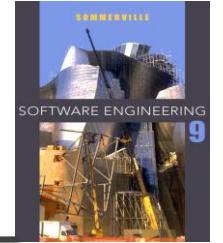
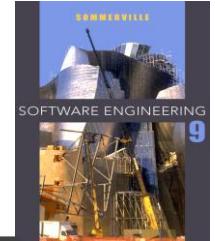


"SOFTWARE ENGINEERING",
by Ian Sommerville , *10th Edition , Pearson, 2015.*



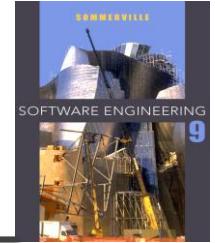
Chapter 1- Introduction

Lecture 1



Topics covered

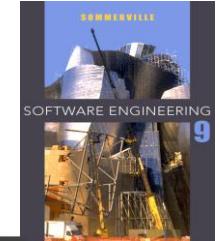
- ✧ Professional software development
 - What is meant by software engineering.
- ✧ Software engineering ethics
 - A brief introduction to ethical issues that affect software engineering.
- ✧ Case studies
 - An introduction to three examples that are used in later chapters in the book.



Software engineering

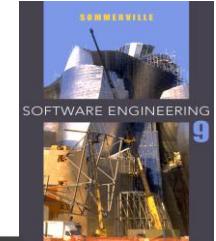
- ✧ The economies of all developed nations are dependent on software. More and more systems are software controlled.
- ✧ Software engineering is concerned with theories, methods and tools for professional software development.
- ✧ Software engineering involves wider responsibilities than simply the application of technical skills.

Software engineering



- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- ✧ Engineering discipline
 - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
- ✧ All aspects of software production
 - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Software engineering



What is software?

Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.

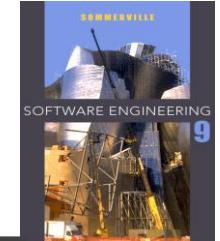
What are the attributes of good software?

Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.

What is software engineering?

Software engineering is an engineering discipline that is concerned with all aspects of software production.

Software engineering



What are the fundamental software engineering activities?

Software specification, software development, software validation and software evolution.

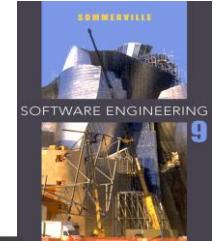
What is the difference between software engineering and computer science?

Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

What is the difference between software engineering and system engineering?

System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.

Frequently asked questions about software engineering



What are the key challenges facing software engineering?

Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.

What are the costs of software engineering?

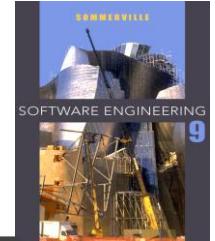
Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.

What are the best software engineering techniques and methods?

While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.

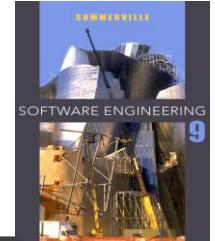
What differences has the web made to software engineering?

The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.



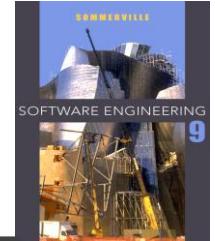
Importance of software engineering

- ✧ More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- ✧ It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.



Software costs

- ✧ Software costs often dominate (govern) computer system costs. The costs of software on a PC are often greater than the hardware cost.
- ✧ Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs.
- ✧ Software engineering is concerned with cost-effective software development.



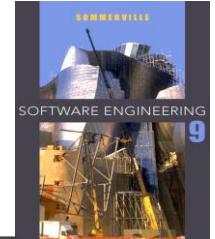
Software products

✧ Generic products

- Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
- Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

✧ Customized products

- Software that is commissioned by a specific customer to meet their own needs.
- Examples – embedded control systems, air traffic control software, traffic monitoring systems.



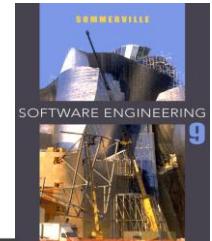
Product specification

✧ Generic products

- The **specification** of what the software should do **is owned by the software developer** and decisions on **software change** are made **by the developer**.

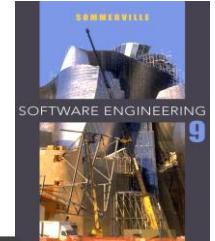
✧ Customized products

- The **specification** of what the software should do **is owned by the customer** for the software and they make decisions on **software changes that are required**.



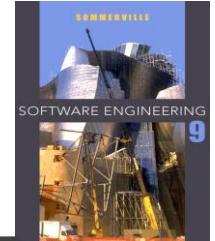
Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.



Software process activities

- ✧ **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- ✧ **Software development**, where the software is designed and programmed.
- ✧ **Software validation**, where the software is checked to ensure that it is what the customer requires.
- ✧ **Software evolution**, where the software is modified to reflect changing customer and market requirements.



General issues that affect most software

✧ Heterogeneity

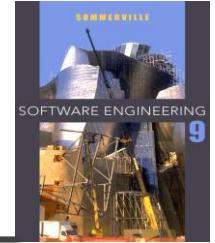
- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

✧ Business and social change

- Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

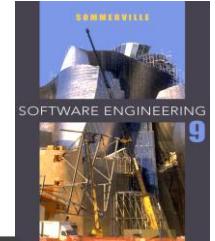
✧ Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.



Software engineering diversity

- ❖ There are many different types of software system and there is no universal set of software techniques that is applicable to all of these.
- ❖ The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.



Application types

✧ Stand-alone applications

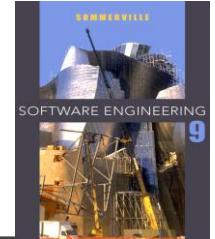
- These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

✧ Interactive transaction-based applications

- Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.

✧ Embedded control systems

- These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.



Application types

✧ Batch processing systems

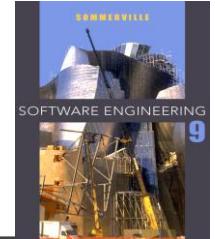
- These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.

✧ Entertainment systems

- These are systems that are primarily for personal use and which are intended to entertain the user.

✧ Systems for modelling and simulation

- These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.



Application types

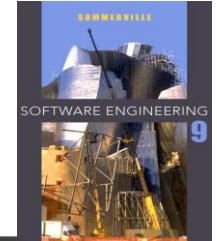
✧ Data collection systems

- These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

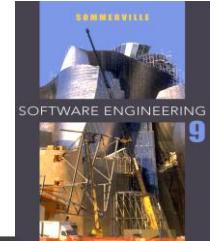
✧ Systems of systems

- These are systems that are composed of a number of other software systems.

Software engineering fundamentals



- ✧ Some fundamental principles apply to all types of software system, irrespective of the development techniques used:
 - Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
 - Dependability and performance are important for all types of system.
 - Understanding and managing the software specification and requirements (what the software should do) are important.
 - Where appropriate, you should reuse software that has already been developed rather than write new software.

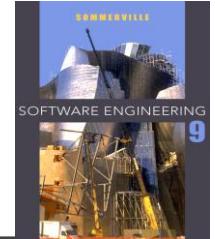


Software engineering and the web

- ✧ The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.

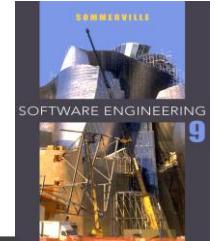
- ✧ Web services allow application functionality to be accessed over the web.

- ✧ Cloud computing is an approach to the provision of computer services where applications run remotely on the 'cloud'.



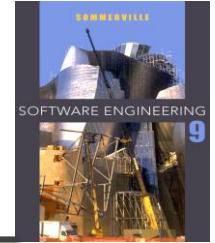
Web software engineering

- ✧ Software reuse is the dominant approach for constructing web-based systems.
 - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- ✧ Web-based systems should be developed and delivered incrementally.
 - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- ✧ User interfaces are constrained by the capabilities of web browsers.



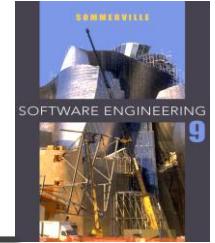
Web-based software engineering

- ✧ Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- ✧ The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system.



Key points

- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production.
- ✧ Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.
- ✧ The high-level activities of specification, development, validation and evolution are part of all software processes.
- ✧ The fundamental notions of software engineering are universally applicable to all types of system development.



Key points

- ✧ There are many different types of system and each requires appropriate software engineering tools and techniques for their development.
- ✧ The fundamental ideas of software engineering are applicable to all types of software system.

Chapter 1- Introduction

Lecture 2

Software engineering ethics

- Software engineering involves wider responsibilities than simply the application of technical skills.
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- Ethical behaviour is more than simply upholding the law but involves following a set of principles that are morally correct.

Issues of professional responsibility

- **Confidentiality**
 - Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- **Competence**
 - Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.

Issues of professional responsibility

- **Intellectual property rights**
 - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- **Computer misuse**
 - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

- The professional societies in the US have cooperated to produce a code of ethical practice.
- Members of these organisations sign up to the code of practice when they join.
- The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Motivations for the code of ethics

- Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems.
- Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm.
- To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession.

The ACM/IEEE Code of Ethics

PREAMBLE

PREAMBLE

- The short version of the code summarizes aspirations (goals) at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals.
- Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
- Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Ethical principles

1. **PUBLIC** - Software engineers shall act consistently with the public interest.
2. **CLIENT AND EMPLOYER** - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **PRODUCT** - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. **JUDGMENT** - Software engineers shall maintain integrity and independence in their professional judgment.
5. **MANAGEMENT** - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. **PROFESSION** - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. **COLLEAGUES** - Software engineers shall be fair to and supportive of their colleagues.
8. **SELF** - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical dilemmas

- Disagreement in principle with the policies of senior management.
- Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system.
- Participation in the development of military weapons systems or nuclear systems.

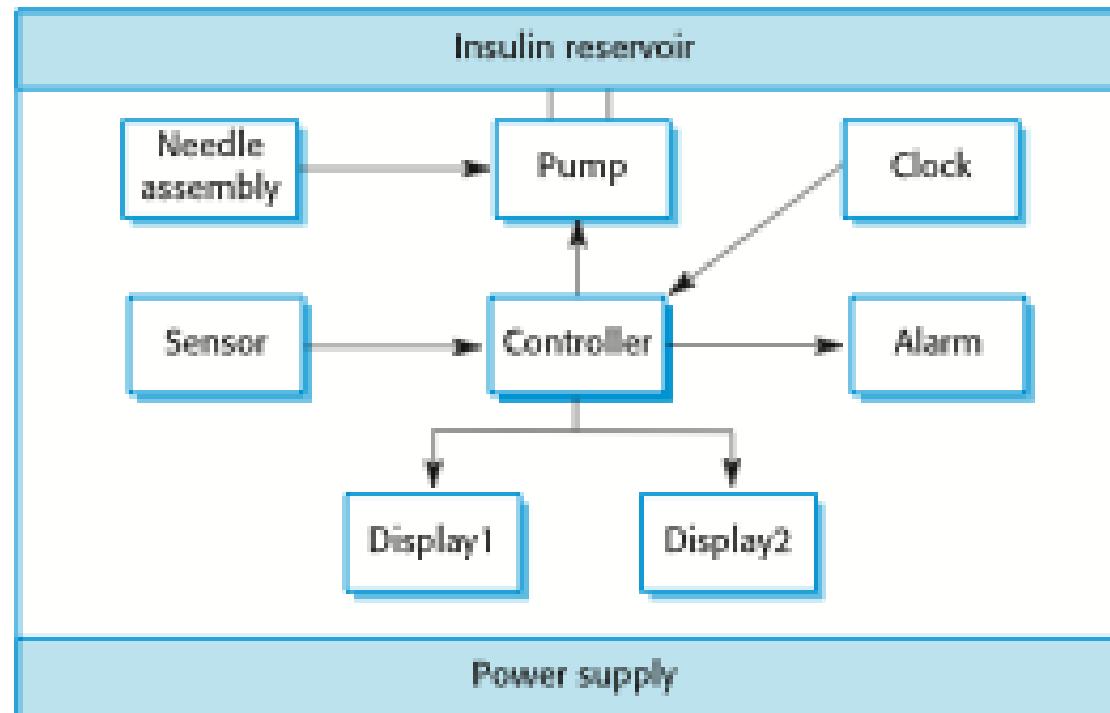
Case studies

- A personal insulin pump
 - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- A mental health case patient management system
 - A system used to maintain records of people receiving care for mental health problems.
- A wilderness weather station
 - A data collection system that collects data about weather conditions in remote areas.

