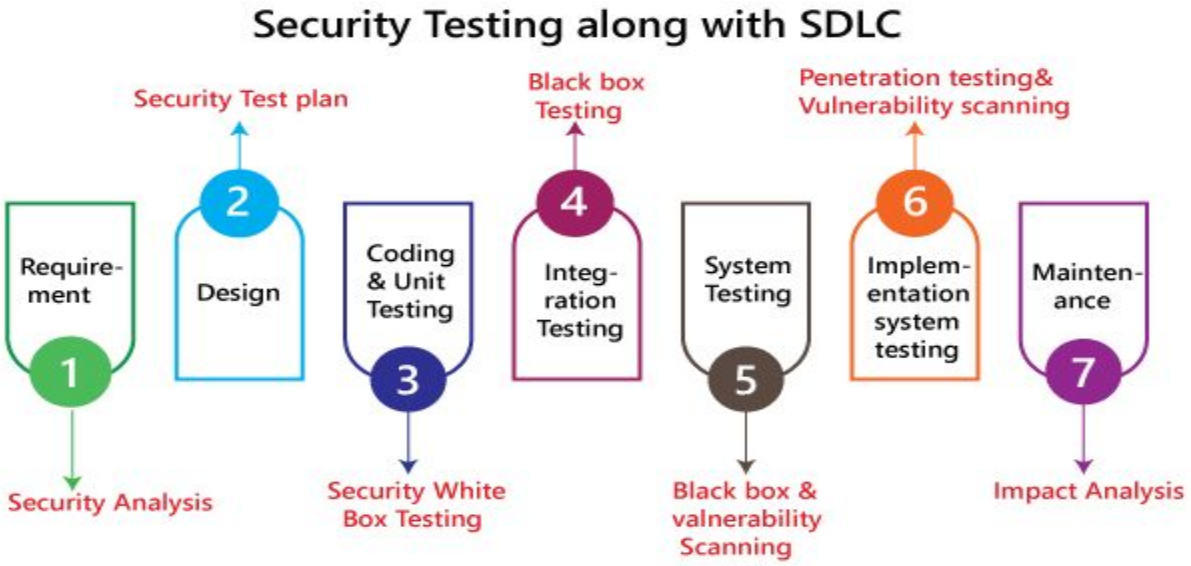
Principle of Security testing

- Here, we will discuss the following aspects of security testing:
- **Availability:** In this, the data must be retained by an official person, and they also guarantee that the data and statement services will be ready to use whenever we need it.
- **Integrity:** In this, we will secure those data which have been changed by the unofficial person. The primary objective of integrity is to permit the receiver to control the data that is given by the system.
- The integrity systems regularly use some of the similar fundamental approaches as confidentiality structures. Still, they generally include the data for the communication to create the source of an algorithmic check rather than encrypting all of the communication. And also verify that correct data is conveyed from one application to another.

- **Authorization:** It is the process of defining that a client is permitted to perform an action and also receive the services. The example of authorization is Access control.
- **Confidentiality:** It is a security process that protracts the leak of the data from the outsider's because it is the only way where we can make sure the security of our data.
- **Authentication:** The authentication process comprises confirming the individuality of a person, tracing the source of a product that is necessary to allow access to the private information or the system.
- **Non- repudiation:** It is used as a reference to the digital security, and it a way of assurance that the sender of a message cannot disagree with having sent the message and that the recipient cannot repudiate having received the message. The non-repudiation is used to ensure that a conveyed message has been sent and received by the person who claims to have sent and received the message.

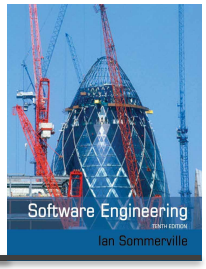
How we perform security testing

- The security testing is needed to be done in the initial stages of the software development life cycle because if we perform security testing after the software execution stage and the deployment stage of the SDLC, it will cost us more.
- Now let us understand how we perform security testing parallel in each stage of the software development life cycle(SDLC).



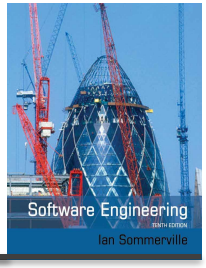
- **Steps**
- **SDLC:** Requirement stage: **Security Procedures:** In the requirement phase of SDLC, we will do the security analysis of the business needs and also verify that which cases are manipulative and waste.
- **Step2: SDLC:** Design stage
- **Security Procedures:** In the design phase of SDLC, we will do the **security testing for risk** exploration of the design and also embraces the security tests at the development of the test plan.
- **Step3: SDLC:** Development or coding stage
- **Security Procedures:** In the coding phase of SDLC, we will perform the white box testing along with static and dynamic testing.
- **Step4: SDLC:** Testing (functional testing, integration testing, system testing) stage
- **Security Procedures:** In the testing phase of SDLC, we will do one round of **vulnerability scanning** along with black-box testing.

- **Step 5: SDLC:** Implementation stage:
- **Security Procedures:** In the implementation phase of SDLC, we will perform **vulnerability scanning** again and also perform one round of **penetration testing**.
- **Step 6: SDLC:** Maintenance stage
- **Security Procedures:** In the Maintenance phase of SDLC, we will do the **impact analysis** of impact areas.
- And the **test plan** should contain the following:
- The test data should be linked to security testing.
- For security testing, we need the test tools.
- With the help of various security tools, we can analyze several test outputs.
- Write the test scenarios or test cases that rely on security purposes.



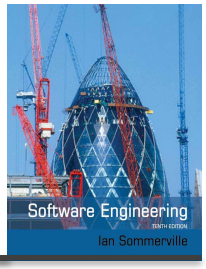
Chapter 9 – Software Evolution

Topics covered

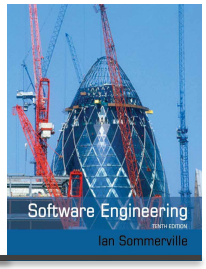


- Evolution processes
- Legacy systems
- Software maintenance

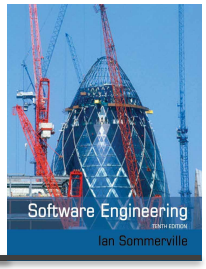
Software Evolution



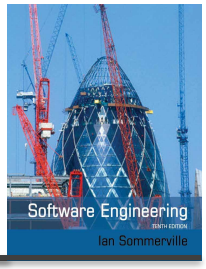
- Large software systems usually have a long lifetime.
- For example, military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more.
- Business systems are often more than 10 years old. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment.
- Successful software products and apps may have been introduced many years ago with new versions released every few 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

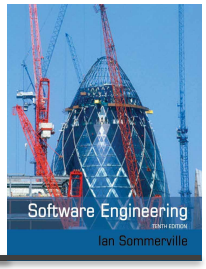


- Businesses have to change their software to ensure that they continue to get value from it.
- Their systems are critical business assets, and they have to invest in change to maintain the value of these assets.
- Consequently, most large companies spend more on maintaining existing systems than on new systems development.
- Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader “system of systems.”

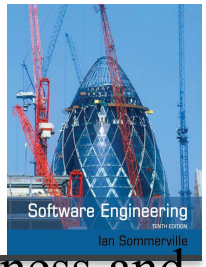


- In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems.
- Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.
- Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment.

Software change

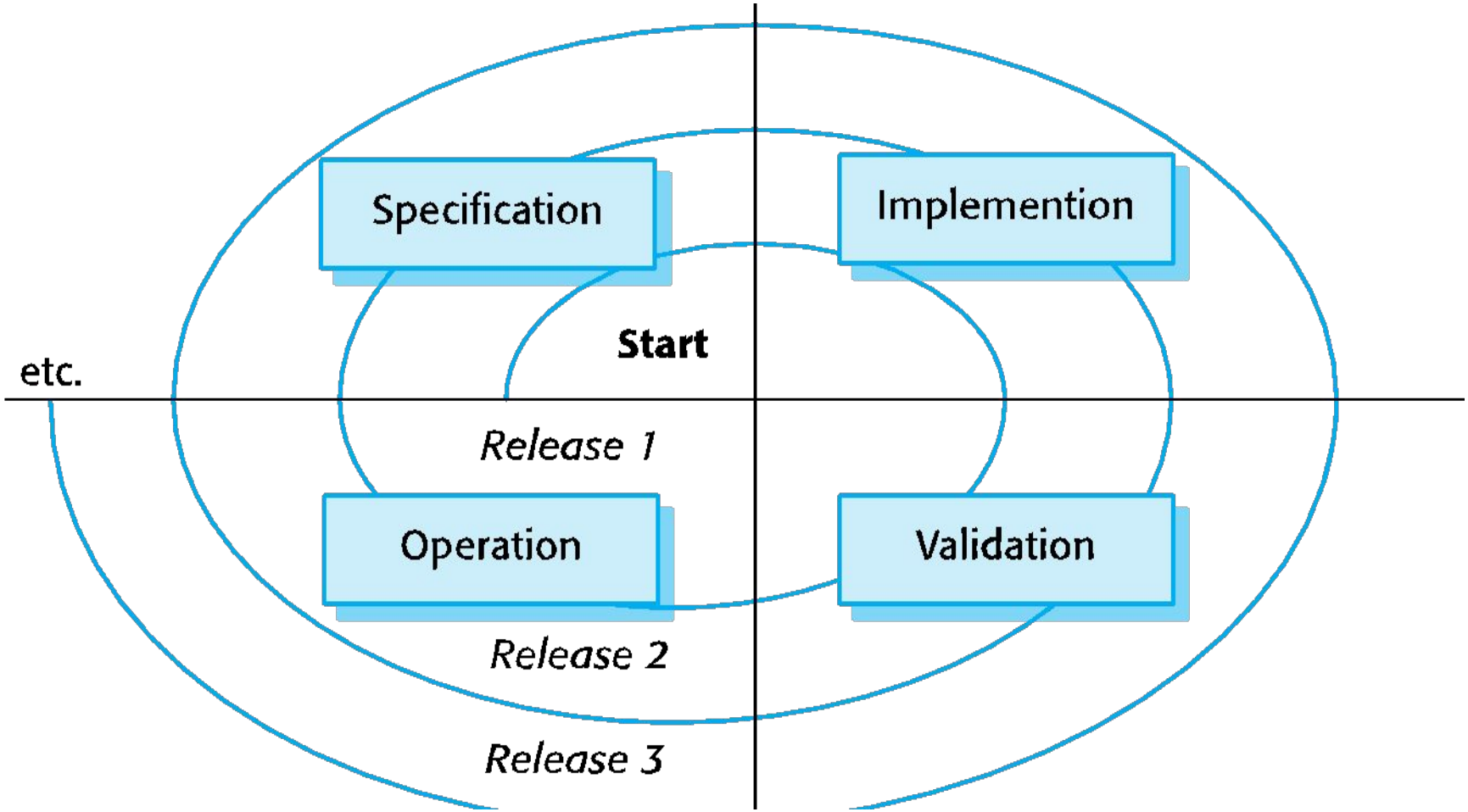


- Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for all organizations is implementing and managing change to their existing software systems.

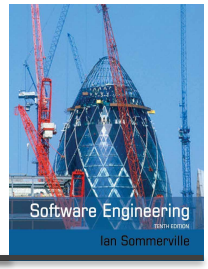


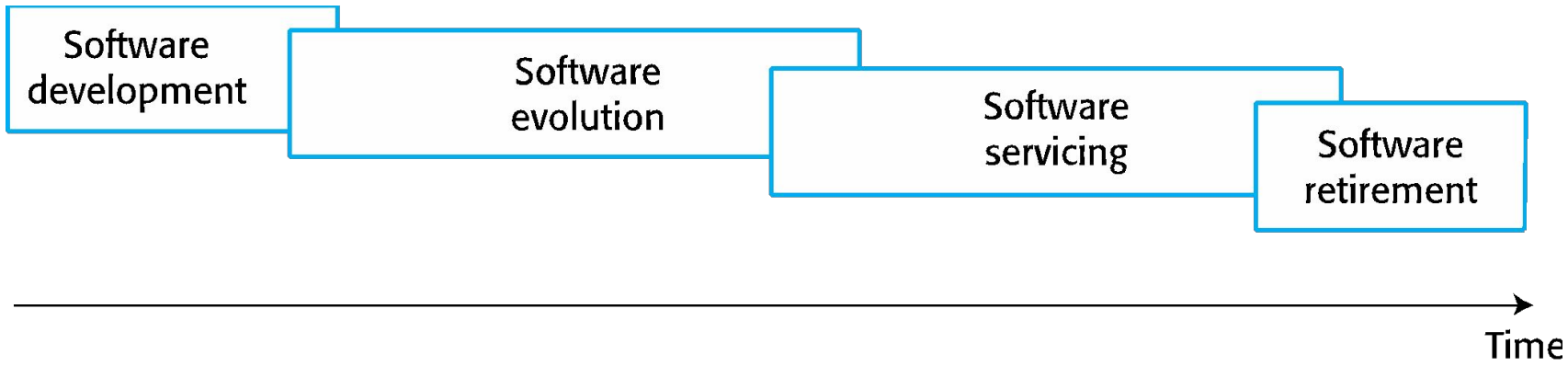
- The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals.
- Software engineering is therefore a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure below).
- 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.

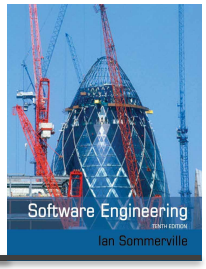
A spiral model of development and evolution



Evolution and servicing

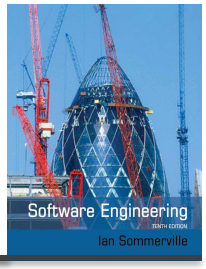




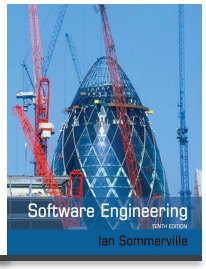


- During the servicing phase, the software is still useful, but only small tactical changes are made to it.
- During this stage, the company is usually considering how the software can be replaced.
- In the final stage, the software may still be used, but only essential changes are made.
- Users have to work around problems that they discover.
- Eventually, the software is retired and taken out of use.
- This often incurs further costs as data is transferred from an old system to a newer replacement system.

Evolution and servicing

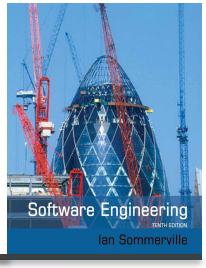


- Evolution
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- Servicing
  - At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.
- Phase-out
  - The software may still be used but no further changes are made to it.



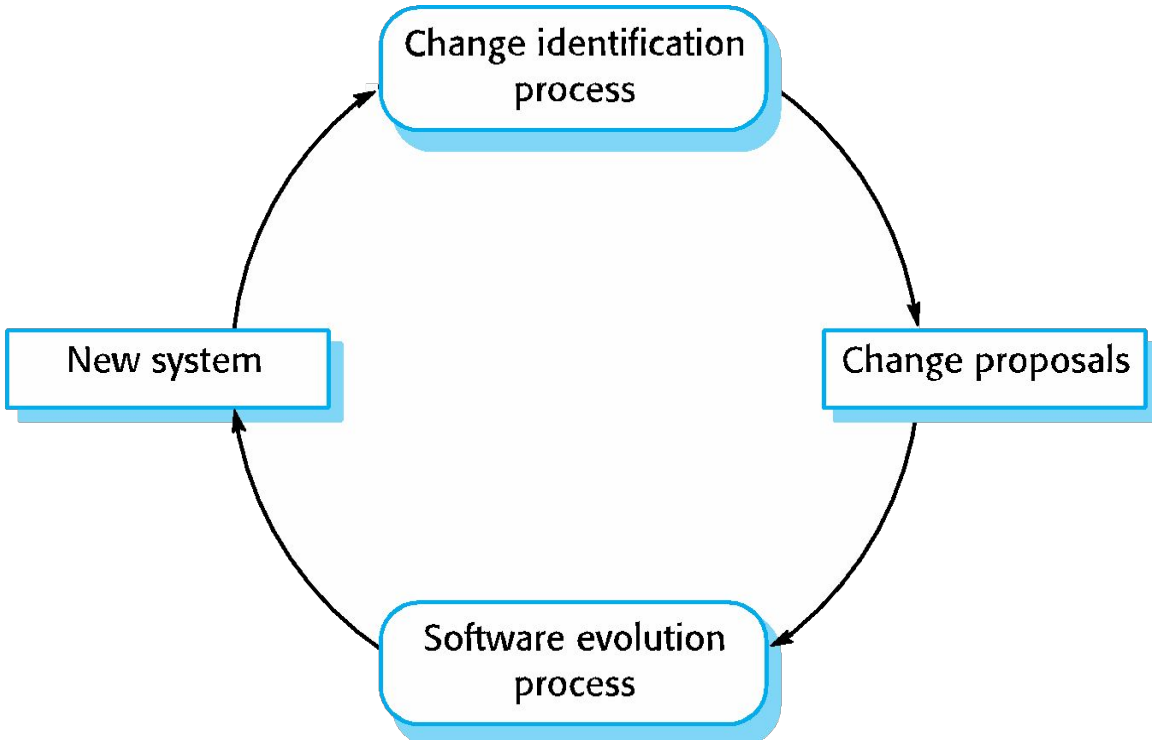
Evolution processes

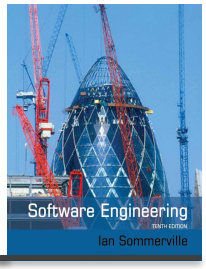
Evolution processes



- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience 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.
- **Change identification and evolution continues throughout the system lifetime.**

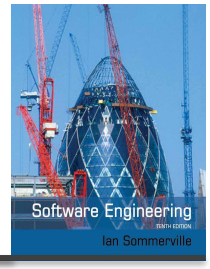
Change identification and evolution processes

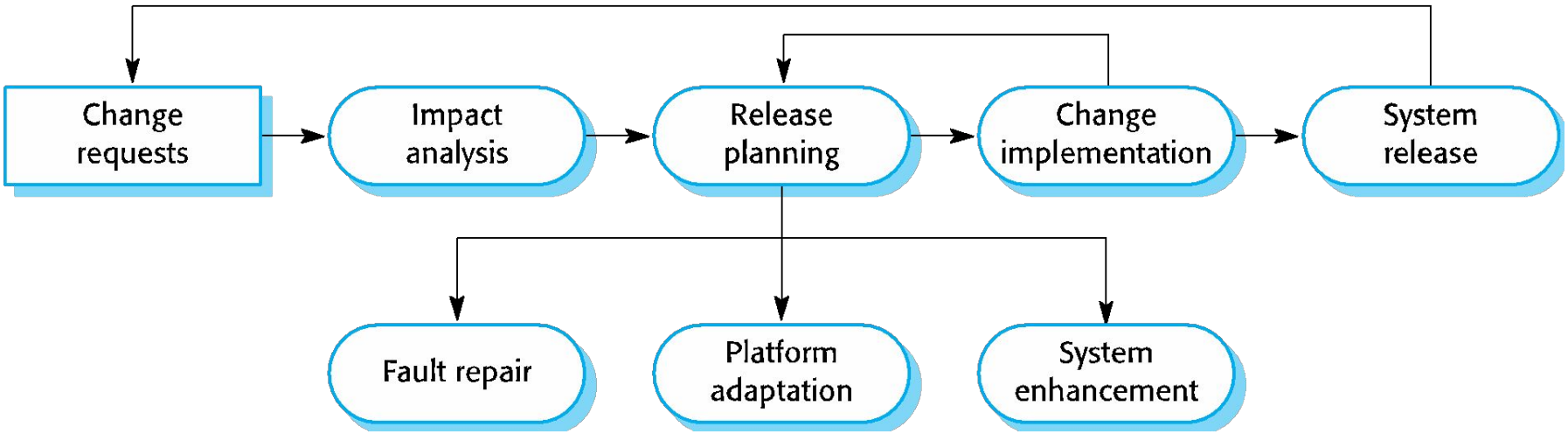


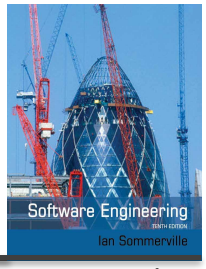


- Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed.
- This analysis allows the cost and the impact of the change to be assessed.
- This is part of the general process of change management, which should also ensure that the correct versions of components are included in each system release.
- 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.

A general Model for the software evolution process

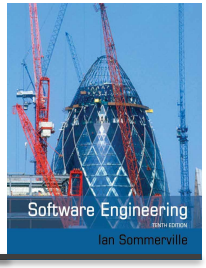




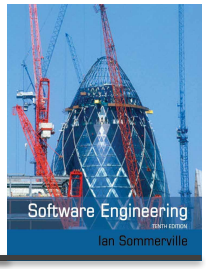


- If the proposed changes are accepted, a new release of the system is planned.
- During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered.
- A decision is then made on which changes to implement in the next version of the system.
- The changes are 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.
- **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.**

Change implementation

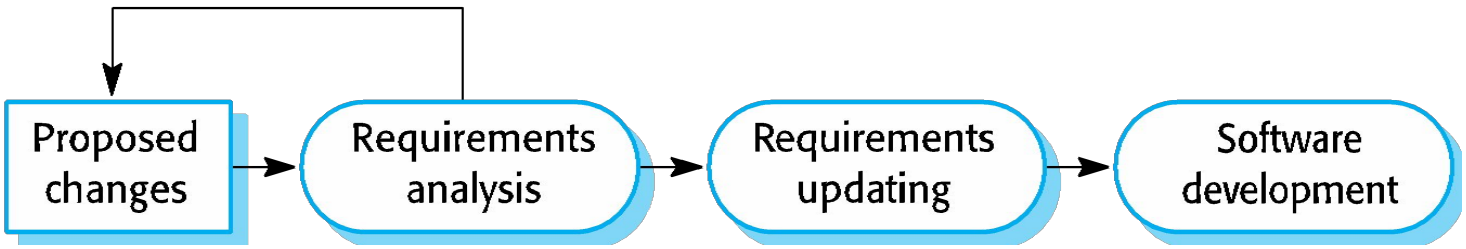


- Where **different teams are involved**, a critical difference between development and evolution is that the first stage of change implementation **requires program understanding**.
- During the program understanding phase, **new developers have to understand how the program is structured, how it delivers 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.

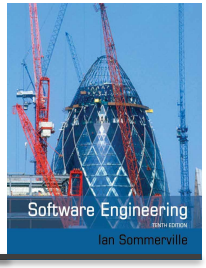


- If **requirements specification and design documents** are available, these **should be updated during the evolution process** to reflect the changes that are required (Figure 9.5).
- **New software requirements should be written**, and these should be analyzed and validated.
- If the design has been 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.

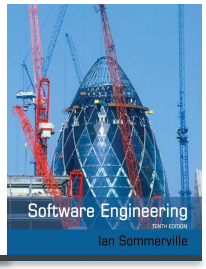
Change implementation (Fig 9.5)



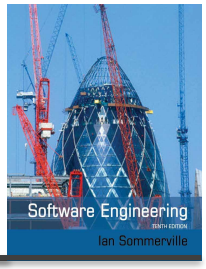
Urgent change requests



- Urgent changes may have to be implemented without going through all stages of the software engineering process.
- change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:
  - If a serious system fault has to be repaired to allow normal operation to continue;
  - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
  - If there are business changes that require a very rapid response (e.g. the release of a competing product).

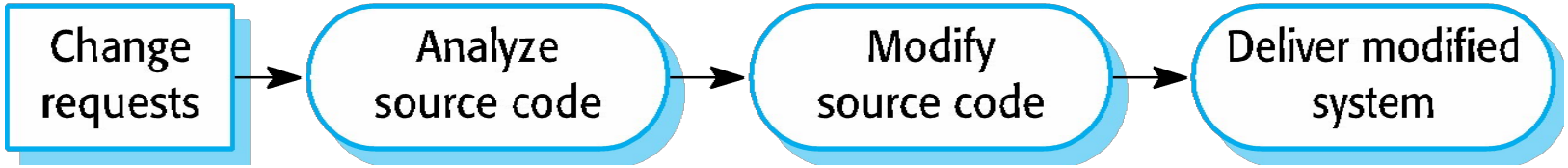


- In these cases, the need to make the change quickly means that you may **not be able to update all of the software documentation**.
- Rather than modify the requirements and design, you **make an emergency fix to the program** to solve the immediate problem (Figure 9.6).
- The danger here is that the requirements, the software design, and **the code can become inconsistent.**
- While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation.
- Eventually, the **original change is forgotten**, and the system **documentation and code are never realigned**.
- This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

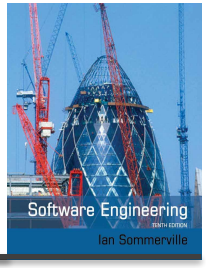


- Emergency system repairs have to be completed as quickly as possible.
- You **choose a quick and workable solution rather than the best solution** as far as system structure is concerned.
- This tends to accelerate the process of software ageing so that **future changes become progressively more difficult and maintenance costs increase.**
- Ideally, **after emergency code repairs are made, the new code should be refactored and improved to avoid program degradation**. Of course, the code of the repair may be reused if possible.
- However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

The emergency repair process (Fig. 9.6)

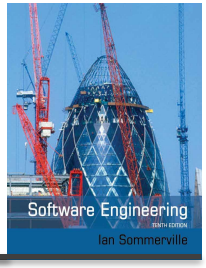


Agile methods and evolution

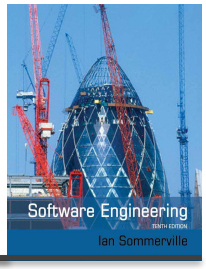


- Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional user stories.

Handover problems

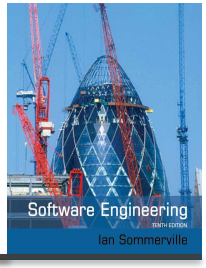


- Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
  - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.
- Where a plan-based approach has been used for development but the evolution team prefer to use agile methods.
  - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.



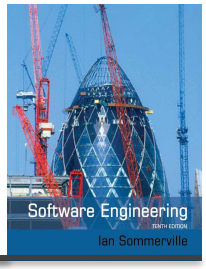
Legacy systems

Legacy systems

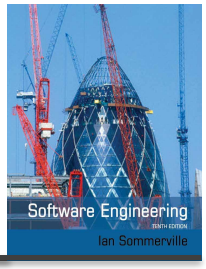


- **A legacy system is outdated computing software and/or hardware that is still in use.**

- Risks of replacing a legacy system:
  - Specification is difficult because existing documentation is typically incomplete.
  - Changing business processes (now adjusted to the system) may entail high costs.
  - Undocumented, yet important business rules may be embedded in the system; a new system may break these rules.
  - The new system may be delivered late, may cost more than expected, and may not function properly.

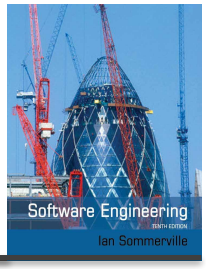


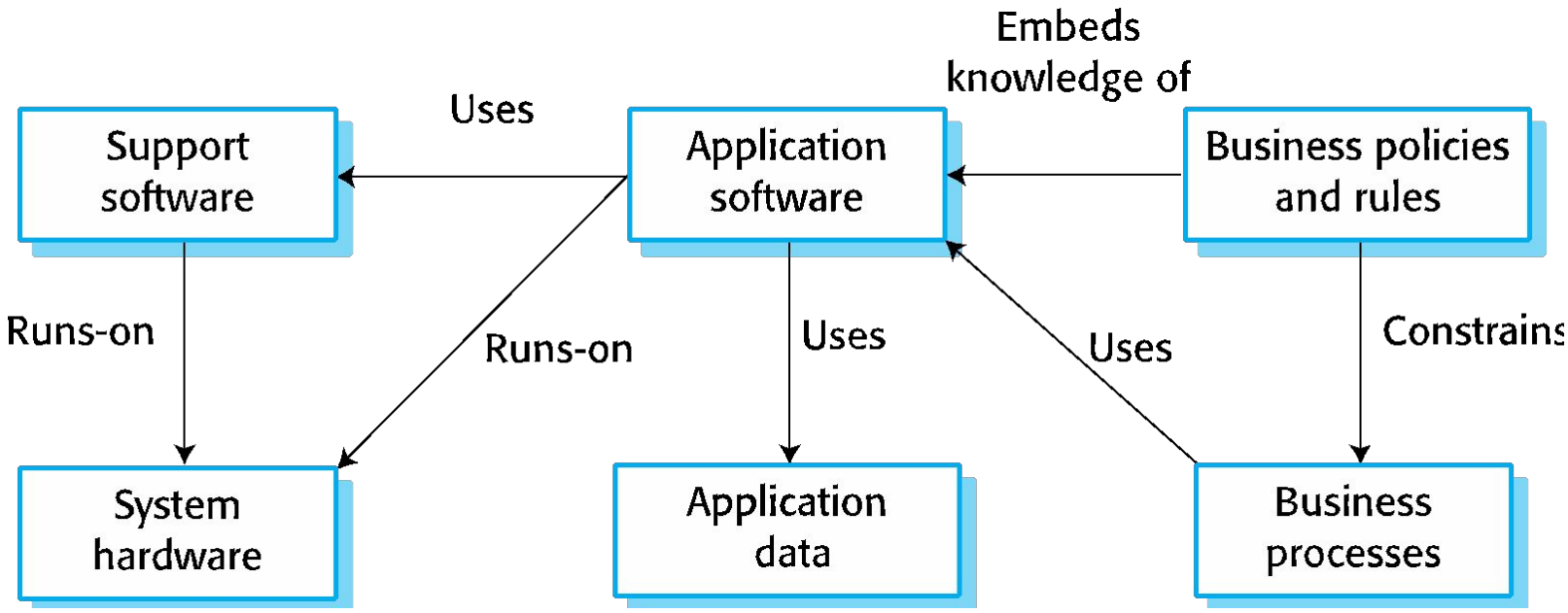
- Factors that make changes to legacy systems expensive:
  - In large systems, different parts were implemented by different teams, without consistent programming style
  - It is difficult to find personnel who knows the obsolete programming languages used in old systems
  - In may cases the only documentation is provided by the source code; even this may be missing
  - It is difficult to understand the system given its *ad hoc* updating over the years
  - Data used by the system is difficult to understand and manipulate; it can also be obsolete and/or redundant



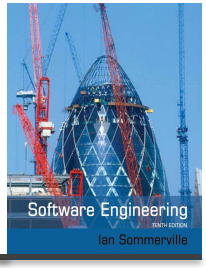
- The reasons are varied as to why a company would continue to use a legacy system.
- **Investment**: Although maintaining a legacy system is expensive over time, upgrading to a new system requires an up-front investment, both in dollars and manpower.
- **Fear:** Change is hard, and moving a whole company —or even a single department — to a new system can inspire some internal resistance.
- **Difficulty**: The legacy software may be built with an obsolete programming language that makes it hard to find personnel with the skills to make the migration. There may be little documentation about the system and the original developers have left the company. Sometimes simply planning the migration of data from a legacy system and defining the scope of requirements for a new system are overwhelming.

The elements of a legacy system



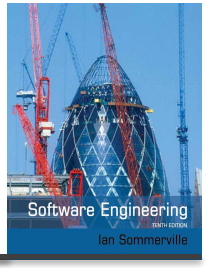


Legacy system components



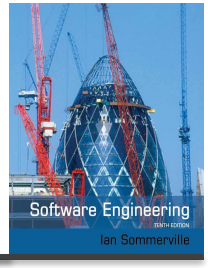
- *System hardware:* Legacy systems may have been written for hardware that is no longer available.
- *Support software:* The legacy system may rely on a range of support software, which may be obsolete or unsupported.
- *Application software (legacy s/w system):* The application system that provides the business services is usually made up of a number of application programs.
- *Application data:* These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.