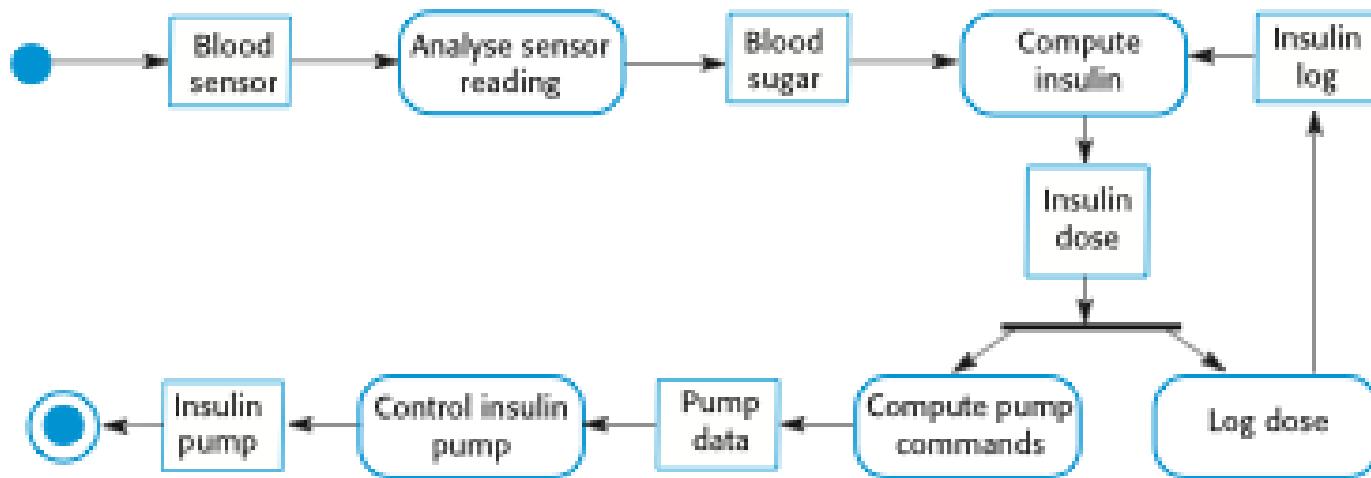
Insulin pump control system

- Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- Calculation based on the rate of change of blood sugar levels.
- Sends signals to a micro-pump to deliver the correct dose of insulin.
- Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term

Insulin pump hardware architecture



Activity model of the insulin pump



Essential high-level requirements

- The system shall be available to deliver insulin when required.
- The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
- The system must therefore be designed and implemented to ensure that the system always meets these requirements.

A patient information system for mental health care

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems

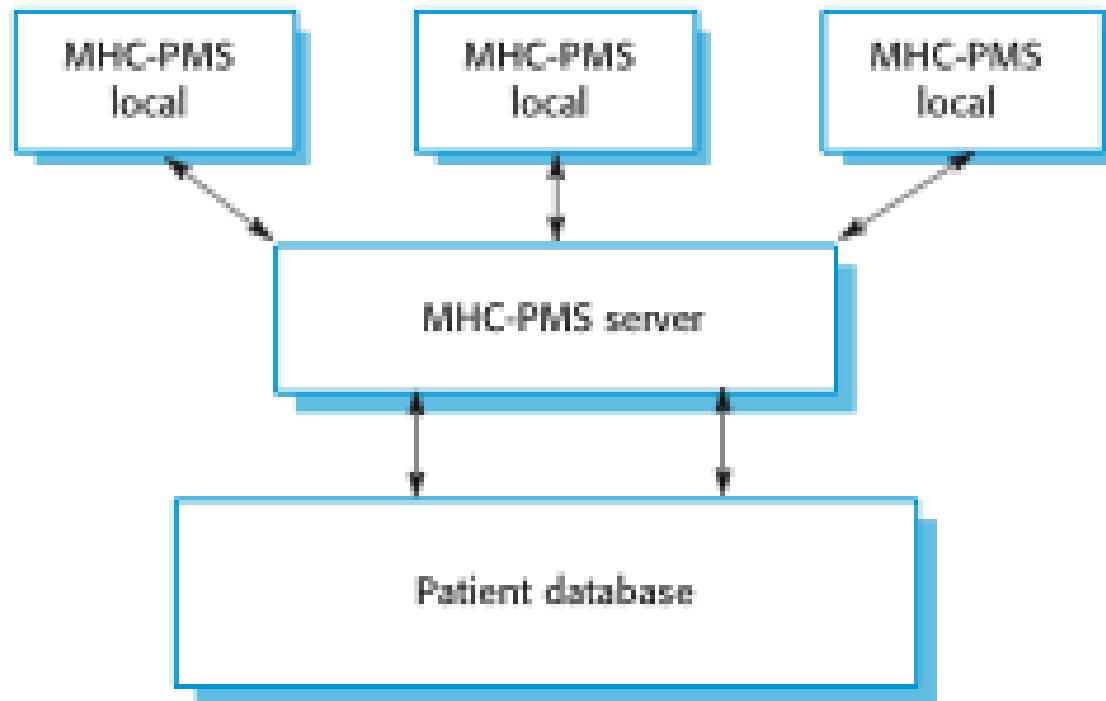
MHC-PMS

- The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics.
- It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- When the local systems have secure network access, they use patient information in the

MHC-PMS goals

- To generate management information that allows health service managers to assess performance against local and government targets.
- To provide medical staff with timely information to support the treatment of patients.

The organization of the MHC-PMS



MHC-PMS key features

- Individual care management
 - Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.
- Patient monitoring
 - The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
- Administrative reporting

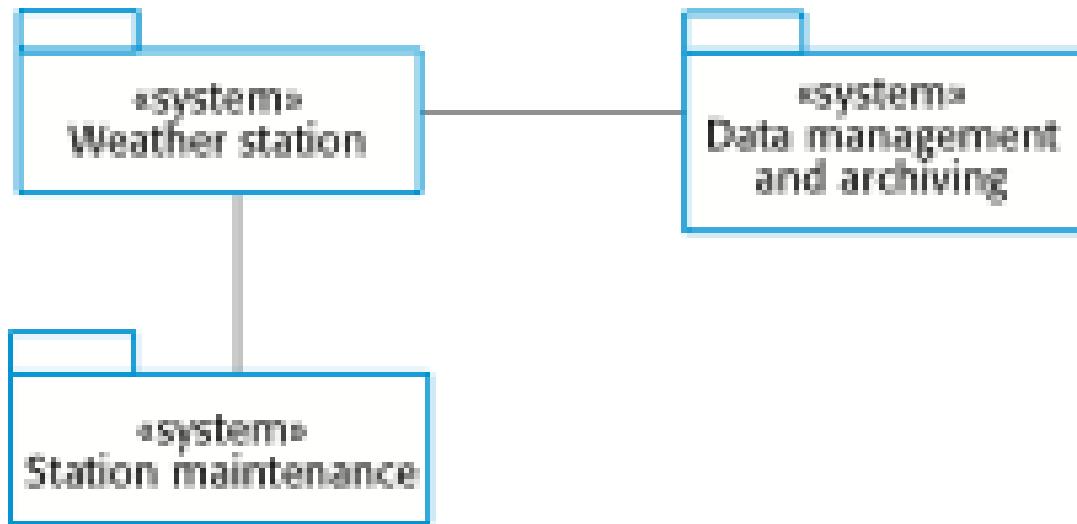
MHC-PMS concerns

- Privacy
 - It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.
- Safety
 - Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
 - The system must be available when needed

Wilderness weather station

- The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
 - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure

The weather station's environment



Weather information system

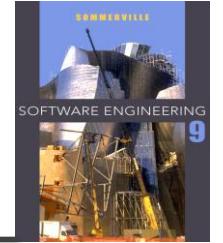
- The weather station system
 - This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.
- The data management and archiving system
 - This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.
- The station maintenance system
 - This system can communicate by satellite with all wilderness weather stations to monitor the health of

Additional software functionality

- Monitor the instruments, power and communication hardware and report faults to the management system.
- Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- Support dynamic reconfiguration where parts

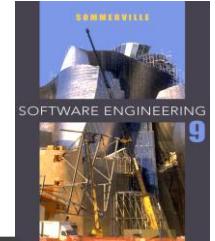
Key points

- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.
- Professional societies publish codes of conduct which set out the standards of behaviour expected of their members.
- Three case studies are used in the book:
 - An embedded insulin pump control system
 - A system for mental health care patient management
 - A wilderness weather station



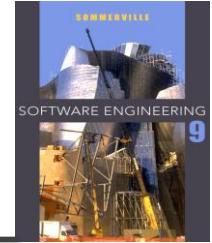
Chapter 2 – Software Testing

Lecture 3



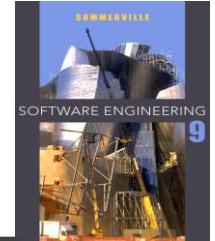
Topics covered

- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing



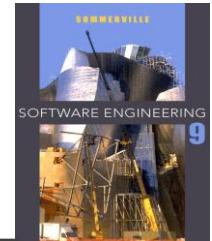
Program testing definition

- ✧ Testing is intended to show that a program does, what it is intended to do and to discover program defects before it is put into use.
- ✧ When you test software, you execute a program using artificial data.
- ✧ You check the results of the test run for errors, anomalies or information about the program's non-functional attributes (quality attributes). [Fun. Req.(describe what the sys should do – function -objective)–Non-fun.Req.(describe how the system works like performance - reliability –security – maintainability)]
- ✧ Can reveal the presence of errors NOT their absence.
- ✧ Testing is part of a more general verification and validation process, which also includes static validation techniques.



Program testing goals

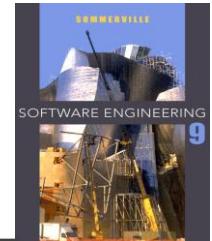
- ✧ To demonstrate to the developer and the customer that the software meets its requirements.
 - For custom software, this means that there should be at least one test for every requirement in the requirements document.
 - For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- ✧ To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
 - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.



Validation and defect testing

- ✧ The first goal leads to **validation testing**
 - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.

- ✧ The second goal leads to **defect testing**
 - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.



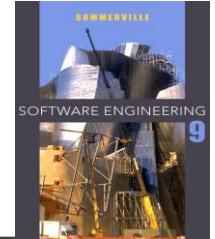
Testing process goals

✧ Validation testing

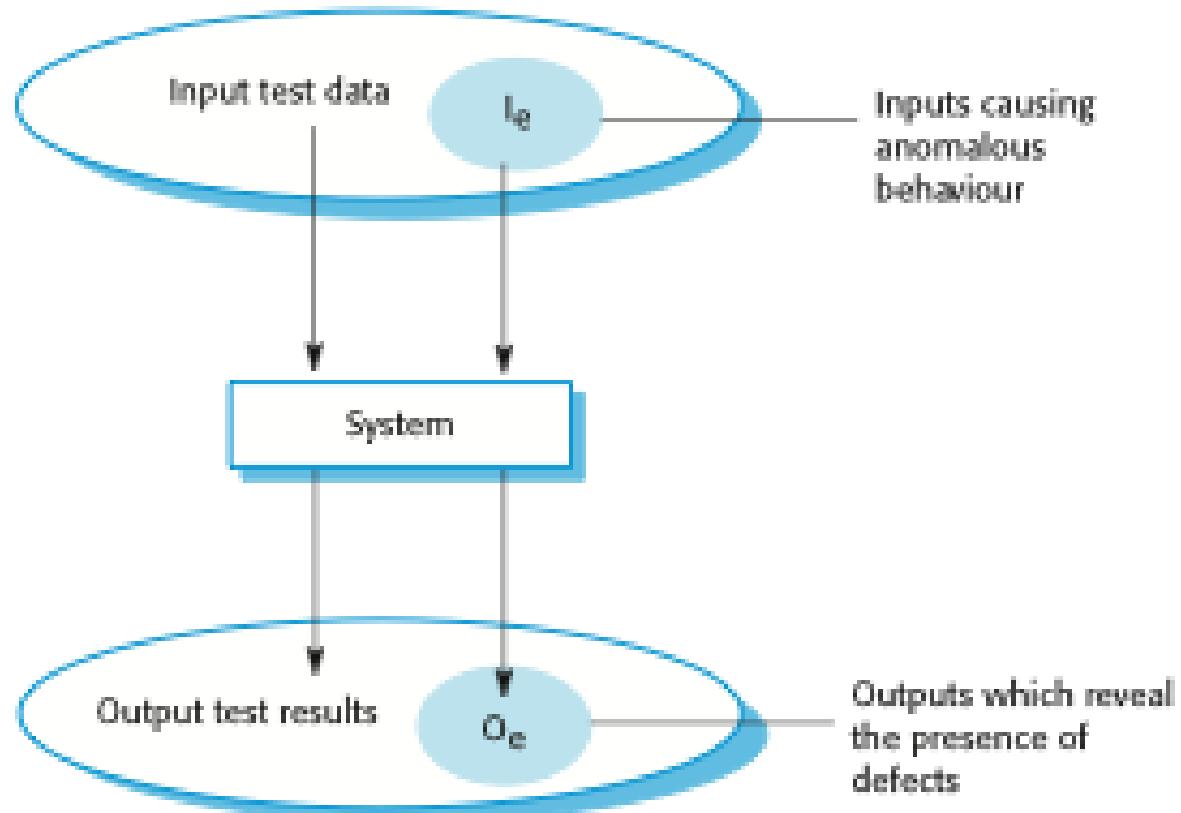
- To demonstrate to the developer and the system customer that the software meets its requirements
- A successful test shows that the system operates as intended.

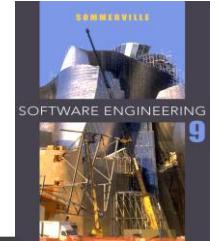
✧ Defect testing

- To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.



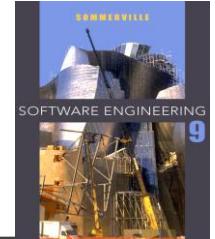
An input-output model of program testing





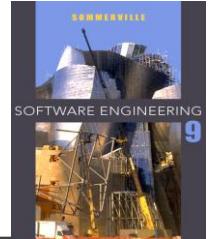
Verification vs validation

- ✧ Verification:
"Are we building the product right".
- ✧ The software should conform (follow) to its specification.
- ✧ Validation:
"Are we building the **right product**".
- ✧ The software should do what the user really requires.



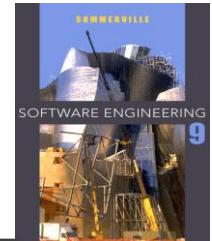
V & V confidence

- ❖ Aim of V & V is to establish **confidence** that the system is '**fit for purpose**'.
- ❖ Depends on: (system's purpose, user expectations and marketing environment)
 - Software purpose
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have low expectations of certain kinds of software.
 - Marketing environment
 - Getting a product to market early may be more important than finding defects in the program.

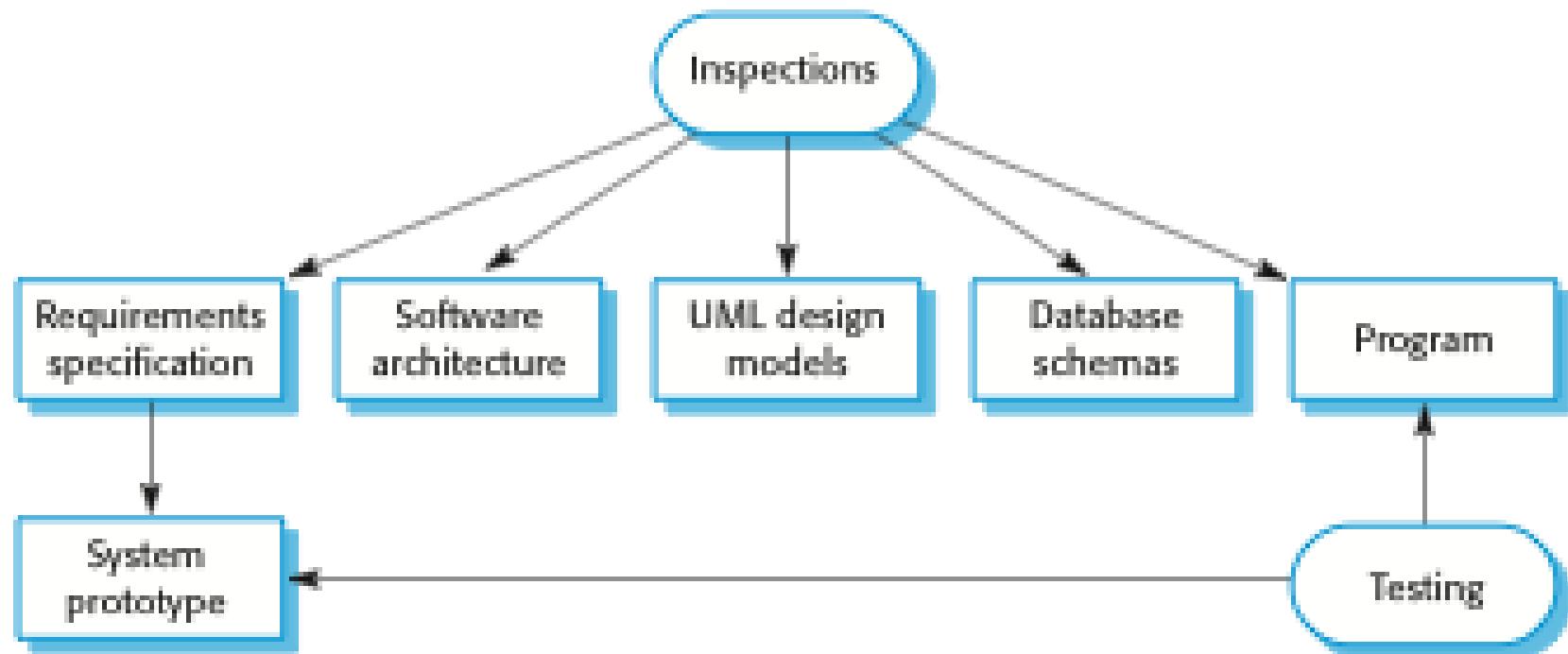


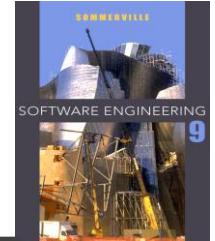
What are the V & V checking approaches? Inspections and testing

- ❖ **Software inspections** Concerned with analysis of the static system representation to discover problems (static verification)
 - May be supplement by tool-based document and code analysis.
- ❖ **Software testing** Concerned with exercising and observing product behaviour (dynamic verification)
 - The system is executed with test data and its operational behaviour is observed.



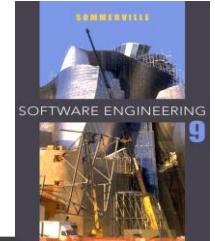
Inspections and testing





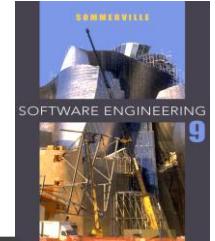
Software inspections

- ✧ These involve people examining the source representation with the aim of discovering anomalies and defects.
- ✧ Inspections not require execution of a system so may be used before implementation.
- ✧ They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- ✧ They have been shown to be an effective technique for discovering program errors.



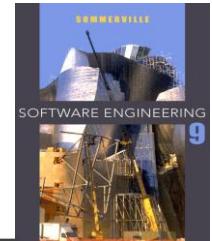
Advantages of inspections

- ✧ During testing, errors can mask (hide) other errors. But because inspection is a static process, you don't have to be concerned with interactions between errors.
- ✧ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.



Inspections and testing

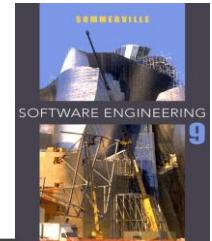
- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the V & V process.
- ✧ Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- ✧ Inspections cannot check non-functional characteristics such as performance, usability, etc.



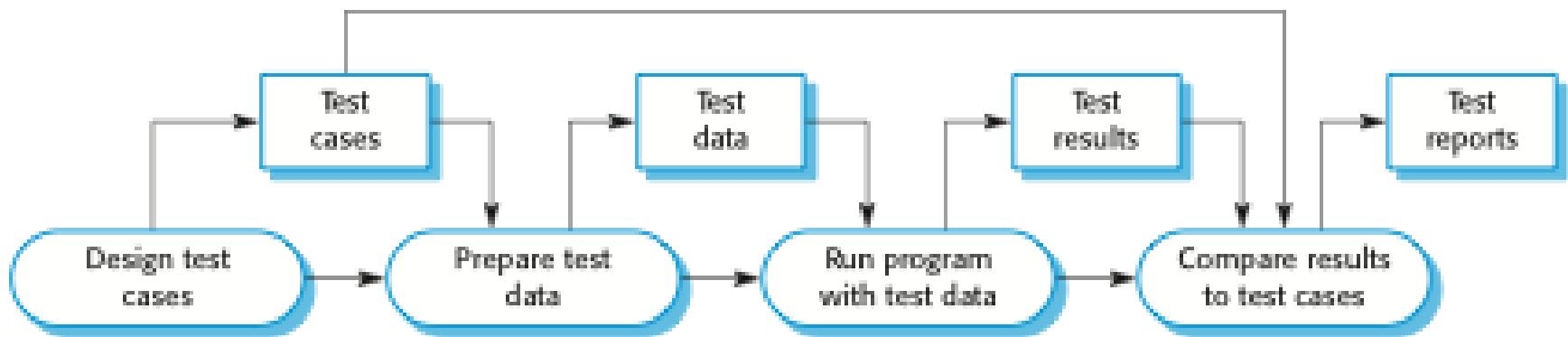
White-box and Black-box Testing

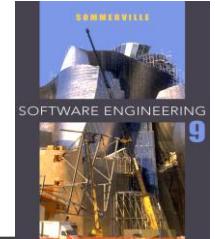
- 1) Black-box testing : Deriving tests from external descriptions of the software, including specifications, requirements, and design

- 2) White-box testing : Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements



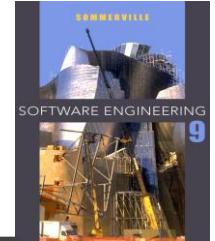
A model of the software testing process





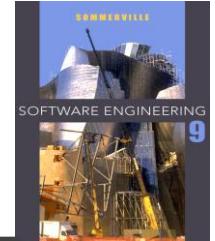
Stages of testing

- ✧ **Development testing**, where the system is tested during development to discover bugs and defects.
- ✧ **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
- ✧ **User testing**, where users or potential users of a system test the system in their own environment.



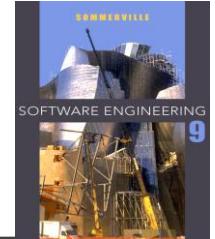
Development testing:

- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
 - **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
 - **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
 - **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.



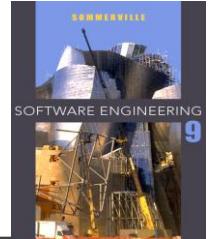
Unit testing

- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ It is a defect testing process.
- ✧ Units may be:
 - Individual functions or methods within an object
 - Object classes with several attributes and methods
 - Composite components with defined interfaces used to access their functionality.