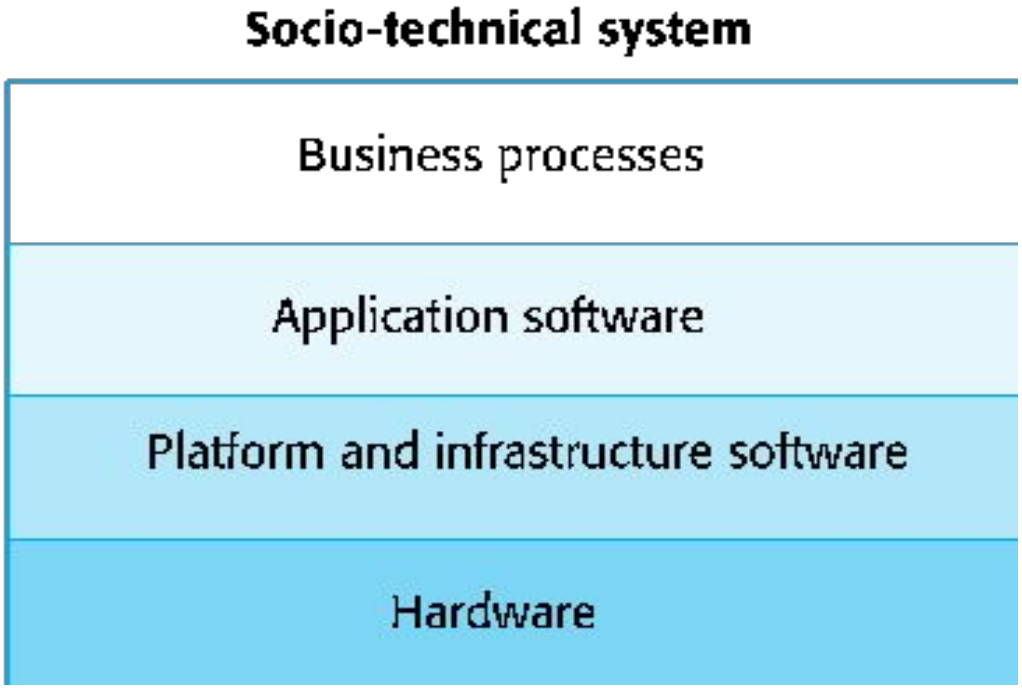
Legacy system components

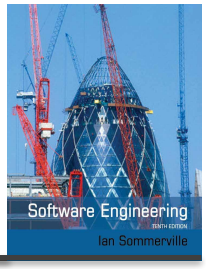


- *Business processes:* These are processes that are used in the business to achieve some business objective.
- Business processes may be designed around a legacy system and constrained by the functionality that it provides.
- *Business policies and rules:* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

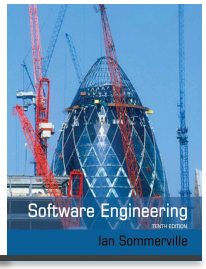
Legacy system layers



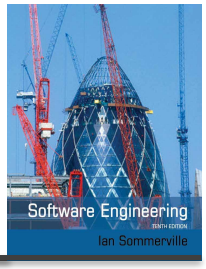




- An alternative way of looking at these components of a legacy system is as a series of layers (shown in the above fig).
- Each layer depends on the layer immediately below it and interfaces with that layer.
- If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers.
- In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

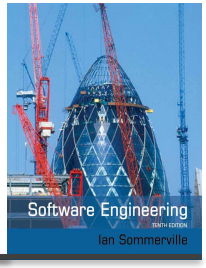


- 1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
- 2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.

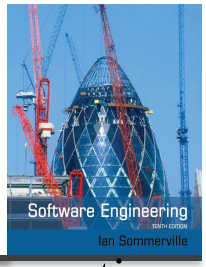


- 3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced.
- This is a particular problem in embedded systems where there is a tight coupling between software and hardware.
- Major changes to the application software may be required to make effective use of the new hardware.

Legacy system management

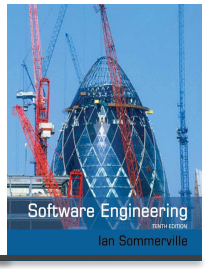


- Organisations that rely on legacy systems must choose a strategy for evolving these systems:
  - **Scrap the system completely and modify business processes** -This option should be chosen when the system is not making an effective contribution to business processes.
  - **Continue maintaining the system** - This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.

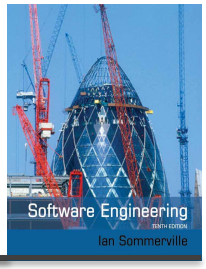


- **Reengineer the system to improve its maintainability**- This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.
- **Replace the system with a new system** - This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost

✧ The strategy chosen should depend on the system quality and its business value.

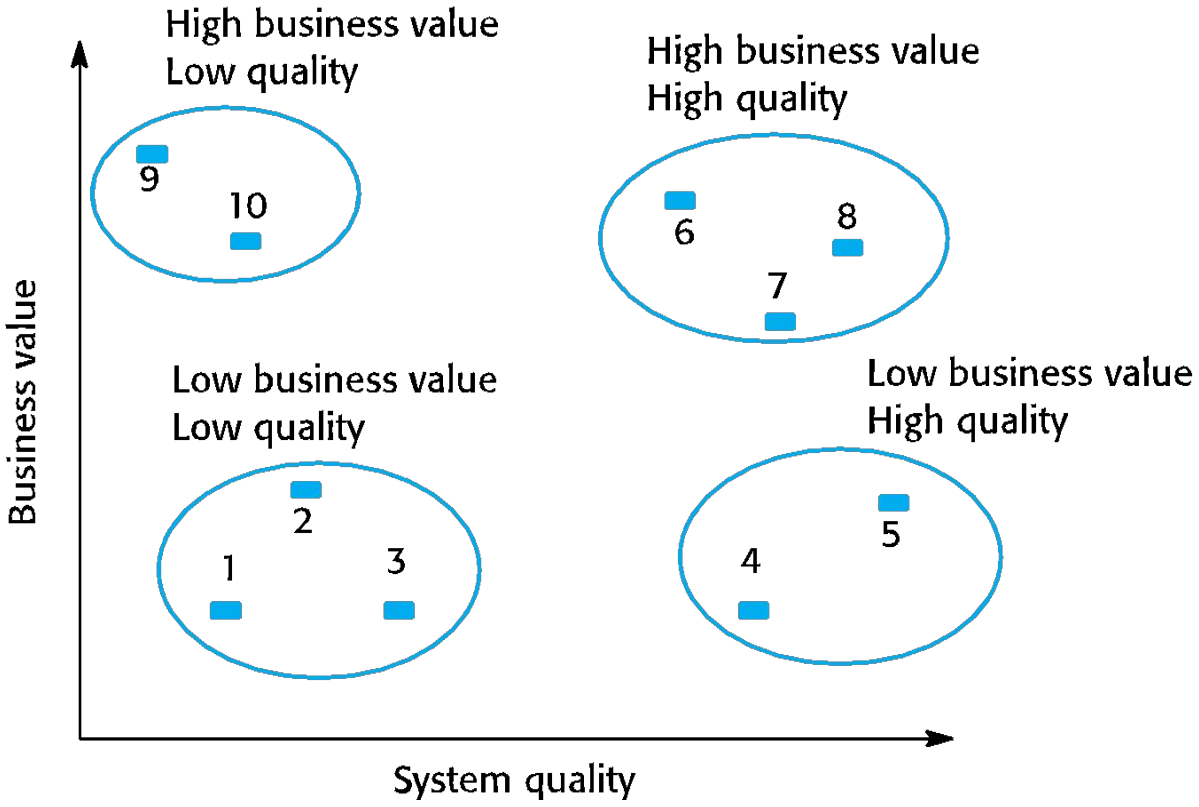


- When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective.
- From a business perspective, you have to decide whether or not the business really needs the system.
- From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware.
- You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

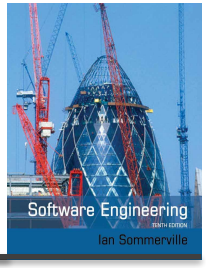


- For example, assume that an organization has 10 legacy systems.
- You should assess the quality and the business value of each of these systems.
- You may then create a chart showing relative business value and system quality.
- An example of this is shown in Figure 9.9.

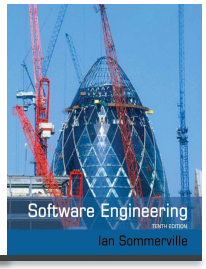
Figure 9.9 An example of a legacy system assessment



Legacy system categories

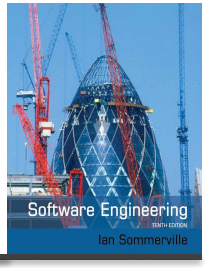


- **Low quality, low business value**
  - Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
- **Low-quality, high-business value**
  - These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.



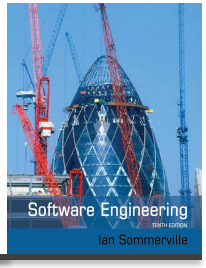
- **High-quality, low-business value**
  - These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
- **High-quality, high business value**
  - These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

Business value assessment

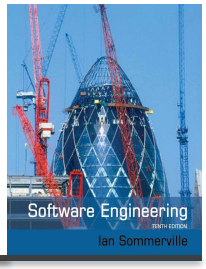


- The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems.
- Assessment should take different viewpoints into account
  - System end-users;
  - Business customers;
  - Line managers;
  - IT managers;
  - Senior managers.
- Interview different stakeholders and collate results.

Issues in business value assessment

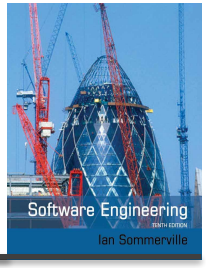


- The use of the system
  - If systems are only used occasionally or by a small number of people, they may have a low business value.
- The business processes that are supported
  - When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. A system may have a low business value if it forces the use of inefficient business processes.



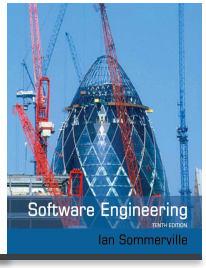
- System dependability (trustworthy or reliable)
  - If a system is not dependable and the problems directly affect business customers, the system has a low business value.
- The system outputs
  - If the business depends on system outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

System quality assessment



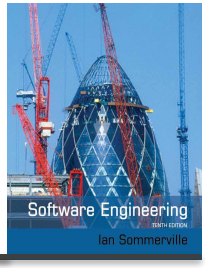
- Business process assessment
  - How well does the business process support the current goals of the business?
- Environment assessment
  - How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - What is the quality of the application software system?

Business process assessment



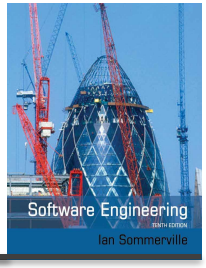
- Use a viewpoint-oriented approach and seek answers from system stakeholders:
  - Is there a defined process model and is it followed?
  - Do different parts of the organisation use different processes for the same function?
  - How has the process been adapted?
  - What are the relationships with other business processes and are these necessary?
  - Is the process effectively supported by the legacy application software?
- Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Factors used in environment assessment



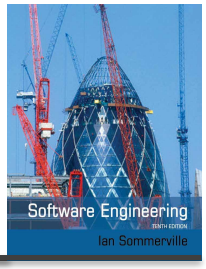
Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Factors used in environment assessment



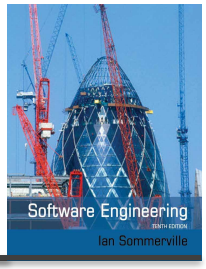
Factor	Questions
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences ? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Factors used in application assessment



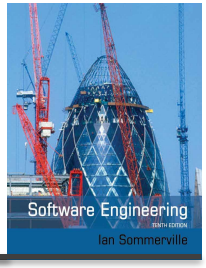
Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?

Factors used in application assessment

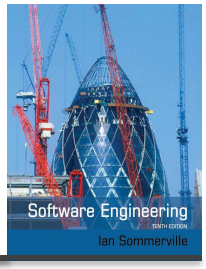


Factor	Questions
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

System measurement

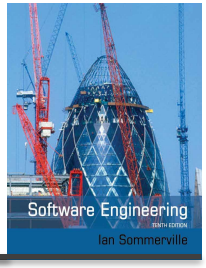


- You may collect quantitative data to make an assessment of the quality of the application system
  - The number of system change requests; The higher this accumulated value, the lower the quality of the system.
  - The number of different user interfaces used by the system; The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.
  - The volume of data used by the system. As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data.
  - Cleaning up old data is a very expensive and time-consuming process



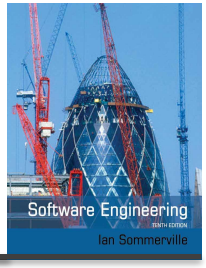
Software maintenance

Software maintenance



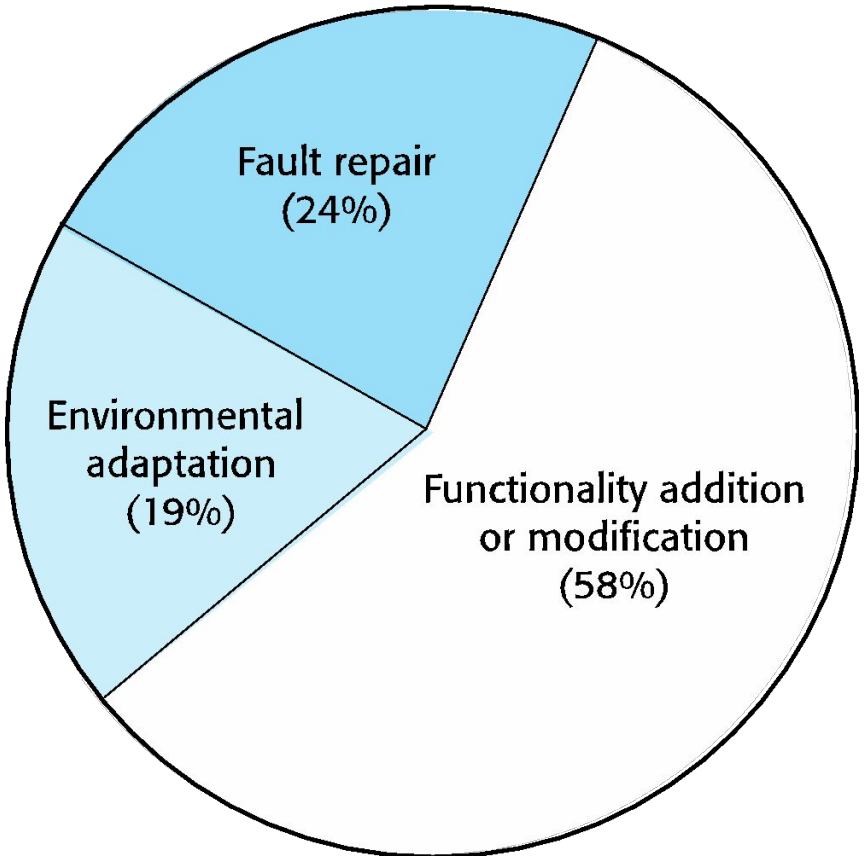
- Software maintenance is the general process of changing a system after it has been delivered.
- The term is usually applied to custom software, where separate development groups are involved before and after delivery.
- The changes made to the software may be simple changes to correct coding errors,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

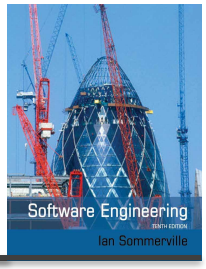
Types of maintenance



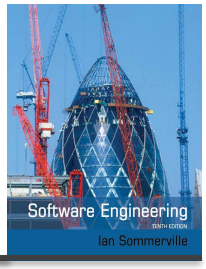
- Fault repairs
  - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.
- Environmental adaptation
  - Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Functionality addition and modification
  - Modifying the system to satisfy new requirements.