Object class testing

- ✧ Complete test coverage of a class involves
 - Testing all operations associated with an object
 - Setting and interrogating all object attributes
 - Exercising the object in all possible states.
- ✧ Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

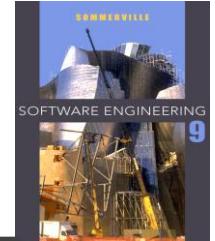


The weather station object interface

WeatherStation

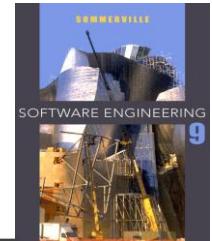
Identifier

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



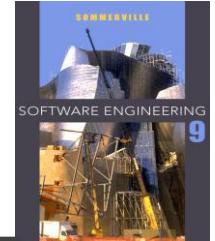
Weather station testing

- ✧ Need to define test cases for report Weather, calibrate, test, startup and shutdown.
- ✧ Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- ✧ For example:
 - Shutdown -> Running-> Shutdown
 - Configuring-> Running-> Testing -> Transmitting -> Running
 - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running



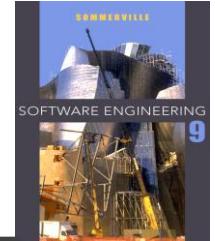
Automated testing

- ✧ Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- ✧ In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- ✧ Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.



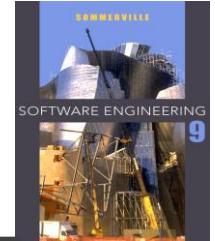
Automated test components

- ✧ A **setup part**, where you initialize the system with the test case, namely the inputs and expected outputs.
- ✧ A **call part**, where you call the object or method to be tested.
- ✧ An **assertion part**, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful. If false, then it has failed.



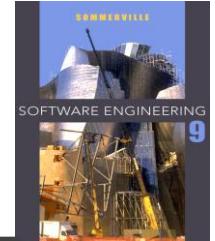
Unit test effectiveness

- ✧ The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- ✧ If there are defects in the component, these should be revealed by test cases.
- ✧ This leads to 2 types of unit test case:
 - The first of these should reflect normal operation of a program and should show that the component works as expected.
 - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.



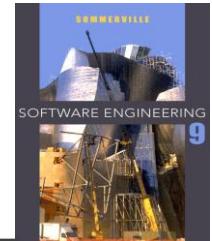
Testing strategies

- ✧ Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
 - You should choose tests from within each of these groups.
- ✧ Guideline-based testing, where you use testing guidelines to choose test cases.
 - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

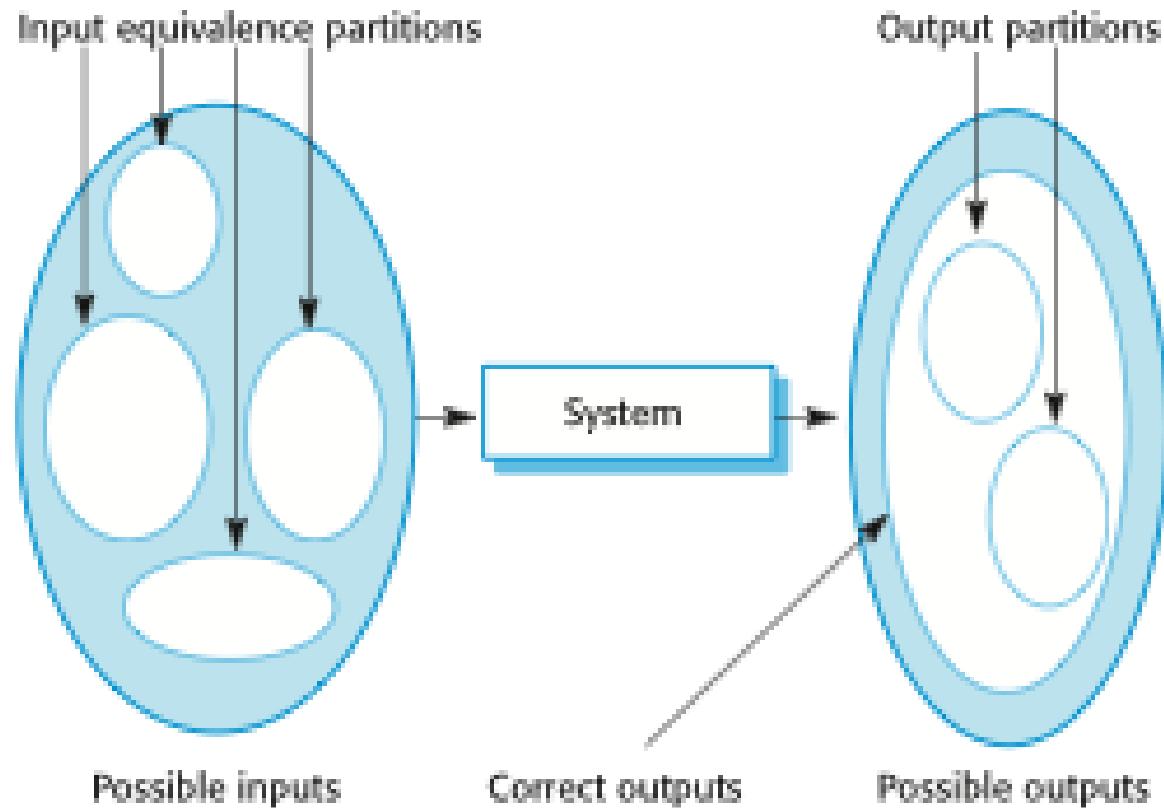


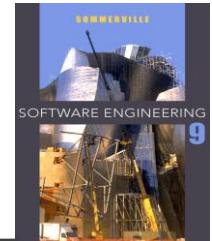
Partition testing

- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

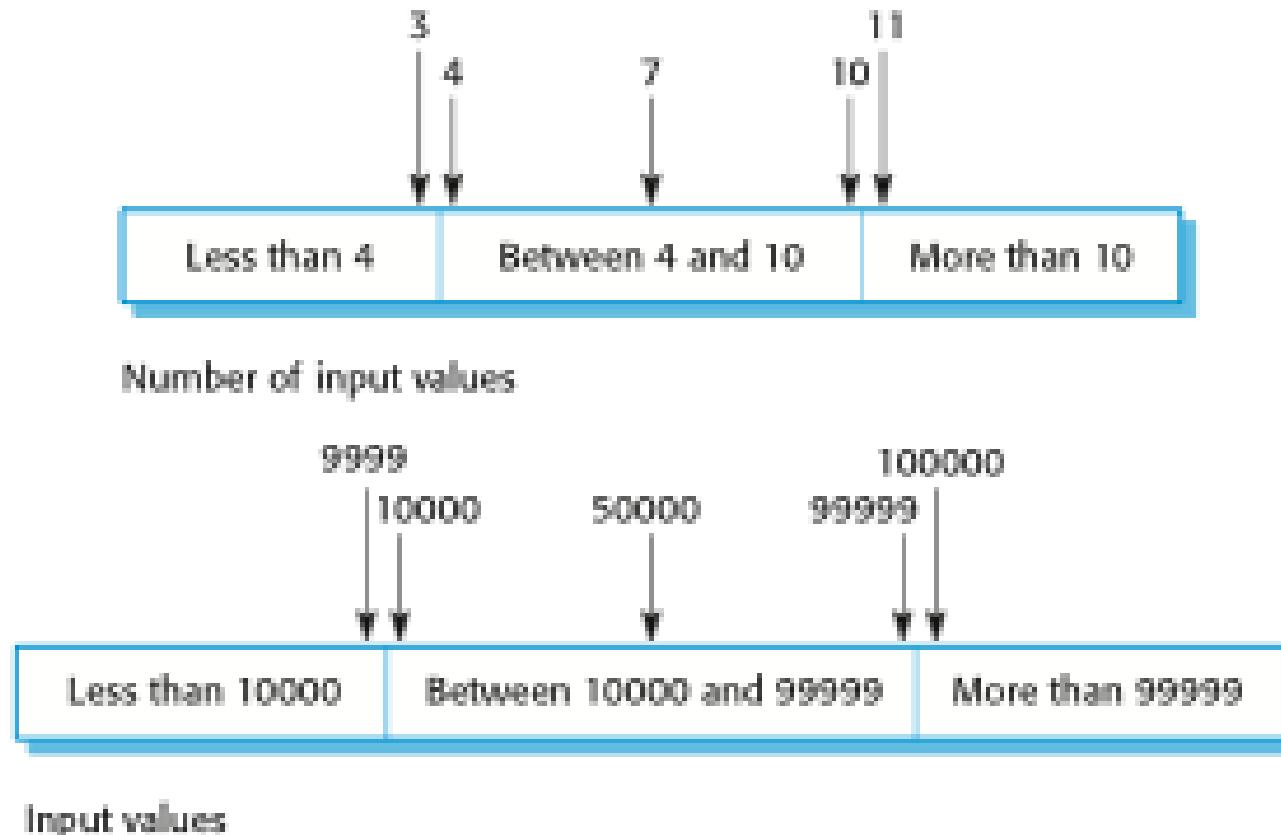


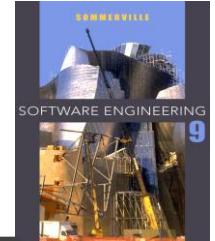
Equivalence partitioning





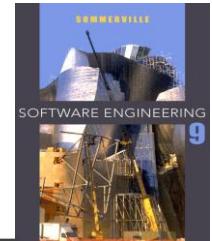
Equivalence partitions





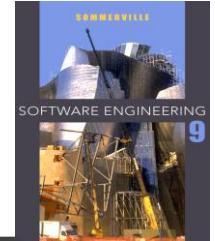
Testing guidelines (sequences)

- ✧ Test software with sequences which have only a single value.
- ✧ Use sequences of different sizes in different tests.
- ✧ Derive tests so that the first, middle and last elements of the sequence are accessed.
- ✧ Test with sequences of zero length.



General testing guidelines

- ✧ Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system
 - Choose inputs that force the system to generate all error messages
 - Design inputs that cause input buffers to overflow
 - Repeat the same input or series of inputs numerous times
 - Force invalid outputs to be generated
 - Force computation results to be too large or too small.



Key points

- ✧ Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- ✧ Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- ✧ Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.