Maintenance effort distribution (fig.9.12)



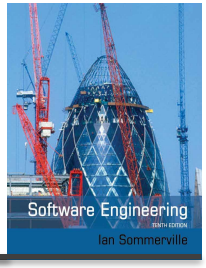


- Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010).
- This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005.
- The authors found that the distribution of maintenance costs had changed very little over 30 years.
- Although we don't have more recent data, this suggests that this distribution is still largely correct.
- Repairing system faults is not the most expensive maintenance activity.
- Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.



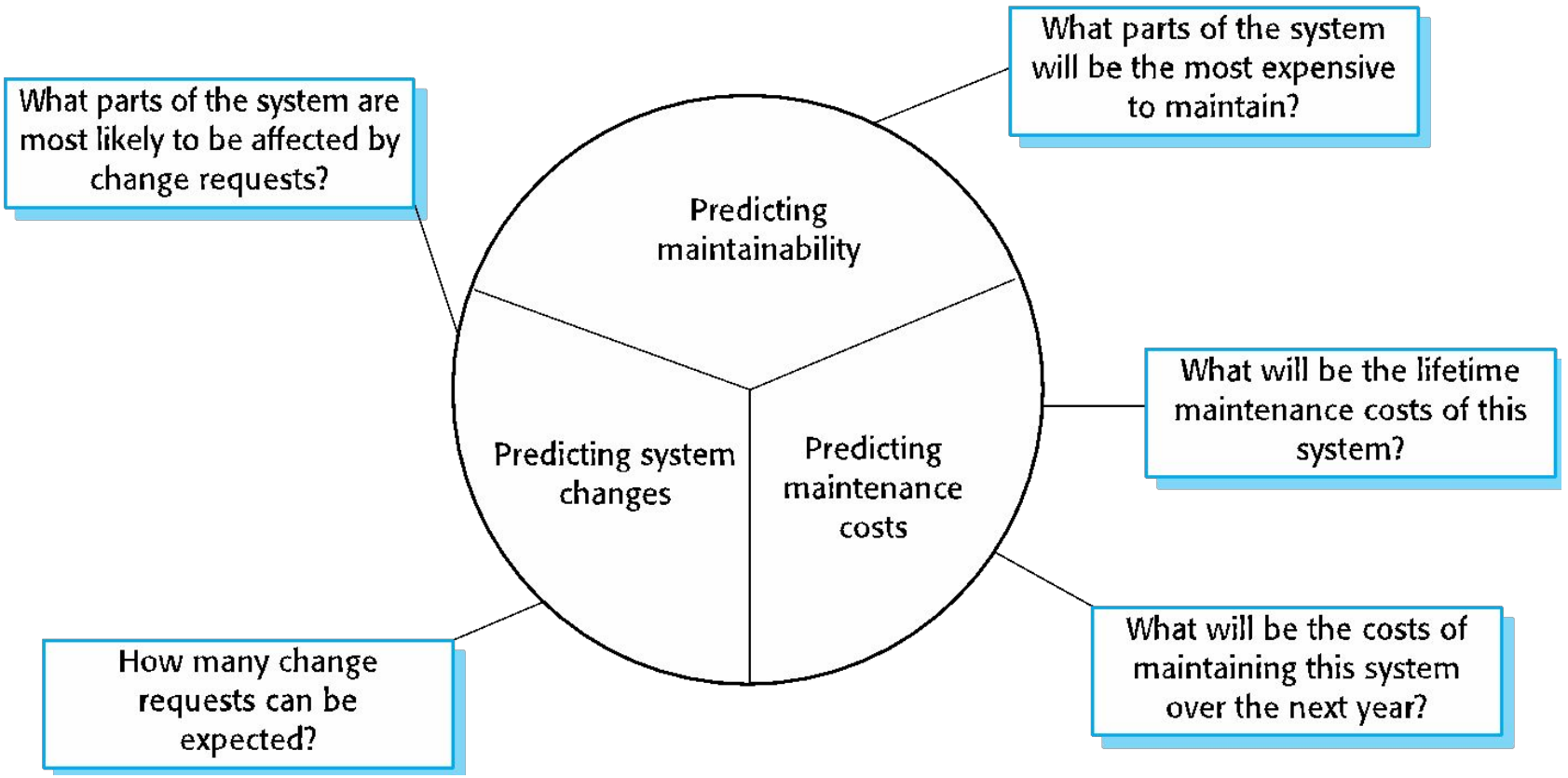
- It is usually more expensive to add new features to a system during maintenance than it is to add the same features during development
  - A new team has to understand the programs being maintained
  - Separating maintenance and development means there is no incentive for the development team to write maintainable software
  - Program maintenance work is unpopular
    - Maintenance staff are often inexperienced and have limited domain knowledge.
  - As programs age, their structure degrades and they become harder to change

Maintenance prediction

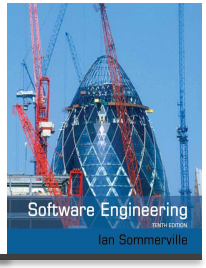


- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
  - Change acceptance depends on the maintainability of the components affected by the change;
  - Implementing changes degrades the system and reduces its maintainability;
  - Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Maintenance prediction

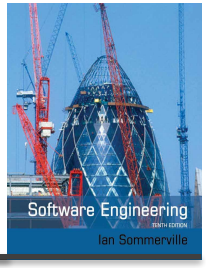


Change prediction



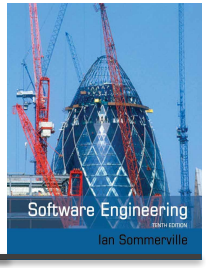
- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - Number and complexity of system interfaces;
  - Number of inherently volatile system requirements;
  - The business processes where the system is used.

Complexity metrics



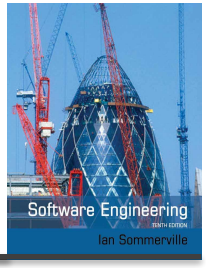
- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size.

Process metrics



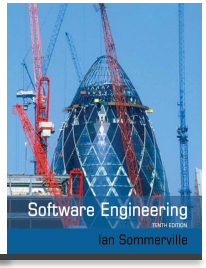
- Process metrics may be used to assess maintainability
  - Number of requests for corrective maintenance;
  - Average time required for impact analysis;
  - Average time taken to implement a change request;
  - Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

Software reengineering



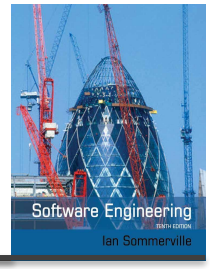
- Restructuring or rewriting part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Reengineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

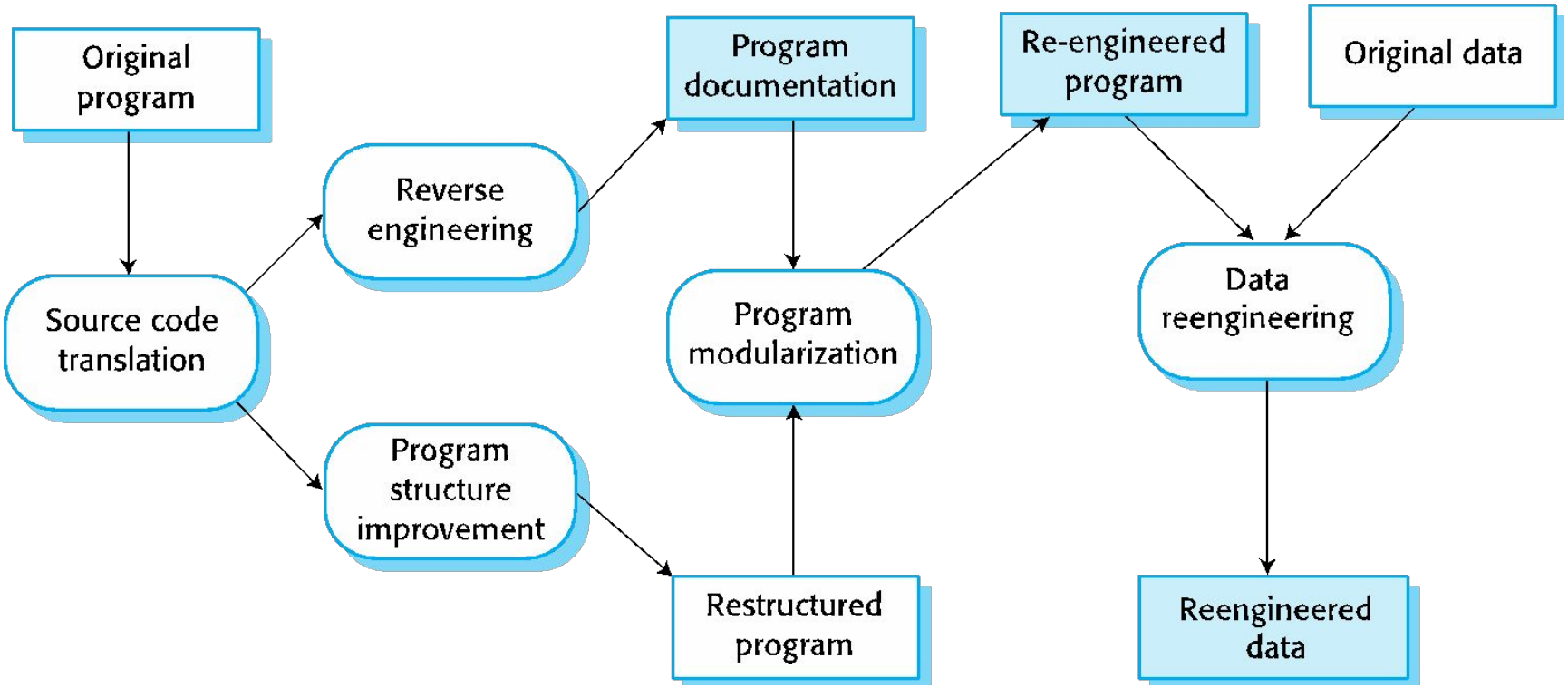
Advantages of reengineering



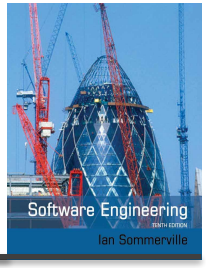
- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process



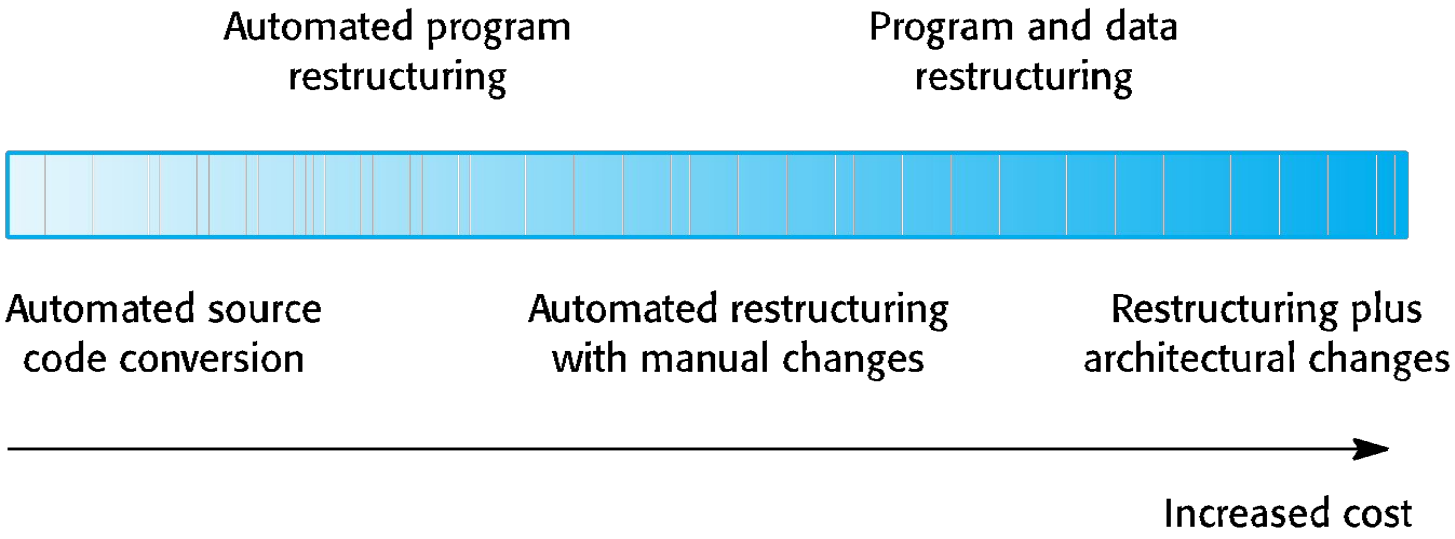


Reengineering process activities

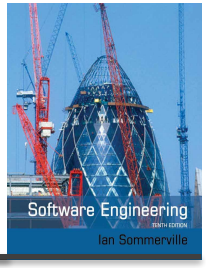


- Source code translation
  - Convert code to a new language.
- Reverse engineering
  - Analyse the program to understand it;
- Program structure improvement
  - Restructure automatically for understandability;
- Program modularisation
  - Reorganise the program structure;
- Data reengineering
  - Clean-up and restructure system data.

Reengineering approaches

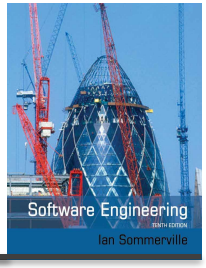


Reengineering cost factors



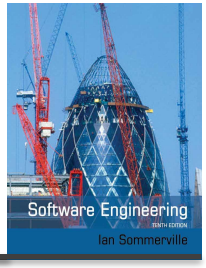
- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
  - This can be a problem with old systems based on technology that is no longer widely used.

Refactoring



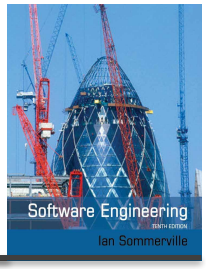
- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering



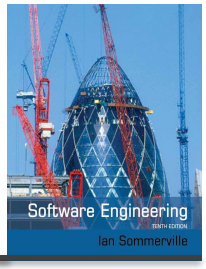
- Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

‘Bad smells’ in program code



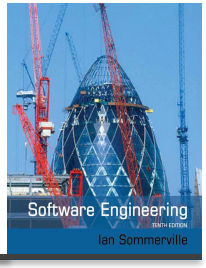
- Duplicate code
  - The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
- Long methods
  - If a method is too long, it should be redesigned as a number of shorter methods.
- Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code



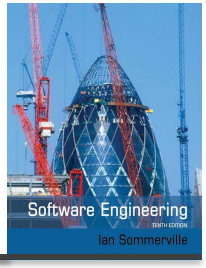
- Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Key points



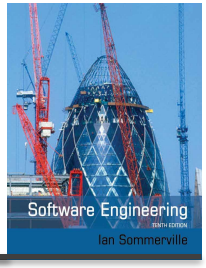
- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.

Key points



- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.
- There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.

Key points



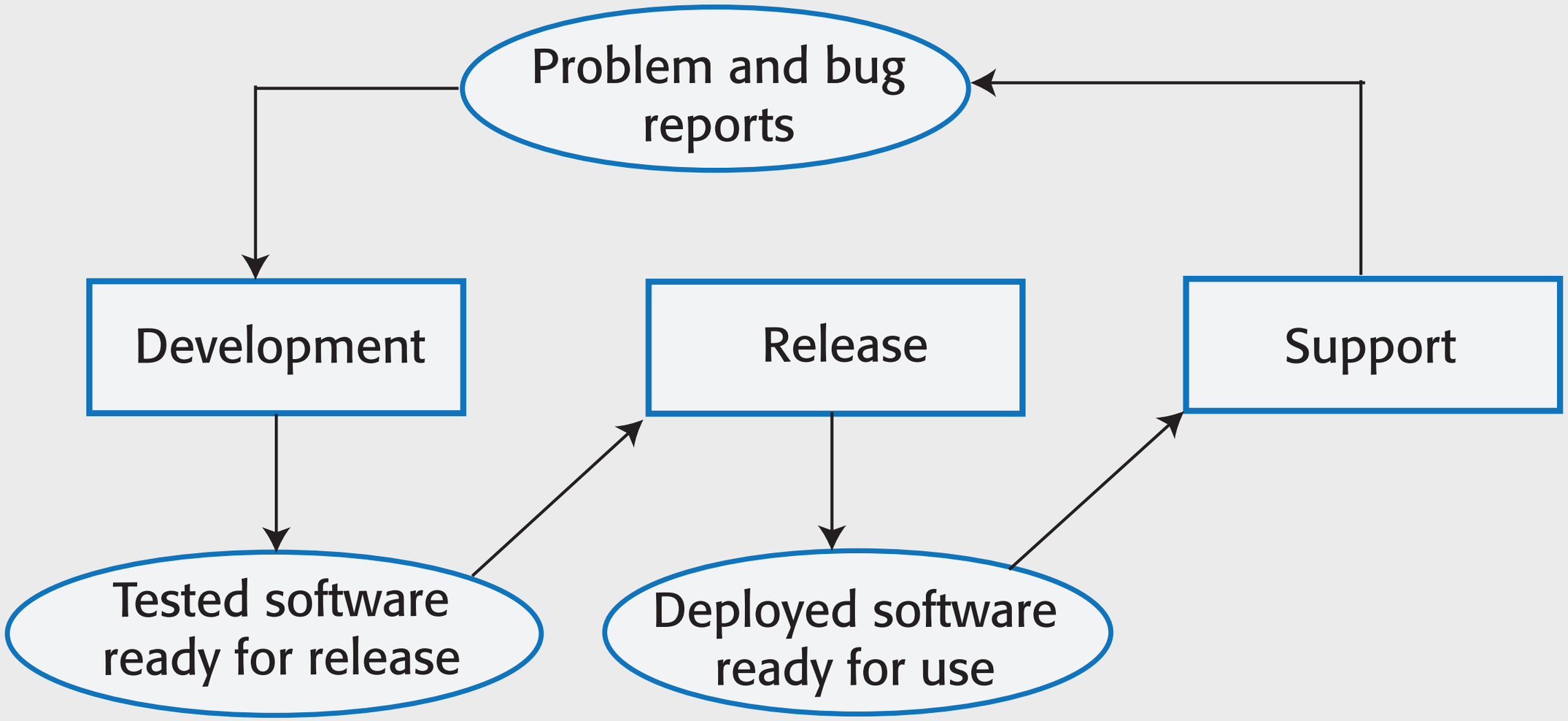
- Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.

DevOps and Code Management

Software support

- Traditionally, separate teams were responsible software development, software release and software support.
- The development team passed over a ‘final’ version of the software to a release team. This team then built a release version, tested this and prepared release documentation before releasing the software to customers.
- A third team was responsible for providing customer support.
  - The original development team were sometimes also responsible for implementing software changes.
  - Alternatively, the software may have been maintained by a separate ‘maintenance team’.

Figure 10.1 Development, release and support



DevOps

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

Figure 10.2 DevOps

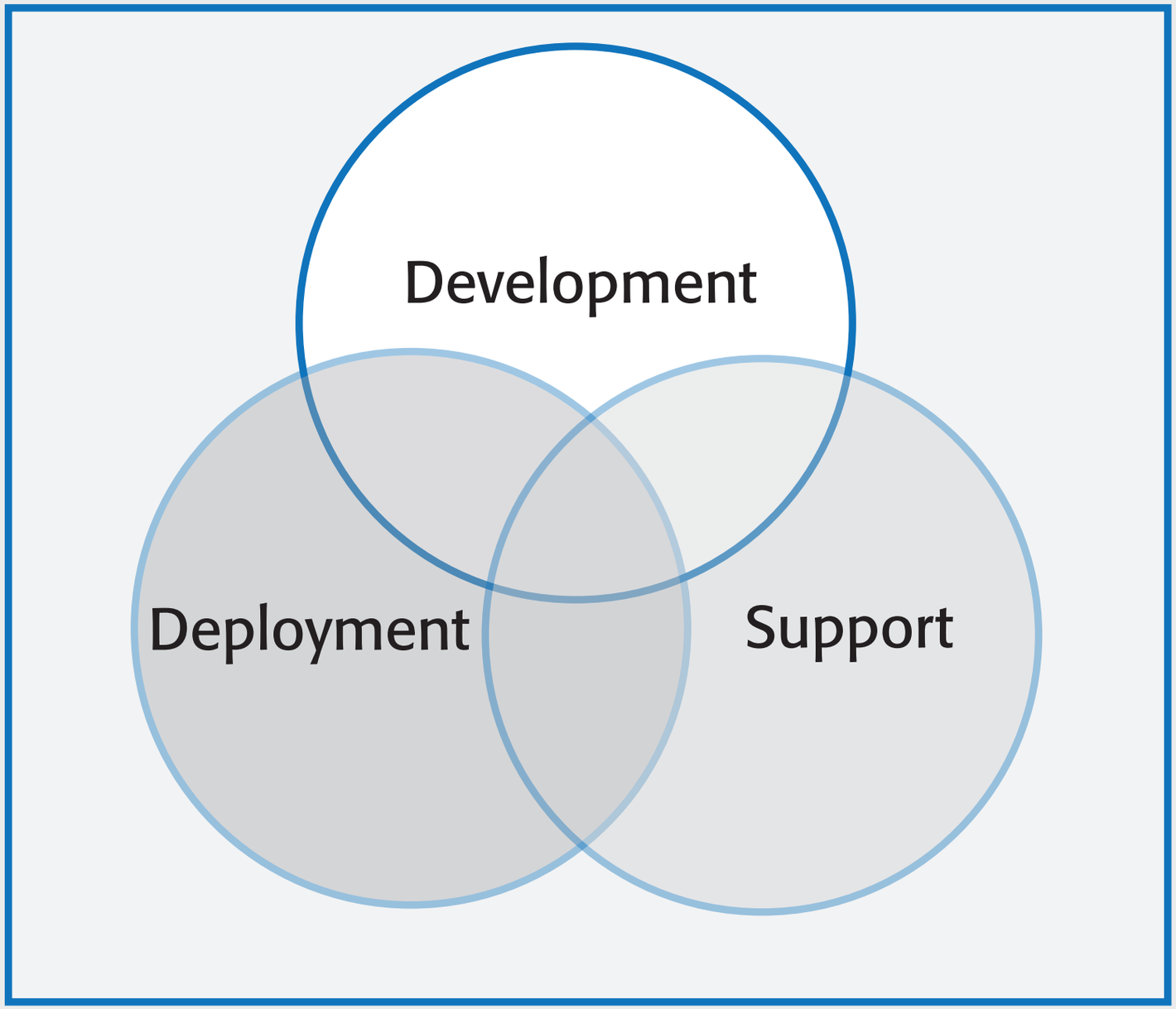


Table 10.1 DevOps principles

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.

Table 10.2 Benefits of DevOps

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 causing 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 were 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

- During the development of a software product, the development team will probably create tens of thousands of lines of code and automated tests.
- These will be organized into hundreds of files. Dozens of libraries may be used, and several, different programs may be involved in creating and running the code.
- Code management is a set of software-supported practices that is 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.

Table 10.3 A code management problem

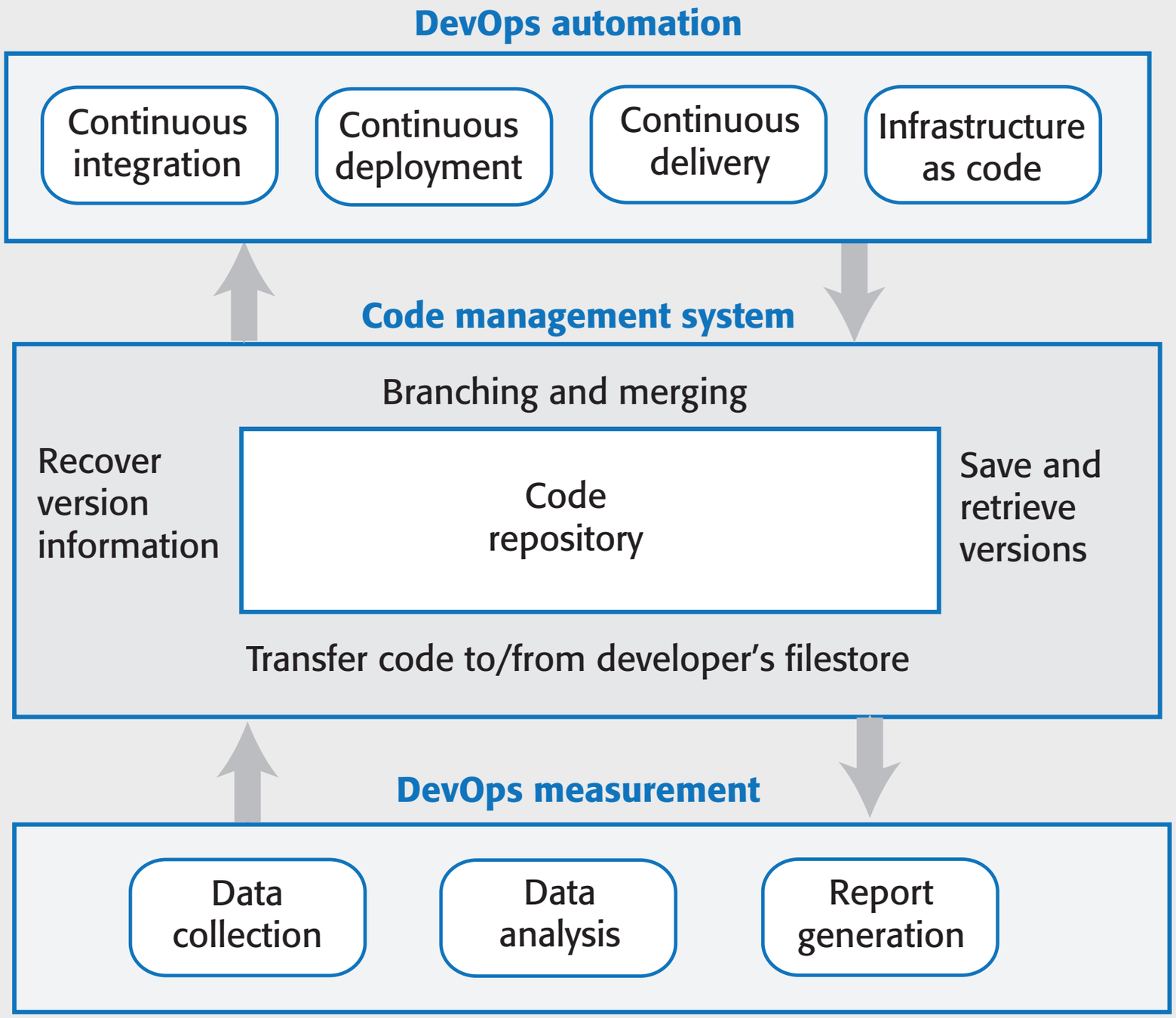
Alice and Bob worked for a company called FinanceMadeSimple and were team members involved in developing a personal finance product. Alice discovered a bug in a module called TaxReturnPreparation. The bug was that a tax return was reported as filed but, sometimes, it was not actually sent to the tax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

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

Code management and DevOps

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

Figure 10.3 Code management and Devops



Code management fundamentals

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

Code repository

- All source code management systems have the general form shown in Figure 10.3. with a shared repository and a set of features to manage the files in that repository:
  - All source code files and file versions are stored in the repository, as are other artefacts such as configuration files, build scripts, shared libraries and versions of tools used.
  - 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.
- Files can be transferred to and from the repository and information about the different versions of files and their relationships may be updated.
  - Specific versions of files and information about these versions can always be retrieved from the repository.

Table 10.4 Features of code management systems

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 why 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.

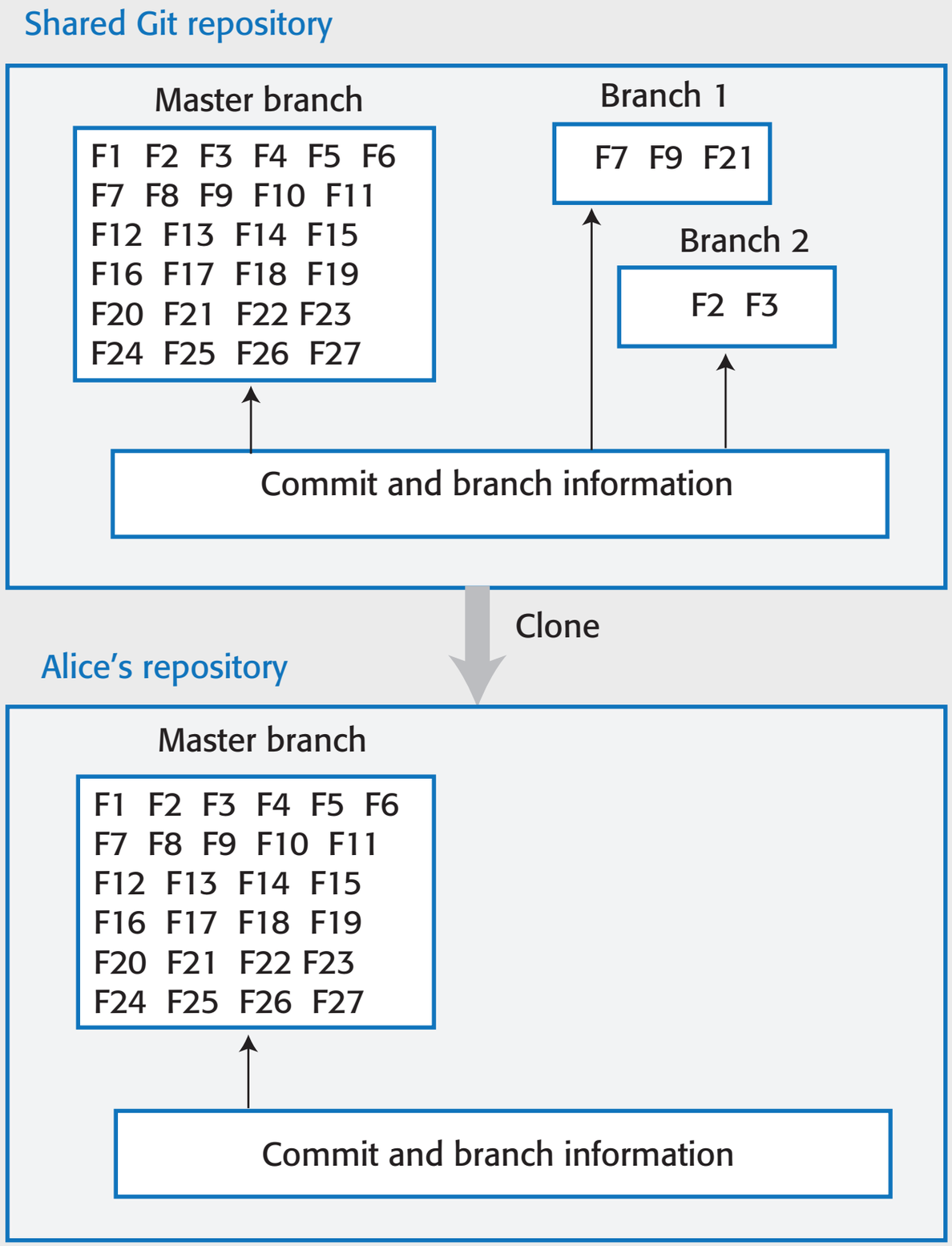
Project support
All of the files associated with a project may be checked out at the same time.There is no need to check out files one at a time.

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.