Chapter 2 – Software Testing

Lecture 4

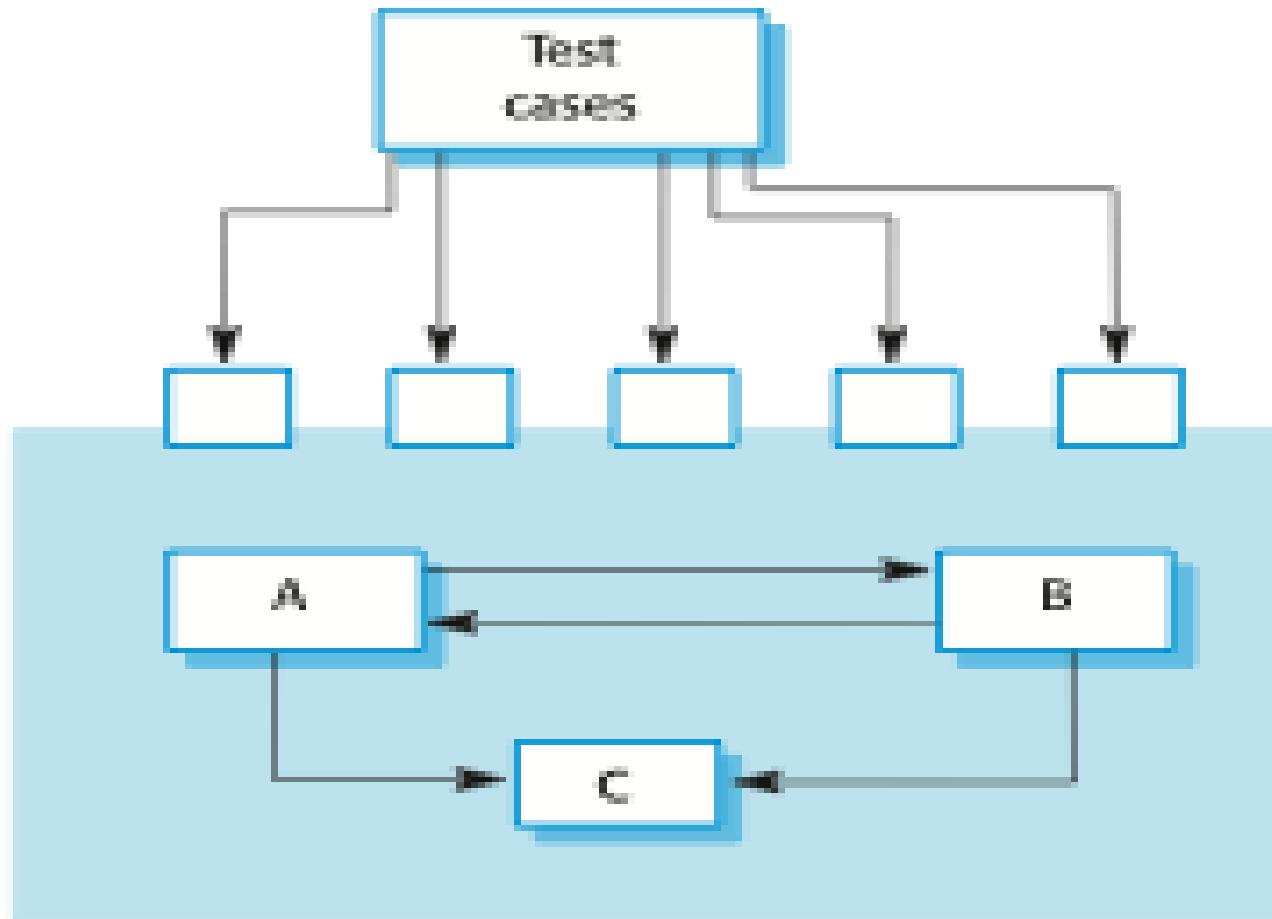
Component testing

- Software components are often composite components that are made up of several interacting objects.
 - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
 - You can assume that unit tests on the individual objects within the component have been completed.

Component testing

- Testing of individual program components;
- Individual components are tested to ensure that they operate correctly.
- Each component is tested independently without other system components.
- Components may be simple entities such as functions or object classes, or may be coherent grouping of these entities.
- Component testing is usually the responsibility of the component developer;
- Tests are derived from the developer's experience.

Interface testing



Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
 - **Parameter interfaces** Data passed from one method or procedure to another.
 - **Shared memory interfaces** Block of memory is shared between procedures or functions.
 - **Procedural interfaces** Sub-system encapsulates a set of procedures to be called by other sub-systems.
 - **Message passing interfaces** Sub-systems request services from other sub-systems

Interface errors

- Interface misuse
 - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
 - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
 - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

System testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing, checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing, tests the emergent (promising) behavior of a system.

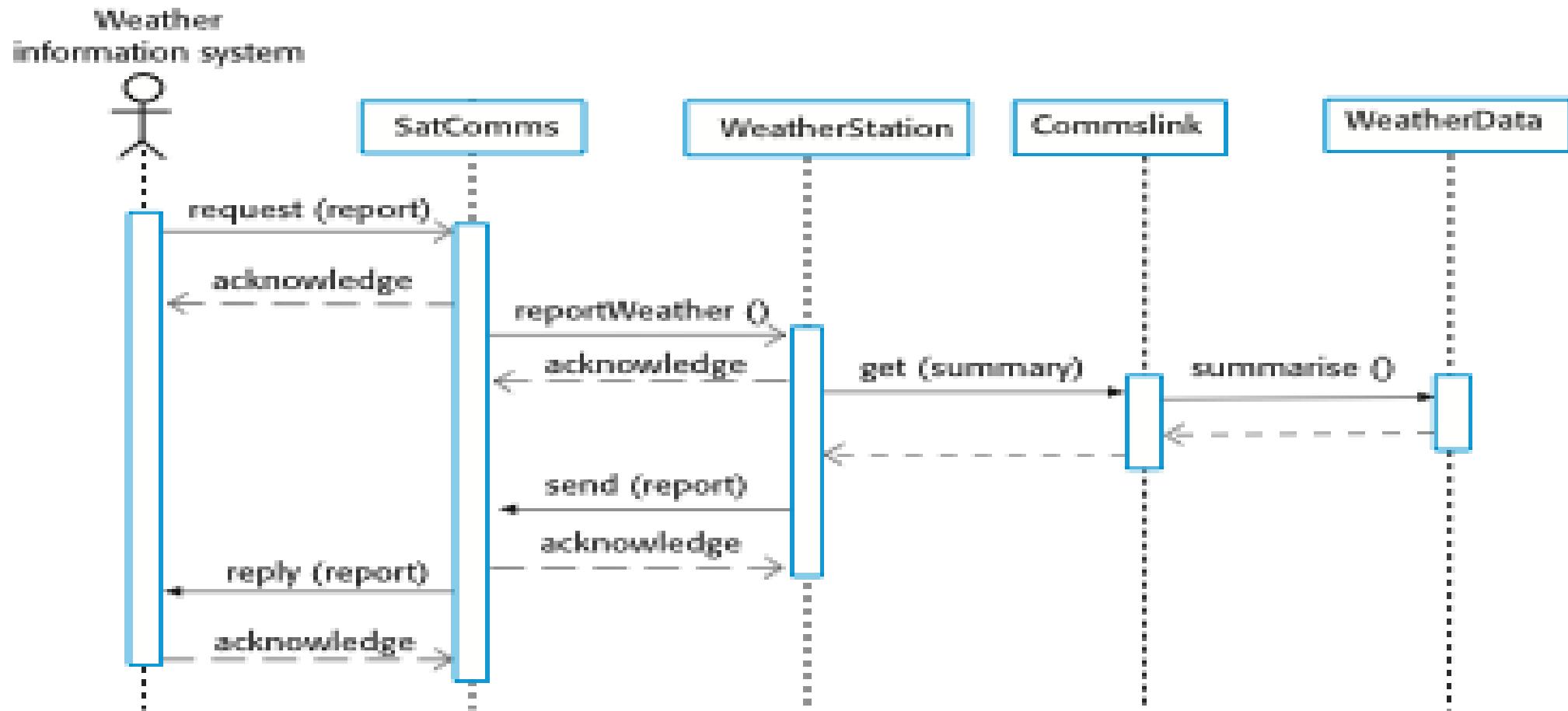
System and component testing

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
 - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Use-case testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

Collect weather data sequence chart



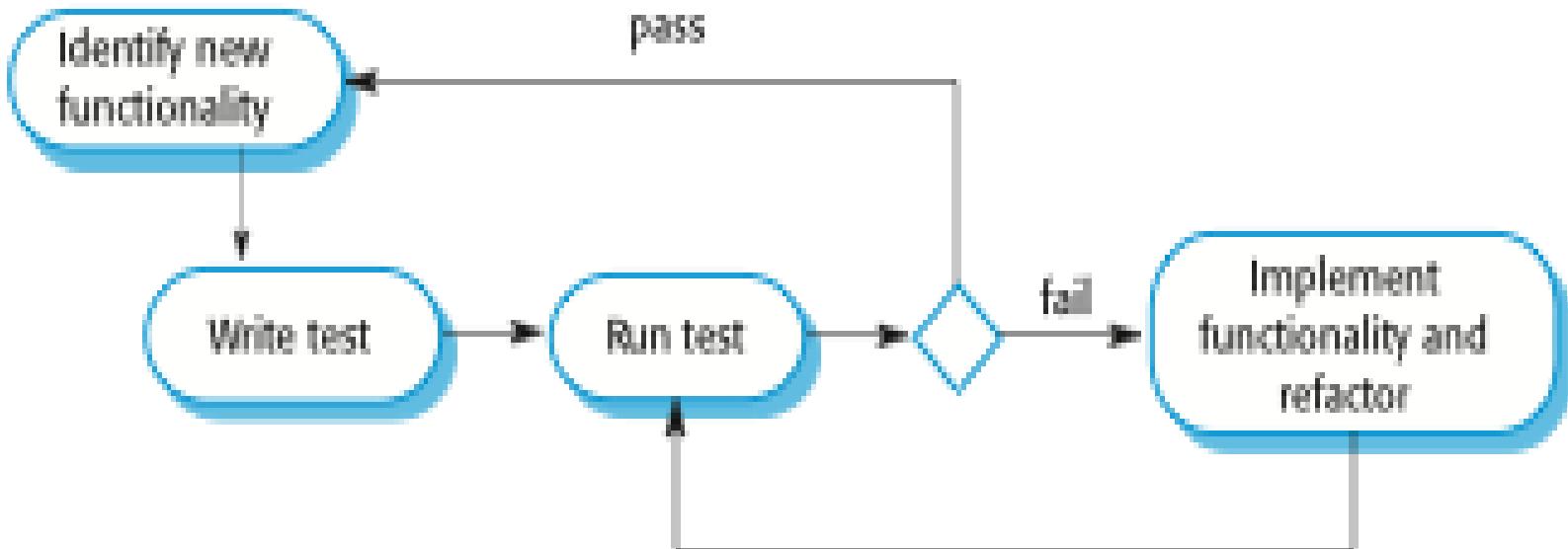
Testing policies

- Full system testing is impossible, so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
 - All system functions that are accessed through menus should be tested.
 - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
 - Where user input is provided, all functions must be tested with both correct and incorrect input.

Test-driven development

- Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- Tests are written before code and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development



TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development

- Code coverage
 - Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing
 - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
 - When a test fails, it should be obvious where the problem lies.
The newly written code needs to be checked and modified.
- System documentation
 - The tests themselves are a form of documentation that describe what the code should be doing.

Regression testing

- After a defect in a program has been discovered, you have to correct it and revalidate the system. This may involve re-inspecting the program or regression testing.
- Regression Testing is used to check that the changes made to a program have not introduced new faults.
- In principle, you should repeat all tests after every defect repair. In practice this is usually too expensive. So in a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- So, you should identify the dependencies between components and the tests associated with each component. You may then run a subset of the system test cases to check the modified component and its dependents.
- Tests must run ‘successfully’ before the change is committed (dedicated).

Integration testing

- System integration involves identifying clusters of component that deliver some system functionality and integrating these by adding code that makes them work together.
- Top-down integration:
 - Top-Down Testing : Test the main procedure, then go down through procedures it calls, and so.
 - The overall skeleton of the system is developed first, and components are added to it.
- Bottom-up integration :
 - Bottom-Up Testing : Test the leaves in the tree (procedures that make no calls), and move up to the root.
- The major problem that arises during integration testing is localizing errors. To simplify error localization, systems should be incrementally integrated.