Git

- 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.
- This 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

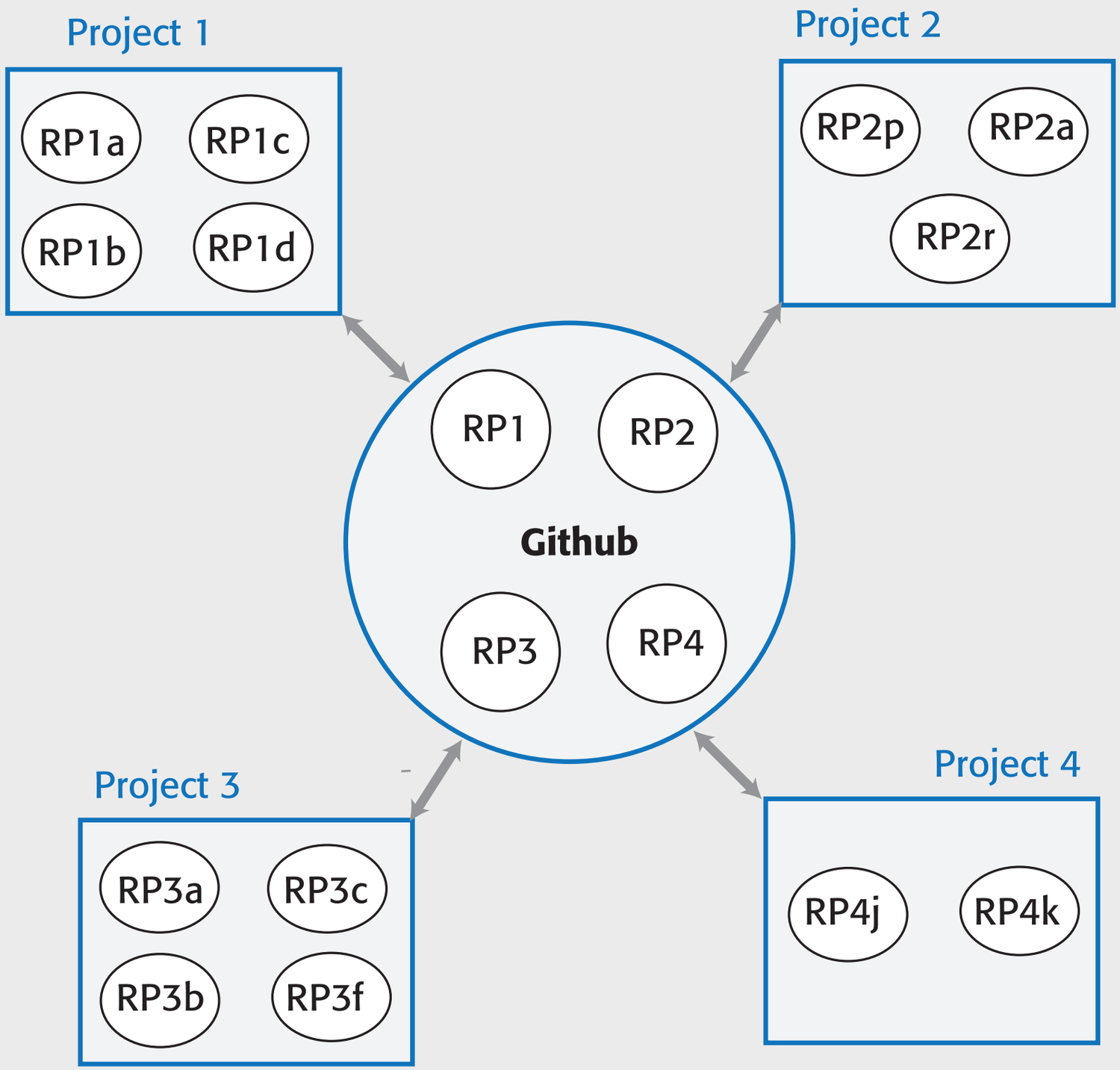
Figure 10.5 Repository cloning in Git



Benefits of distributed code management

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

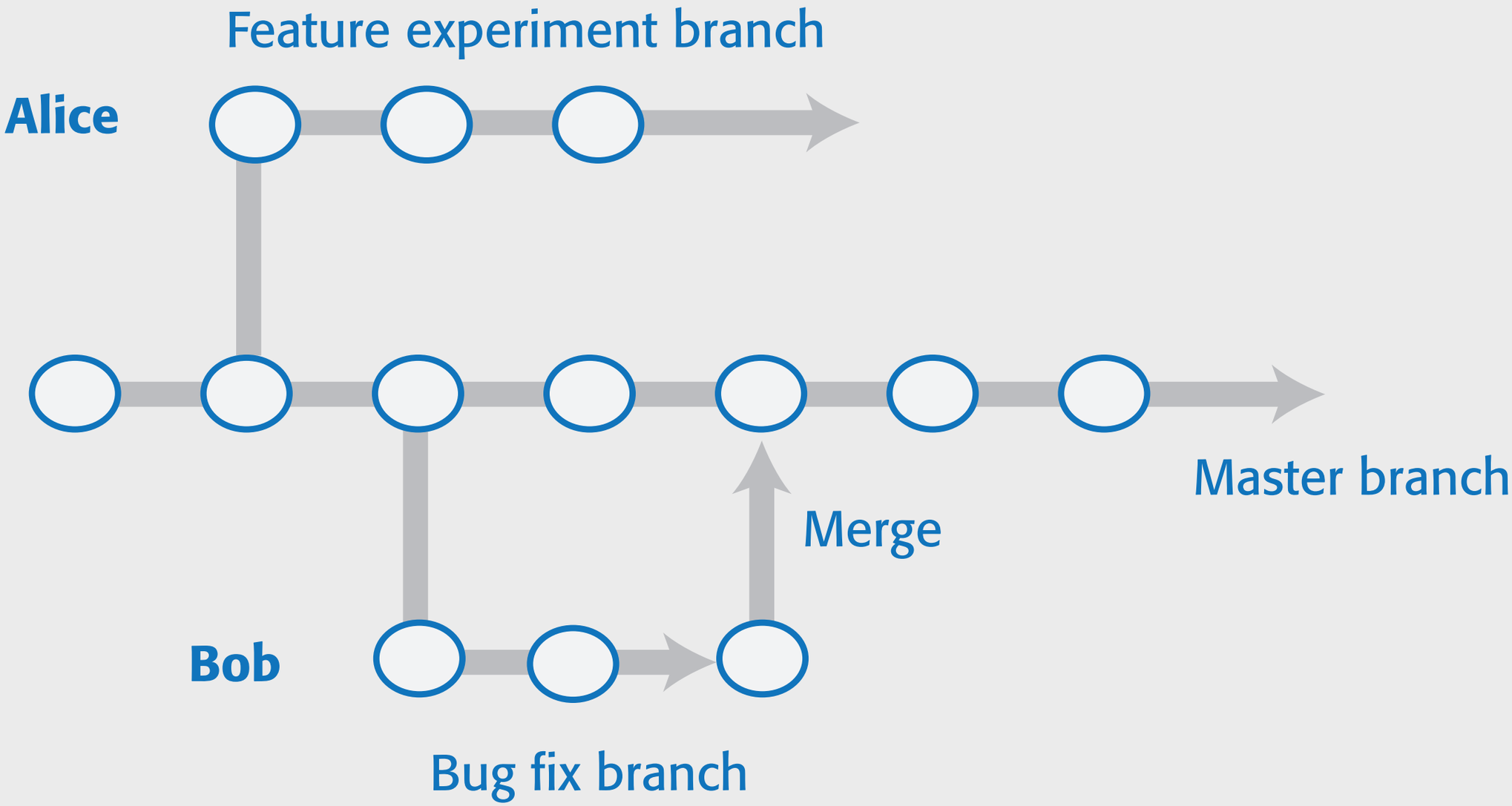
Figure 10.6 Git repositories



Branching and merging

- Branching and merging are fundamental ideas that are supported by all code management systems.
- A branch is an independent, stand-alone version that is created when a developer wishes to change a file.
- The changes made by developers in their own branches may be merged to create a new shared branch.
- The repository ensures that branch files that have been changed cannot overwrite repository files without a merge operation.
  - If Alice or Bob make mistakes on the branch they are working on, they can easily revert to the master file.
  - If they commit changes, while working, they can revert to earlier versions of the work they have done. When they have finished and tested their code, they can then replace the master file by merging the work they have done with the master branch

Figure 10.7 Branching and merging



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.
- As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.
- Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.

Figure 10.5 Aspects of DevOps automation

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.

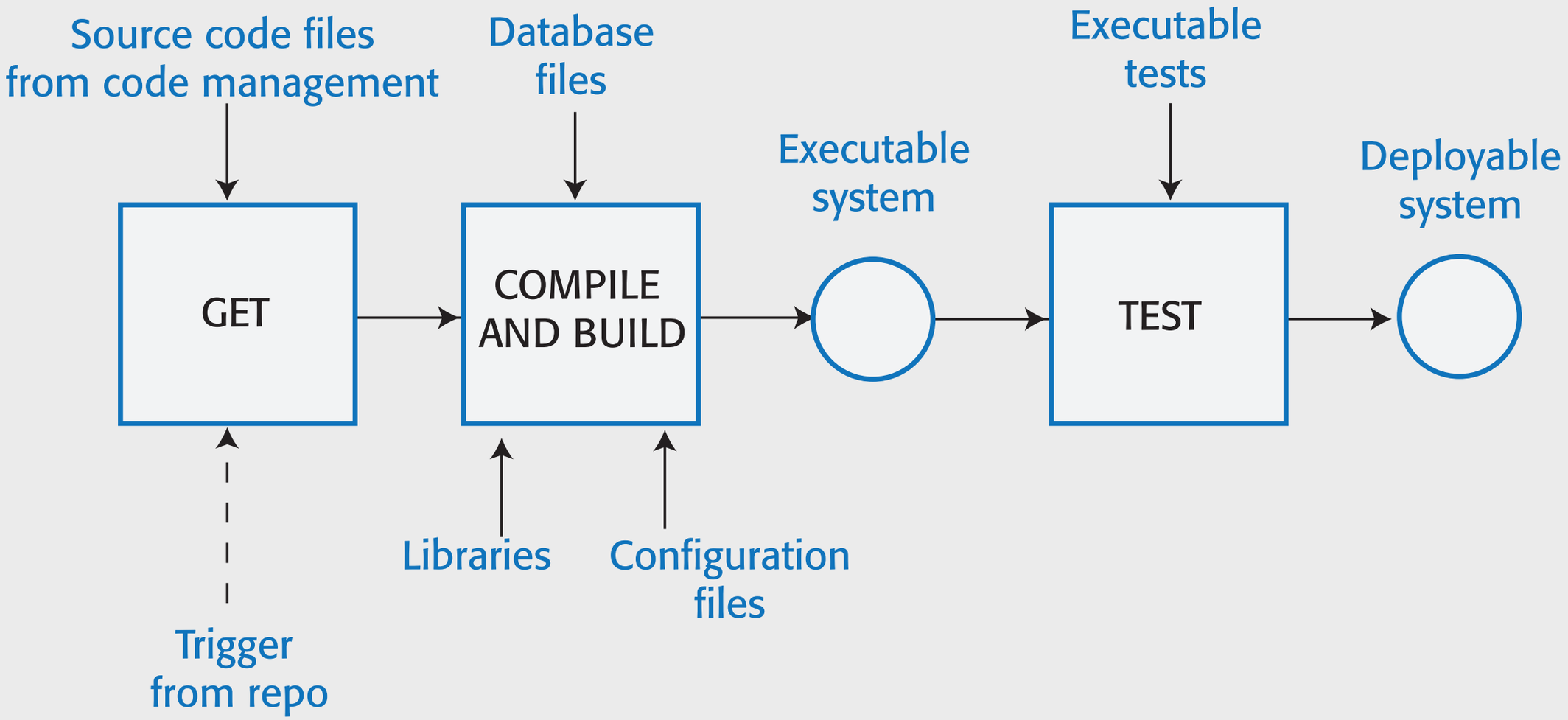
System integration

- System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system.
- Typical activities that are part of the system integration process include:
  - Installing database software and setting up the database with the appropriate schema.
  - Loading test data into the database.
  - Compiling the files that make up the product.
  - Linking the compiled code with the libraries and other components used.
  - Checking that external services used are operational.
  - Deleting old configuration files and moving configuration files to the correct locations.
  - Running a set of system tests to check that the integration has been successful.

Continuous integration

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

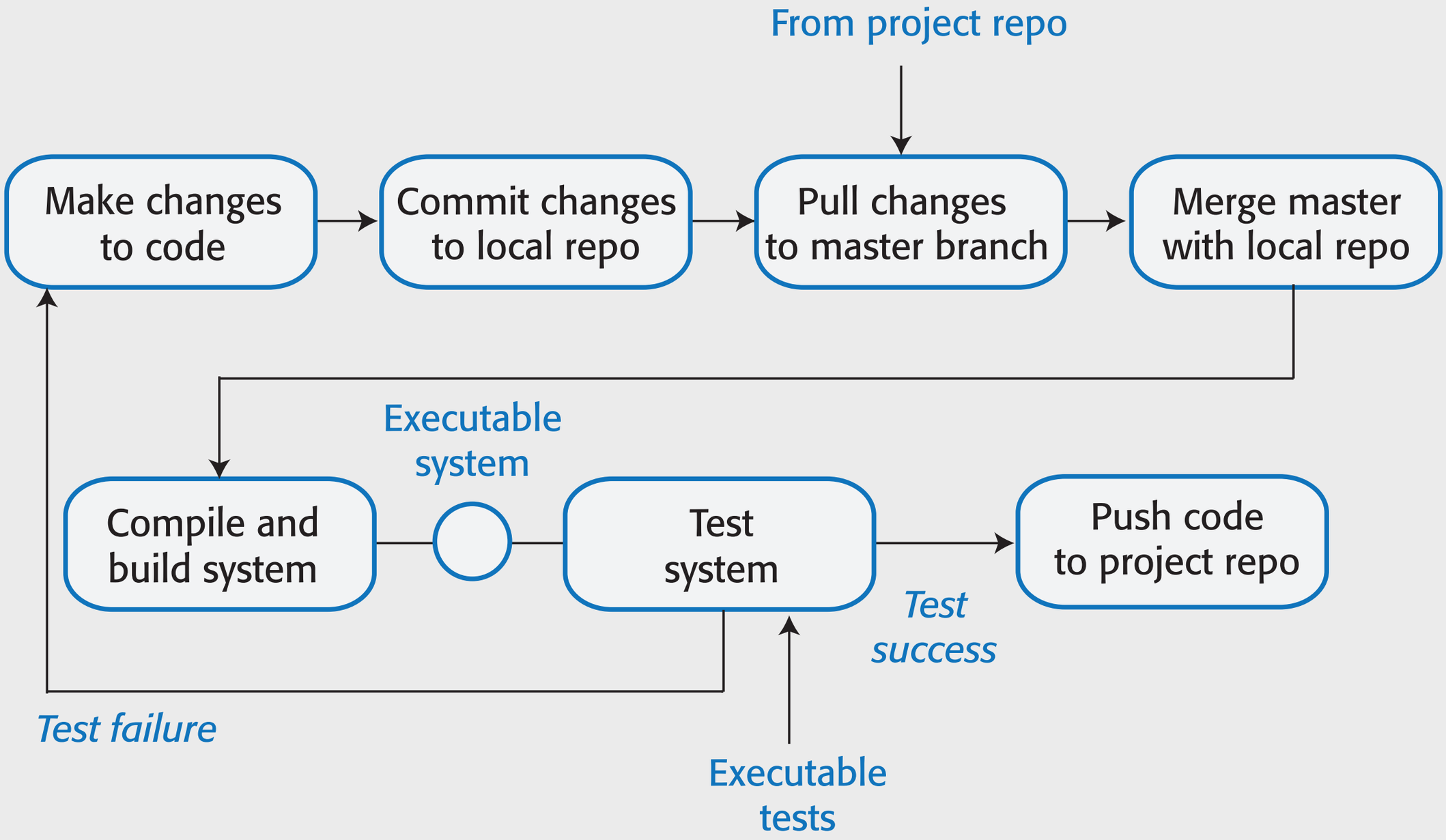
Figure 10.9 Continuous integration



Breaking the build

- 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.
- If this happens to you, your priority should be 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