Incremental integration testing

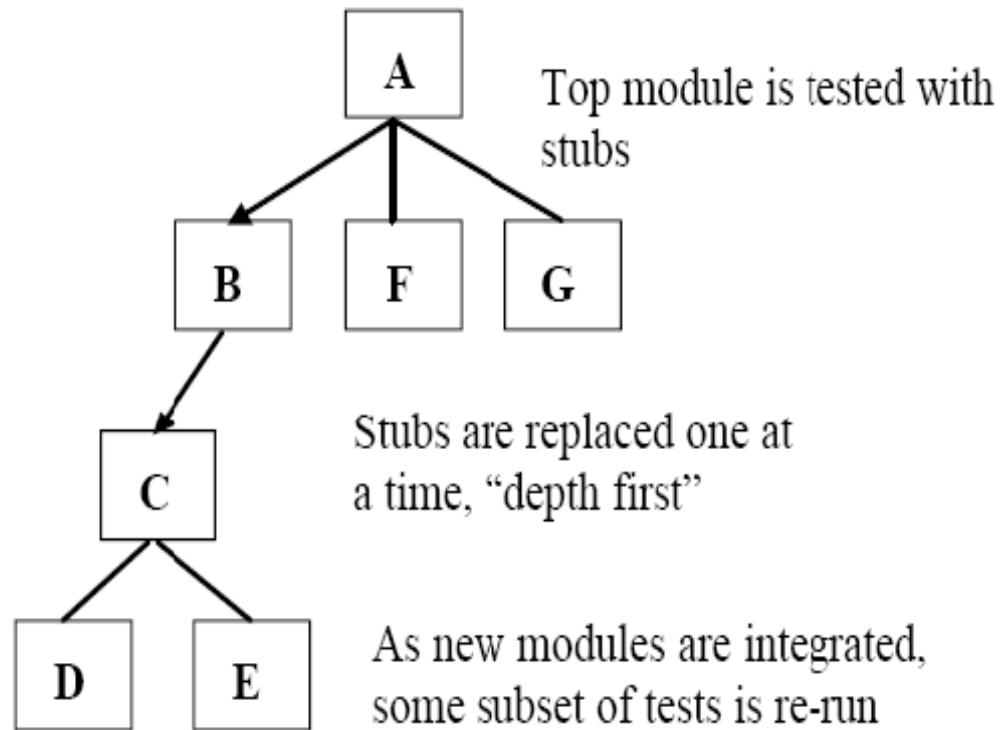
- When a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required to verify the new system functionality.
- Rerunning an existing set of tests is called regression testing. If regression testing exposes problems, then you have to check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added increment functionality.
- Regression testing is clearly an expensive process and is impractical without some automated support.

S/W drivers and stubs

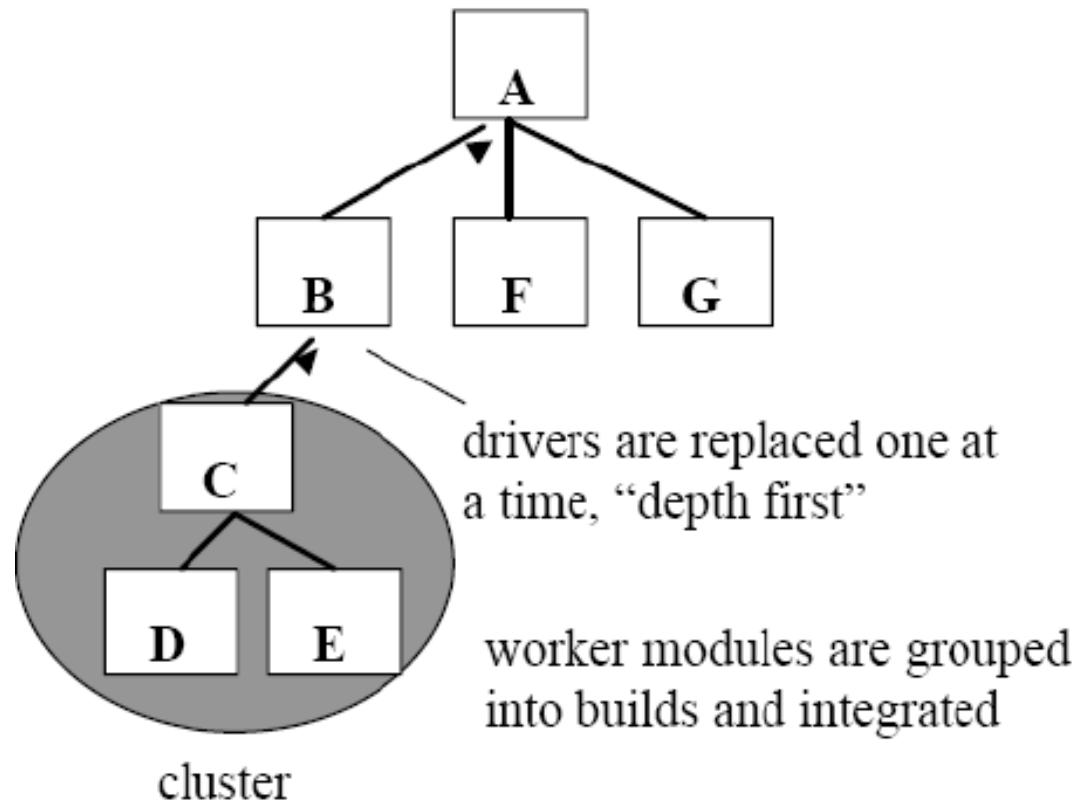
- In both top-down and bottom-up integration, we usually have to develop additional code to simulate other components and allow the system to execute.
- In top-down integration, If the module under test needs to call methods on another module, you should implement a **stub class** with the same name as the class that contains those methods. This class would not implement any behavior, but it should 'fake' it.
- For example, if you have a method called `getCustomerAddress(customerID: Integer)` that returns the address of the specified customer, your class would not access the file (or database). Instead, the method you implement in your stub class, might simply return a static string such as `'return "123 Apple Lane";'`.
- Similarly, in bottom-up integration, if the module under test provides methods that are to be called from another class, you should implement a **driver class**. A driver class simulates the use of a class, by calling its methods in a typical way

Integration testing

Top-Down Integration



Bottom-Up Integration



Release testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
 - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release testing and system testing

- Release testing is a form of system testing.
- Important differences:
 - A separate team that has not been involved in the system development, should be responsible for release testing.
 - System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

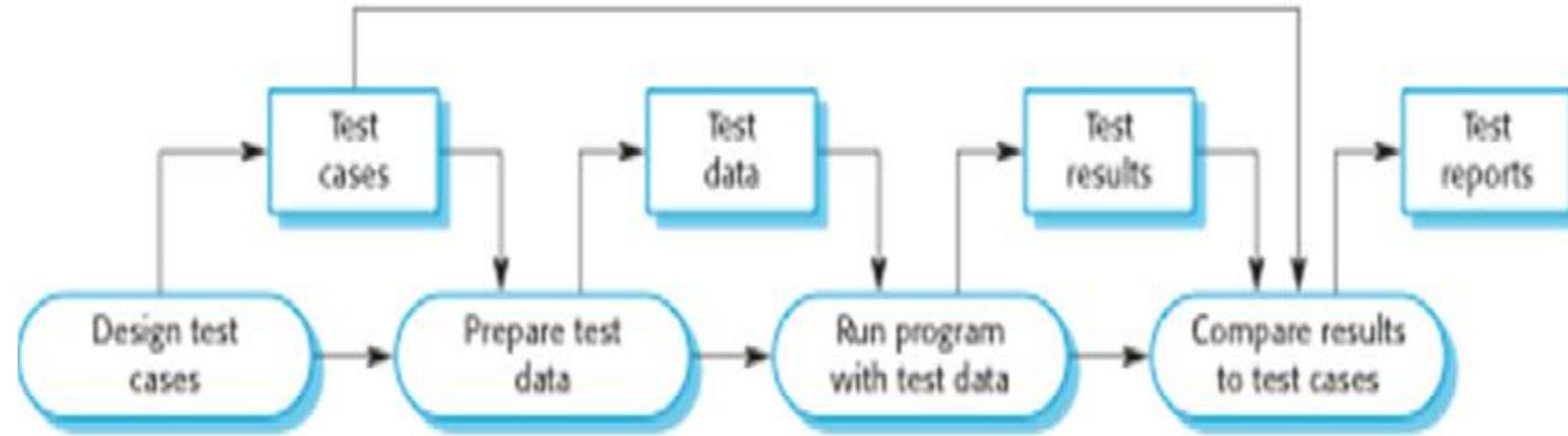
Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately (intentionally) overloaded to test its failure behavior.
- Performance tests have to be designed to ensure that the system can process its intended load

Stress testing

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

A model of the software testing process



What is test case?

- Description of everything needed for one test run.
- Test case must contain: (Test # - Test objective/purpose - Components under test / Identifier of the requirement which is covered by the test case - Testing steps - Hardware and software configuration to run the test - Inputs to the software under test - Expected outputs).

What is test data?

Test data Inputs which have been devised to test the system.

Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Test design approaches: There are various approaches that can be used to test case design:
 - Requirements-based testing;
 - Partition testing;
 - Structural testing;

Requirements based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- Test cases are designed to test the system requirements.
- This is mostly used at the system-testing stage as system requirements are usually implemented by several components.
- A requirement should be written in such a way that a test can be designed so that an observer can check that the requirement has been satisfied.
- Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.

Requirements based testing

- Ex: LIBSYS requirements
- The user shall be able to search either all of the initial set of databases or select a subset from it.
- LIBSYS tests: Possible tests for this requirement, assuming that a search function has been tested, are:
 - 1) Initiate user search for searches for items that are known to be present and known not to be present, where the set of databases includes 1 database.
 - 2) Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes 2 databases
 - 3) Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than 2 databases.
 - 4) Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
 - 5) Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.

Partition testing

- One systematic approach to test case design for both system and component testing. It is based on identifying all partitions for a system or component.
- Input data and output results often fall into different classes where all members of a class are related and have common characteristics such as positive numbers, negative numbers and menu selections.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Once you have identified a set of partitions, we can choose test cases from each of these partitions.

Equivalence partitioning

- Equivalence testing divides the input domain into classes of data from which test cases can be derived to reduce the total number of test cases that must be developed.
- We identify partitions by using the program specification or user documentation.
- A good rule of thumb (scan or brows) for test case selection is to choose test cases on the boundaries of the partitions (Boundary value analysis) plus cases close to the mid-point of the partition.
- An equivalence class is considered *covered* when at least one test has been selected from it.
- In partition testing our goal is to cover all equivalence classes.

Equivalence partitioning

- Boundary Value Analysis (BVA): Look for boundary values to generate test cases
 - To illustrate the derivation of test cases, we use the following Search routine specification. This component searches a sequence of elements for a given element(the key).

Procedure Search

Input: Key, Sequence of elements T.

Output: Found (yes/no), Index in the sequence L

Pre-condition: the sequence has at least one element

Post-condition: the element is found (Found and $T(L) = \text{Key}$)
or the element is not in the array (not Found)

Equivalence partitioning

Procedure Search (Cont.)

From the specification, you can see two equivalence partitions:

- 1) Inputs where key element is a member of the sequence (Found=true).
- 2) Inputs where the key is not a sequence number (Found=false).

Testing guidelines (when you are testing programs with sequences, arrays or lists):

- . Test software with sequences which have only a single value.
- . Use sequences of different sizes in different tests.
- . Derive tests so that the first, middle and last elements of the sequence are accessed.

From these guidelines, two more equivalence partitions can be identified:

- 3) The input sequence has a single element.
- 4) The number of elements in the input sequence is greater than 1.

Equivalence partitioning

Procedure Search (Cont.)

We then identify further partitions by combining these partitions.

Now the identified partitions for the Search routine are:

Partition #	Sequence	Element
1	Single value	In sequence
2	Single value	Not in sequence
3	More than 1 value	First element in sequence
4	More than 1 value	Last element in sequence
5	More than 1 value	Middle element in sequence
6	More than 1 value	Not in sequence

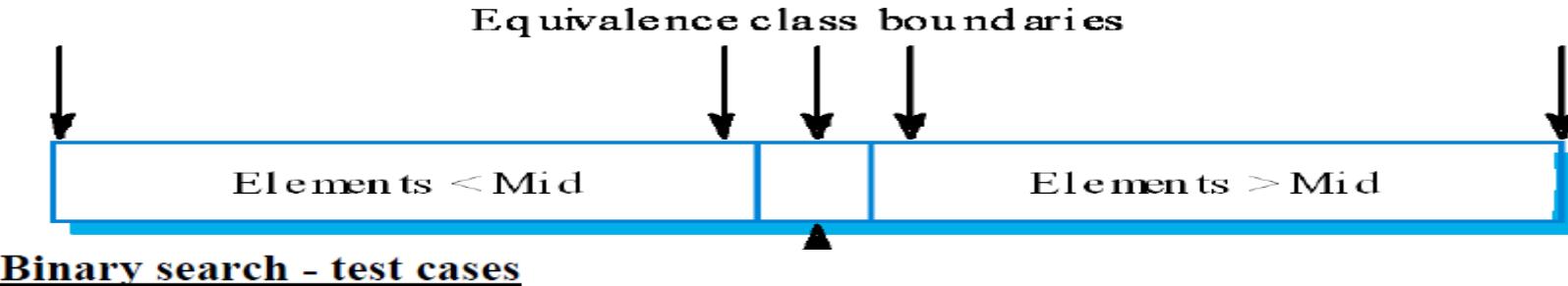
And the set of possible test cases based on these partitions are:

Test case	Input sequence (T)	Key (Key)	Output (Found, L)
1	17	17	true, 1
2	17	0	false, ??
3	17, 29, 21, 23	17	true, 1
4	41, 18, 9, 31, 30, 16, 45	45	true, 7
5	17, 18, 21, 23, 29, 41, 38	23	true, 4
6	21, 23, 29, 33, 38	25	false, ??