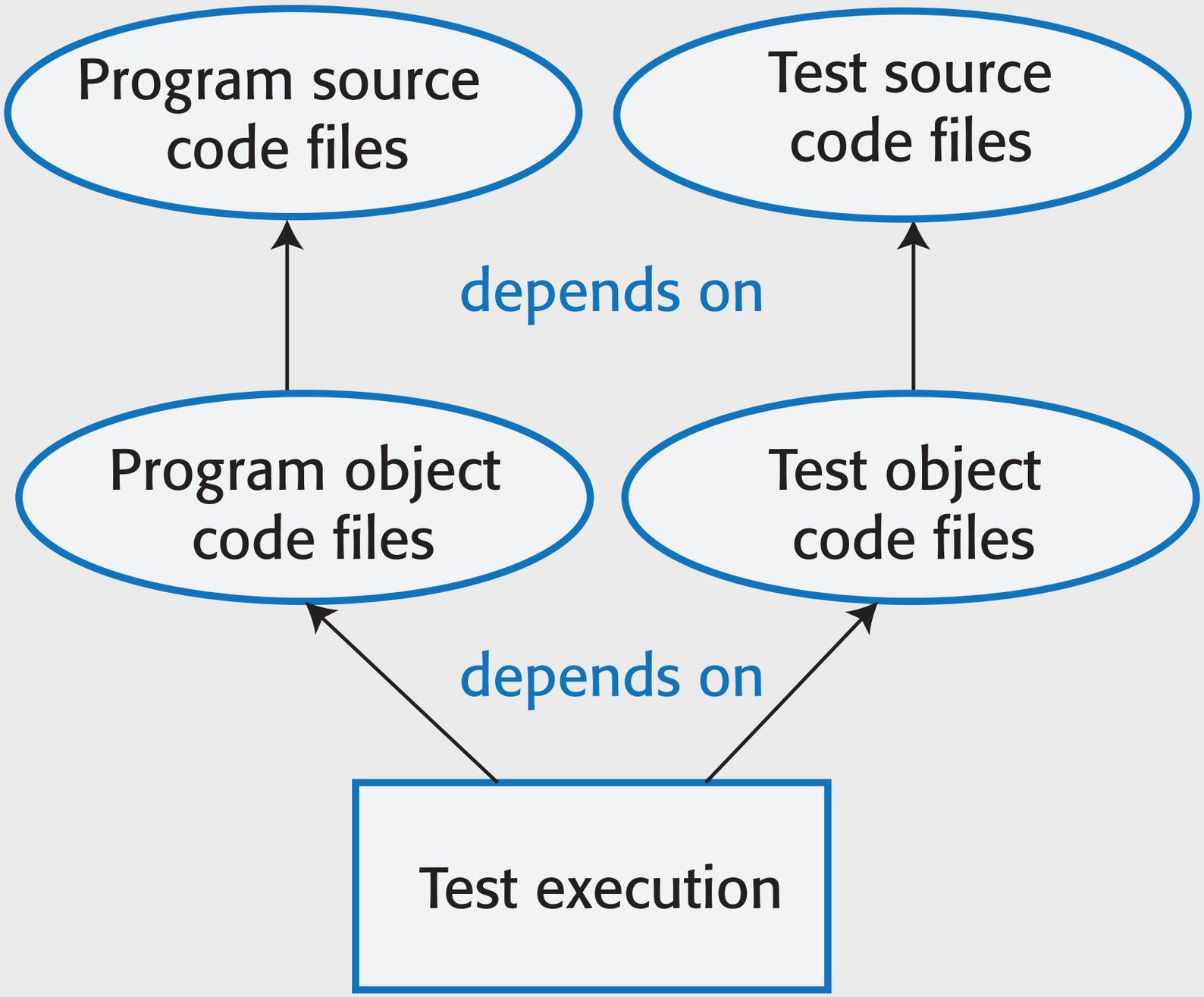
Figure 10.10 Local integration



System building

- Continuous integration is only effective if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow.
- It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- Fast system building is achieved using a process of incremental building, where only those parts of the system that have been changed are rebuilt

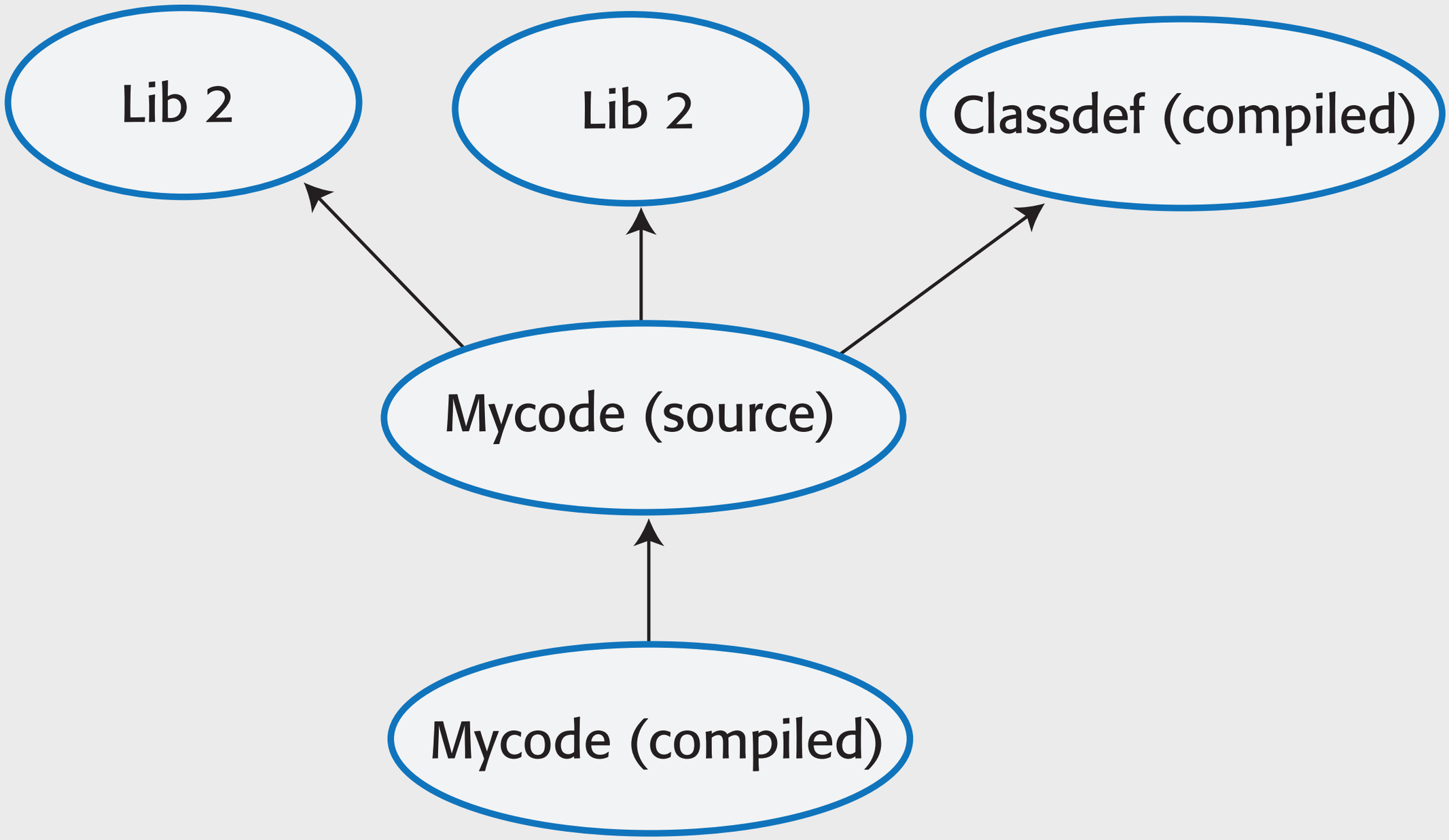
Figure 10.11 A dependency model



Dependencies

- Figure 10.11 is a dependency model that shows the dependencies for test execution.
- The 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.
- Figure 10.12 is a lower-level dependency model that shows the dependencies involved in creating the object code for a source code files called Mycode.

Figure 10.12 File dependencies

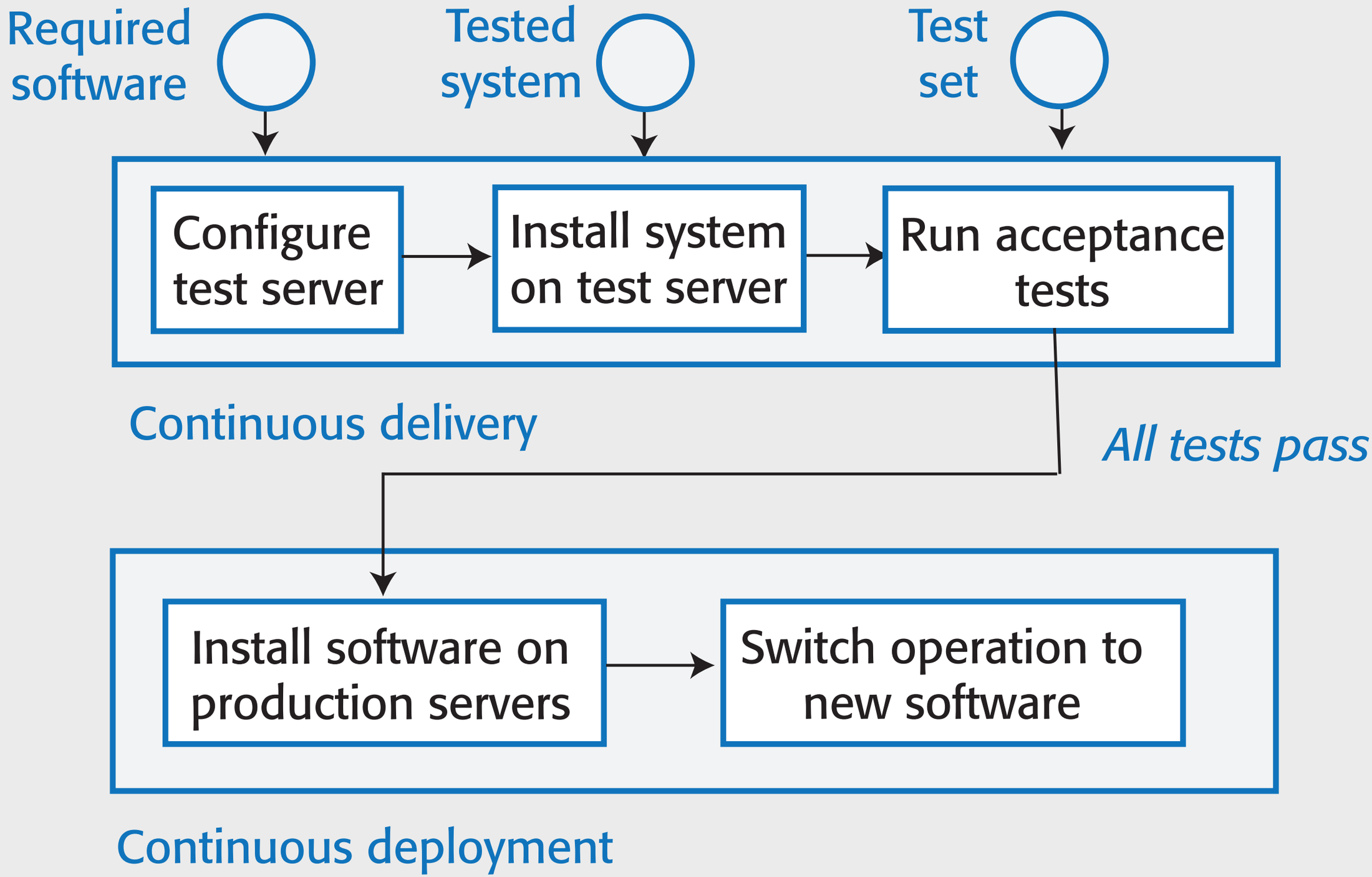


- An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file modification timestamp to decide if a source code file has been changed.
    - The modification date of the compiled code is after the modification date of the source code. The build system infers that no changes have been made to the source code and does nothing.
    - The modification date of the compiled code is before the modification date of the compiled code. The build system recompiles the source and replaces the existing file of compiled code with an updated version.
    - The modification date of the compiled code is after the modification date of the source code. However, the modification date of Classdef is after the modification date of the source code of Mycode. Therefore, Mycode has to be recompiled to incorporate these changes.

Continuous delivery and deployment

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

Figure 10.13 Continuous delivery and deployment



The deployment pipeline

- 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 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.

Figure 10.6 Benefits of continuous deployment

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 time-consuming 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 only affect a small part of the system and it will be obvious what the source of that problem is. If you bundle many changes into a single release, finding and fixing problems is 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.

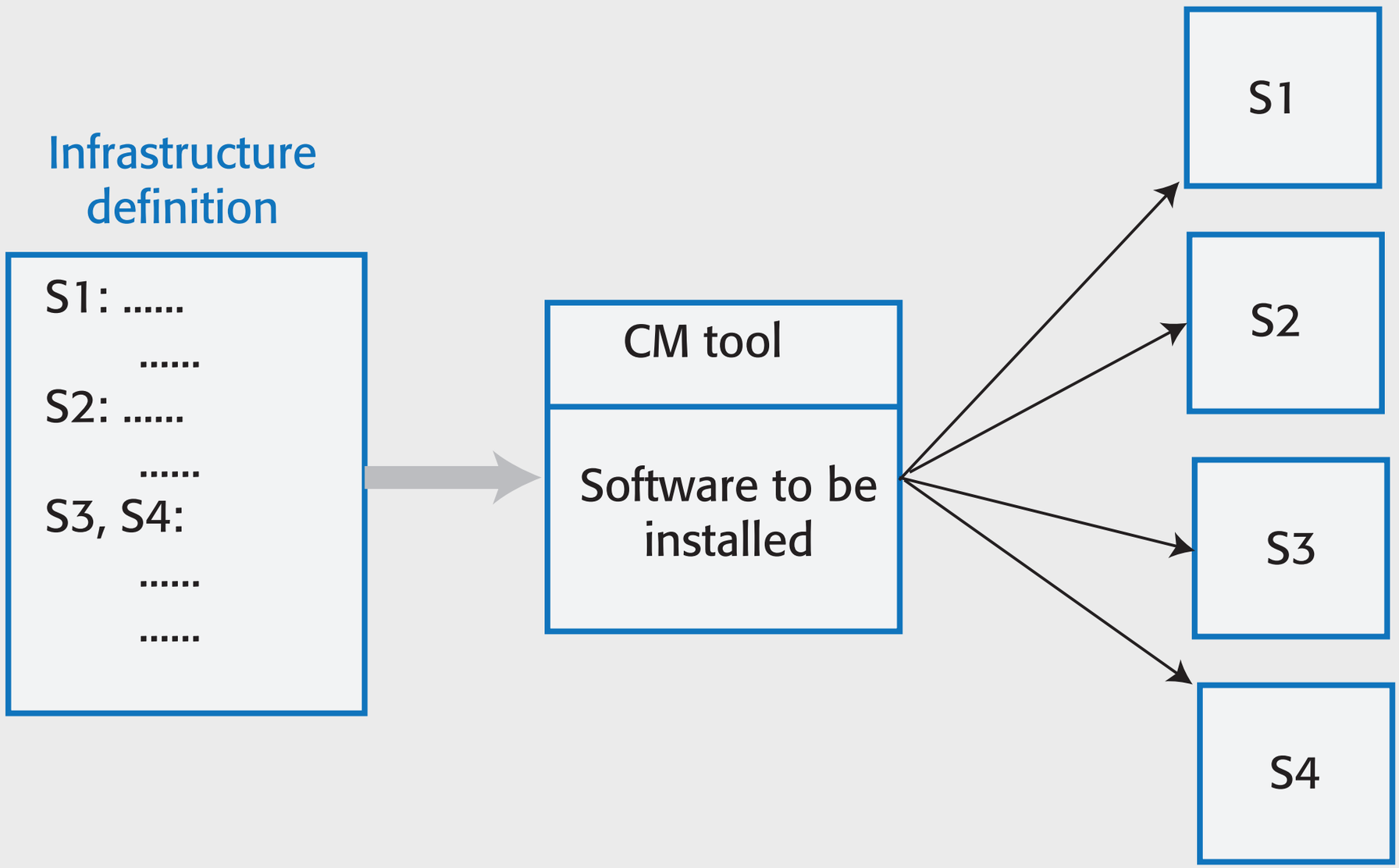
A/B testing

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 then measure and assess how new features are used to see if they do what you expect.