Equivalence partitioning

EX: Binary search equiv. partitions

- By examining the code of the binary search routine, we can see that it involves splitting the search space into three parts.
- Each of these parts makes up equivalence partition. We then design test cases where the key lies at the boundaries of each of these partitions.



Test case	Input array (T)	Key (Key)	Output (Found, L)
1	17	17	true, 1
2	17	0	false, ??
3	17, 21, 23, 29	17	true, 1
4	9, 16, 18, 30, 31, 41, 45	45	true, 7
5	17, 18, 21, 23, 29, 38, 41	23	true, 4
6	17, 18, 21, 23, 29, 33, 38	21	true, 3
7	12, 18, 21, 23, 32	23	true, 4
8	21, 23, 29, 33, 38	25	false, ??

Structural testing

- Sometime called white-box testing, glass-box testing or clear-box testing.
- Is an approach to test case design where tests are derived from knowledge of the program's structure and implementation.
- Objective is to exercise all program statements (not all path combinations).

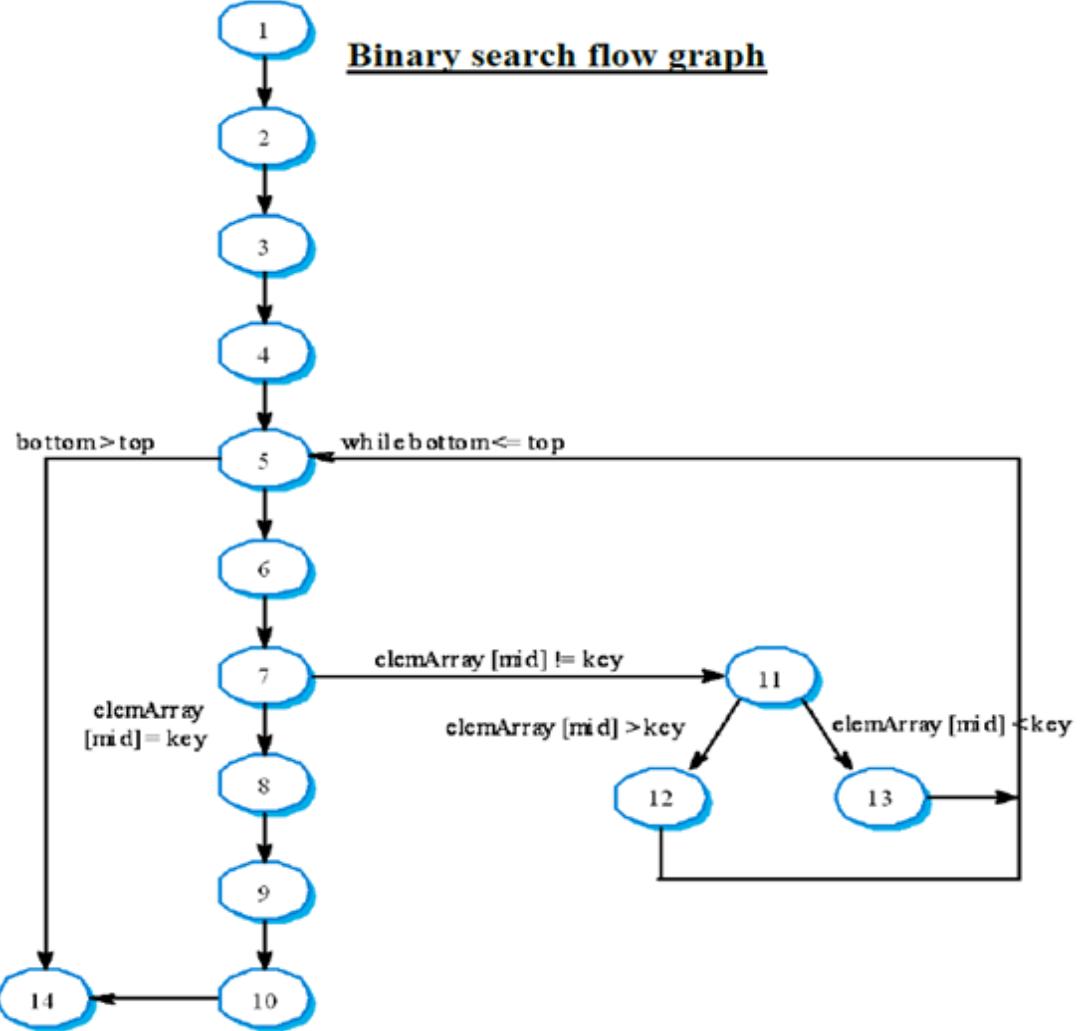
Path testing

- The objective of path testing is to ensure that the set of test cases is such that each independent path through the program is executed at least once.
- Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. This leads to executing all statements in the component at least once. Furthermore, all conditional statements are tested for both true and false cases.
- Mostly used during component testing.
- The starting point for path testing is a program flow graph that can be used to represent the logical flow control and therefore all the execution paths that need testing.
- Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node

Path testing EX: Binary search routine

Void search(int key, int elemArray[], int n_elements , int & index, boolean & found)

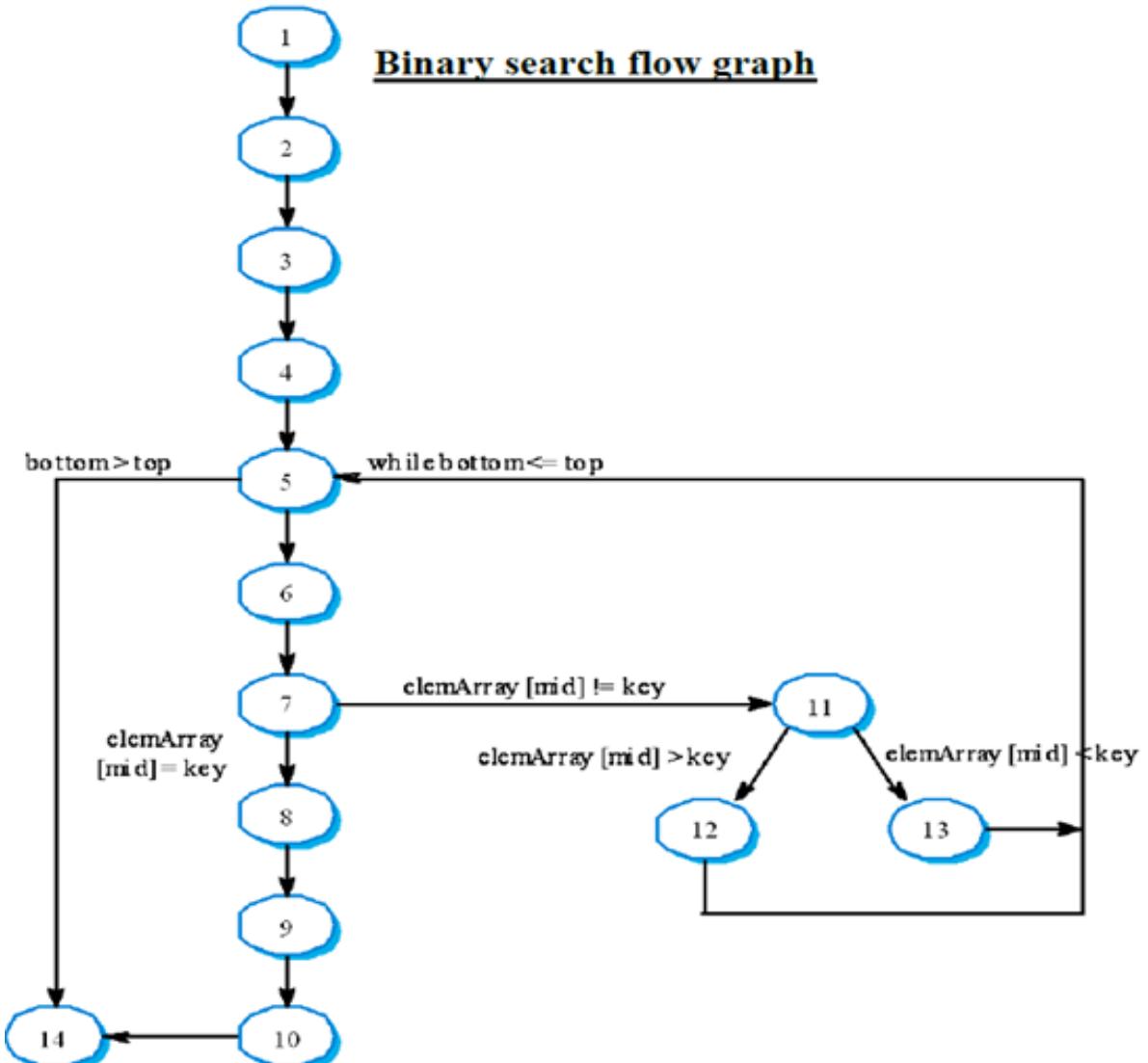
```
{  
    1  int bottom=0;  
    2  int top= n_elements-1;  
  
    3  int mid;  
    4  found = false;  
    5  index=-1;  
    6  while ( bottom <= top)  
    {  
        7      mid=(top+bottom)/2;  
        8      if ( arr[mid]==key)  
        {  
            9          index=mid;  
            10         found=true;  
            11         return;  
        } // if part  
        else  
        {  
  
            11        if ( arr[mid]<key)  
            12            bottom=mid+1;  
            else  
            13                top=mid-1;  
        }  
    } // while loop  
14 } // bin search
```



Path testing EX: Binary search routine

- Independent paths
 - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
 - 1, 2, 3, 4, 5, 14
 - 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
 - 1, 2, 3, 4, 5, 6, 7, 11, 13, 5, ...

- ✧ Test cases should be derived so that all of these paths are executed
- ✧ A dynamic program analyzer may be used to check that paths have been executed



Cyclomatic complexity

- Is a software metric (measurement), used to indicate the complexity of a program. It is quantitative measure of the no. of linearly independent paths through a program's source code. So we can find the number of independent paths in a program by computing the cyclomatic complexity of the program flow graph.
- The easiest way to calculate CC is to sum the number binary decision statements (e.g if, while, for, etc.) + 1.
- Other way can represented using this formula: $(CC=E-N+2*P)$ where E: No. of edges, N: No. of nodes in the flow graph , P: No. of nodes that have exit points.
- If the program includes compound conditions, then you count the number of simple conditions in the compound conditions, when calculating the cyclomatic complexity.