Infrastructure as code

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

Figure 10.14 Infrastructure as code



Benefits of infrastructure as code

- Defining your infrastructure as code and using a configuration management system solves two key problems of continuous deployment.
  - 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.
  - 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.
- 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.

Table 10.7 Characteristics of infrastructure as code

Visibility
Your infrastructure is defined as a stand-alone model that can be read, discussed, understood and reviewed by the whole DevOps team.

Reproducability
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
The complexity of managing a complex infrastructure means that 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.

Containers

- A container provides a stand-alone execution environment running on top of an operating system such as Linux.
- The software installed in a Docker container is specified using a Dockerfile, which is, essentially, a definition of your software infrastructure as code.
- You build an executable container image by processing the Dockerfile.
- Using containers makes it very simple to provide identical execution environments.
  - For each type of server that you use, you define the environment that you need and build an image for execution. You can run an application container as a test system or as an operational system; there is no distinction between them.
  - When you update your software, you rerun the image creation process to create a new image that includes the modified software. You can then start these images alongside the existing system and divert service requests to them.

DevOps measurement

- After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software.

- There are four types of software development measurement:

  - **Process measurement** You collect and analyse data about your development, testing and deployment processes.

  - **Service measurement** You collect and analyse data about the software's performance, reliability and acceptability to customers.

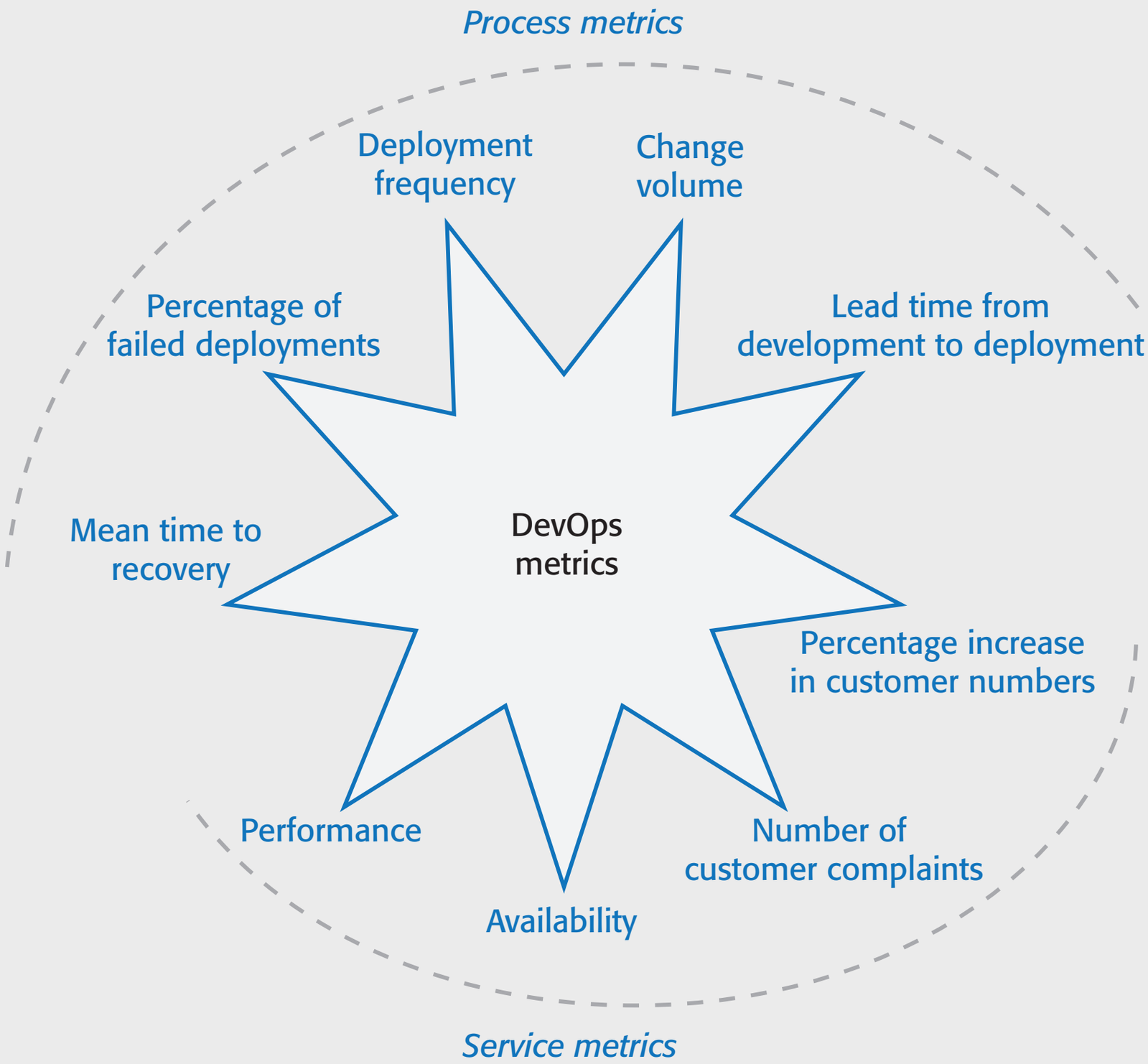
  - **Usage measurement** You collect and analyse data about how customers use your product.

  - **Business success measurement** You collect and analyse data about how your product contributes to the overall success of the business.

Automating measurement

- As far as possible, the DevOps principle of automating everything should be applied to software measurement.
- You should instrument your software to collect data about itself and you should use a monitoring system, as I explained in Chapter 6, to collect data about your software's performance and availability.
- Some process measurements can also be automated.
  - However, there are problems in process measurement because people are involved. They work in different ways, may record information differently and are affected by outside influences that affect the way they work.
-

Figure 10.15 Metrics used in the DevOps scorecard



Metrics scorecard

- Payal Chakravarty from IBM suggests a practical approach to DevOps measurement based around a metrics scorecard with 9 metrics:
  - These are relevant to software that is delivered as a cloud service. They include process metrics and service metrics
  - For the process metrics, you would like to see decreases in the number of failed deployments, the mean time to recovery after a service failure and the lead time from development to deployment.
  - You would hope to see increases in the deployment frequency and the number of lines of changed code that are shipped.
  - For the service metrics, availability and performance should be stable or improving, the number of customer complaints should be decreasing, and the number of new customers should be increasing.

Figure 10.16 Metrics trends

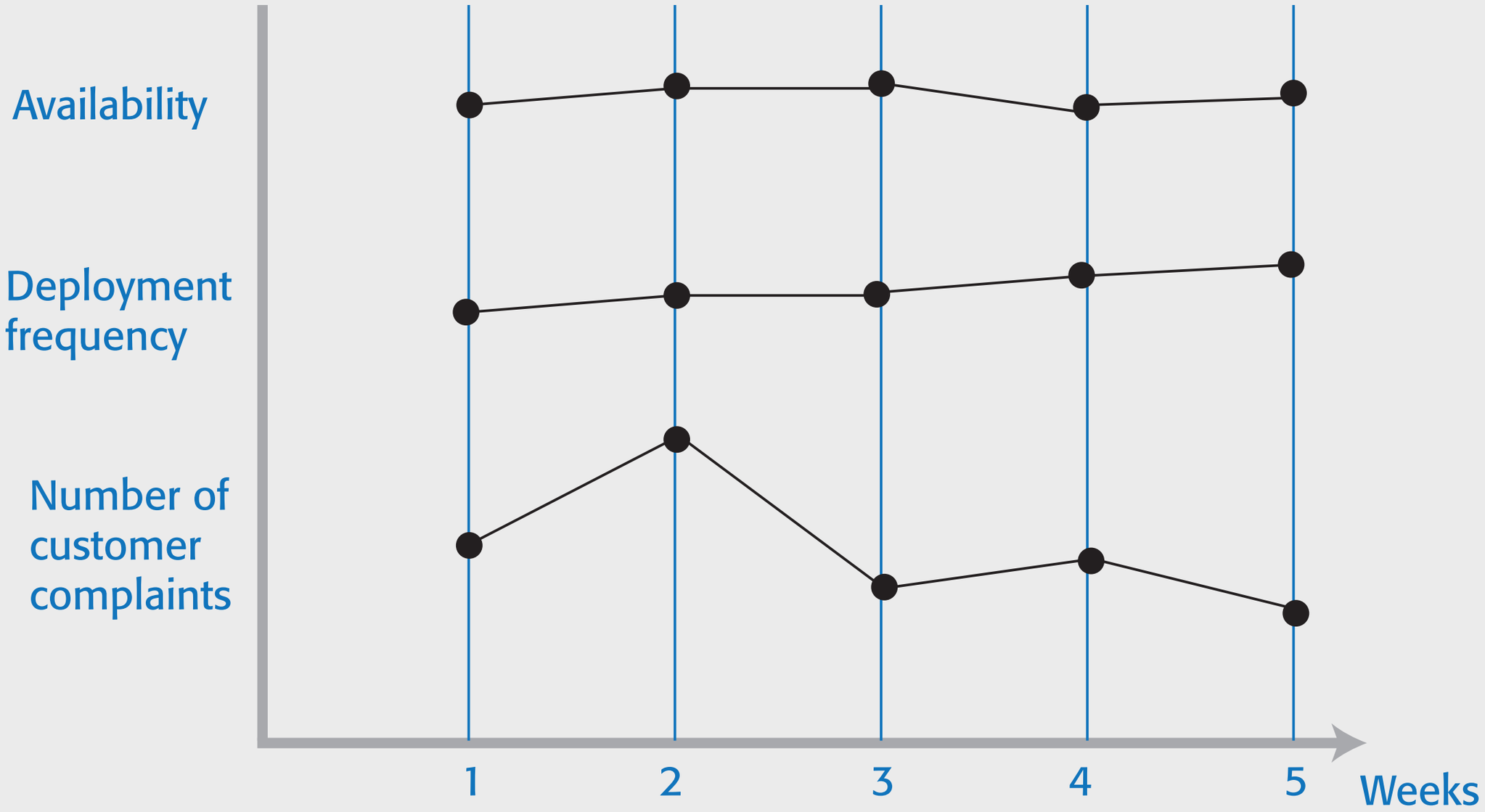
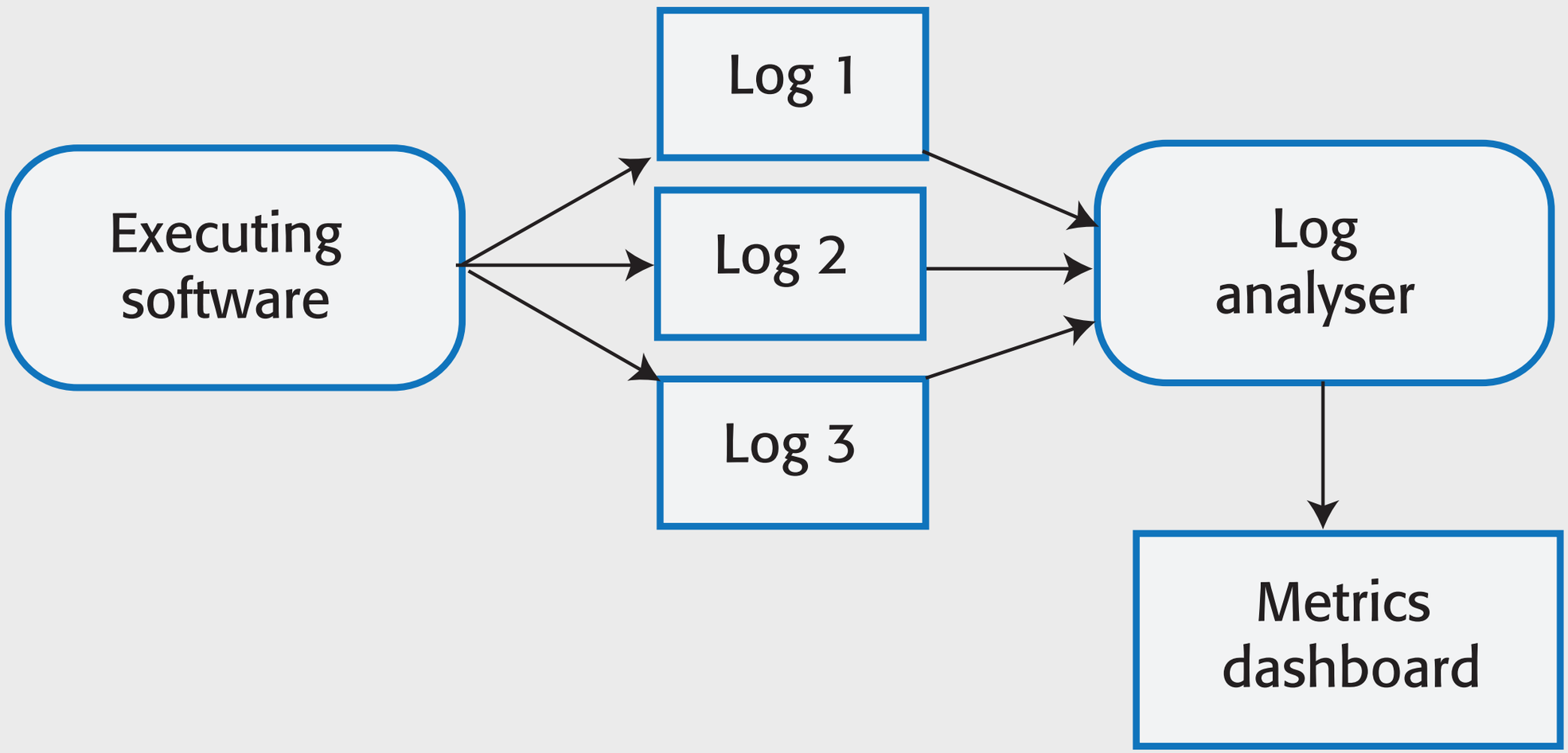


Figure 10.17 Logging and analysis



Key points 1

- DevOps is the integration of software development and the management of that software once it has been deployed for use. The same team is responsible for development, deployment and software support.
- The benefits of DevOps are faster deployment, reduced risk, faster repair of buggy code and more productive teams.
- Source code management is essential to avoid changes made by different developers interfering with each other.
- All code management systems are based around a shared code repository with a set of features that support code transfer, version storage and retrieval, branching and merging and maintaining version information.
- Git is a distributed code management system that is the most widely used system for software product development. Each developer works with their own copy of the repository which may be merged with the shared project repository.

Key points 2

- Continuous integration means that as soon as a change is committed to a project repository, it is integrated with existing code and a new version of the system is created for testing.
- Automated system building tools reduce the time needed to compile and integrate the system by only recompiling those components and their dependents that have changed.
- Continuous deployment means that as soon as a change is made, the deployed version of the system is automatically updated. This is only possible when the software product is delivered as a cloud-based service.
- Infrastructure as code means that the infrastructure (network, installed software, etc.) on which software executes is defined as a machine-readable model. Automated tools, such as Chef and Puppet, can provision servers based on the infrastructure model.
- Measurement is a fundamental principle of DevOps. You may make both process and product measurements. Important process metrics are deployment frequency, percentage of failed deployments, and mean time to recovery from failure.