Cyclomatic complexity

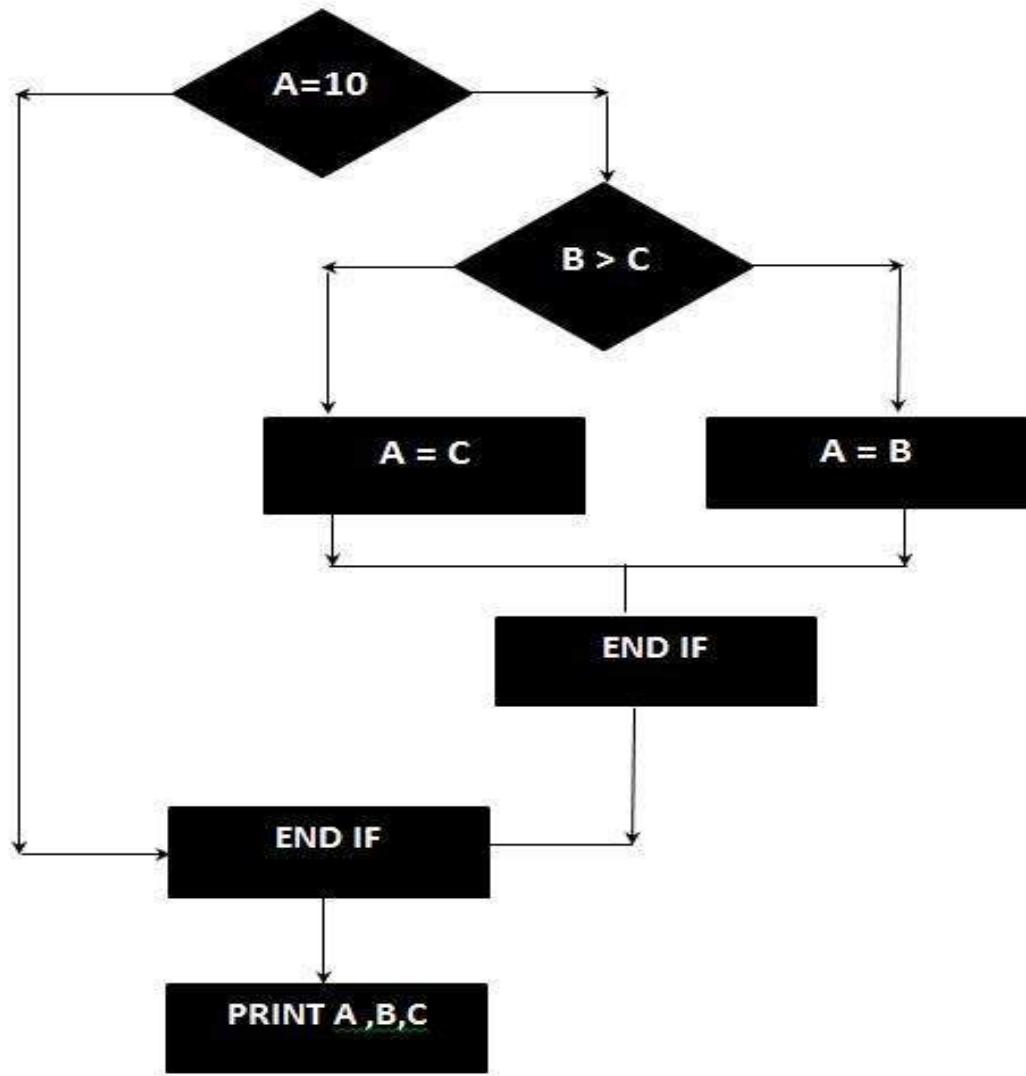
- Therefore, if there are six if-statements and a while loop and all conditional expressions are simple, the cyclomatic complexity is 8.
- If one conditional expression is a compound expression such as “if A and B or C”, then you count this as three simple conditions. The cyclomatic complexity is therefore = 10.
- The cyclomatic complexity of the binary search algorithm is 4 because there are three simple conditions at lines 5, 7 and 11.
- The cyclomatic complexity is software metric that offers an indication of the logical complexity of a program.

Cyclomatic complexity

```
IF A = 10 THEN  
  IF B > C THEN  
    A = B  
  ELSE  
    A = C  
  ENDIF  
ENDIF  
Print A  
Print B  
Print C
```

$$\text{The CC} = 8 - 7 + 2 = 3$$

or =2+1 =3



Loop Testing

- Loops are the basis of most algorithms implemented using software. However, often we do consider them when conducting testing. Loop testing is a white box testing approach that concentrates on the validity of loop constructs. three loops can be defined: simple loops, concatenate loops, and nested loops.
 - **Simple loops:** The following group of tests should be used on simple loops, where n is the maximum number of allowable passes through the loop:
 - Skip the loop entirely, Only one pass through the loop, Two passes through the loop, M passes through the loop where $m < n$., $n-1$, n , $n+1$ passes through the loop.
 - **Nested loop:** For the nested loop the number of possible tests increases as the level of nesting grows. This would result in an impractical number of tests.
 - **Concatenated loops:** Concatenated loops can be tested using the techniques outlined for simple loops, if each of the loops is independent of the other. When the loops are not independent the approach applied to nested loops is recommended.

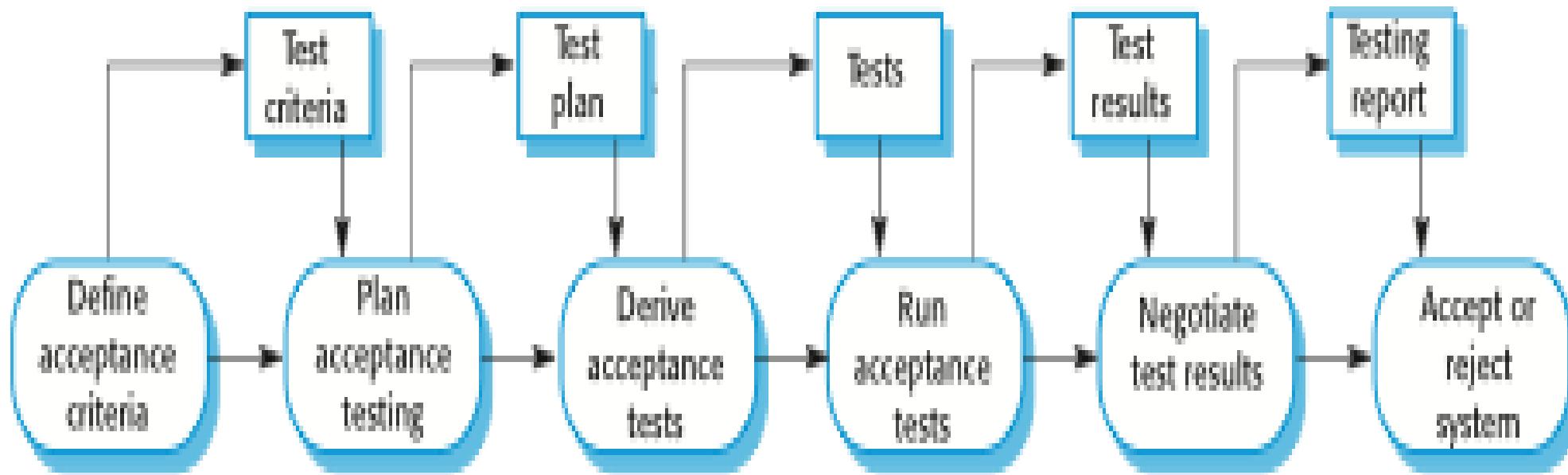
User testing

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
 - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

- Alpha testing
 - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
 - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- Acceptance testing
 - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

The acceptance testing process



Stages in the acceptance testing process

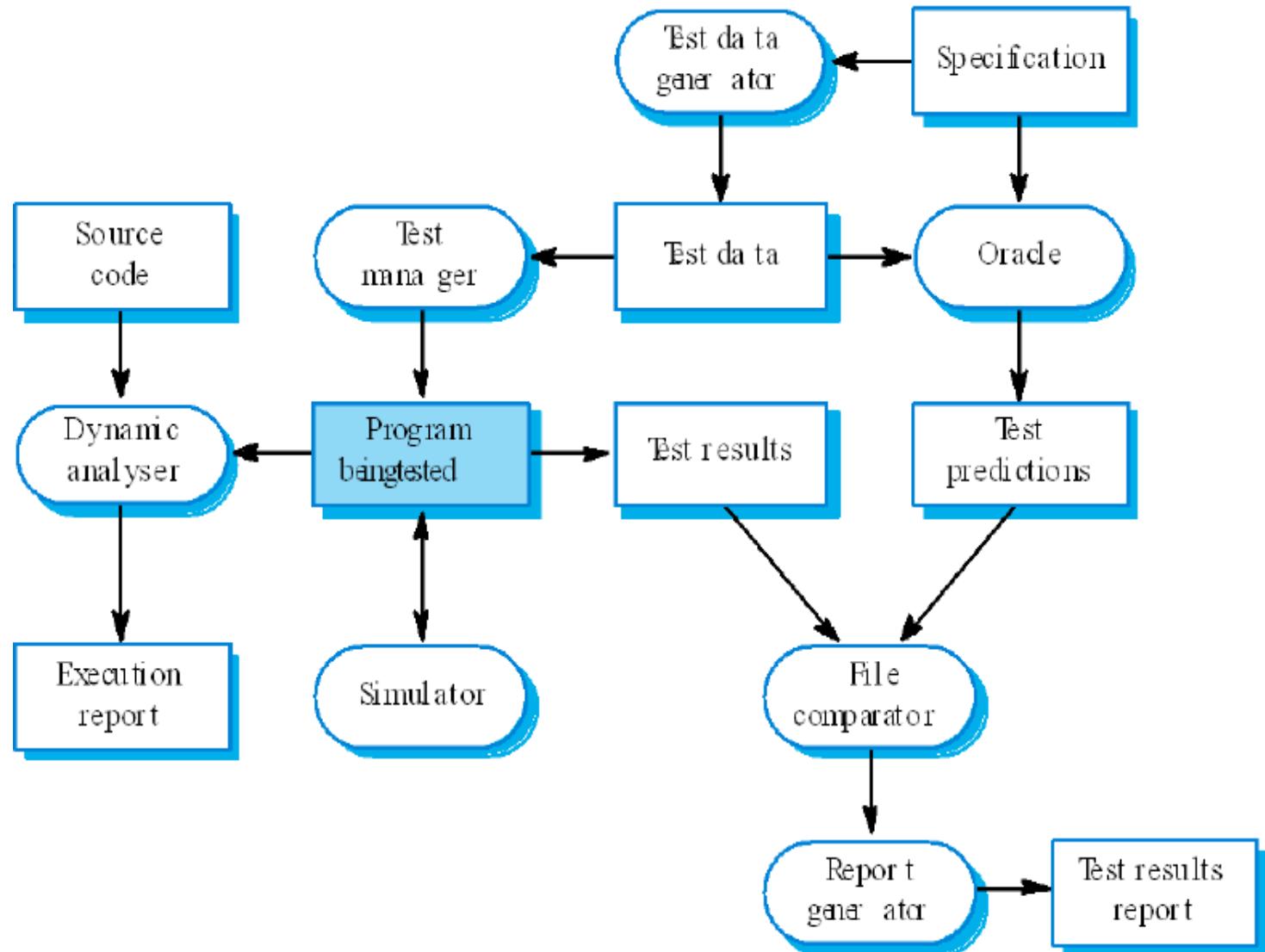
- Define acceptance criteria
- Plan acceptance testing
- Derive acceptance tests
- Run acceptance tests
- Negotiate test results
- Reject/accept system

Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is ‘typical’ and can represent the interests of all system stakeholders.

Testing workbench

- **Test Managers**
- **Test data generators**
- **Test oracles**
- **File comparator**
- **Dynamic analyzer**
- **Simulators**



Testing workbench

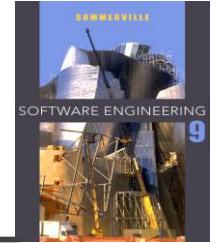
1. *Test manager* Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.
2. *Test data generator* Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. *Oracle* Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-to-back testing (discussed in Chapter 17) involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.

Testing workbench

4. *File comparator* Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.
5. *Report generator* Provides report definition and generation facilities for test results.
6. *Dynamic analyser* Adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.
7. *Simulator* Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulators are script-driven programs that simulate multiple simultaneous user interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.

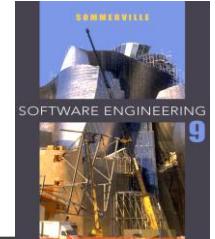
Key points

- When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development where tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.



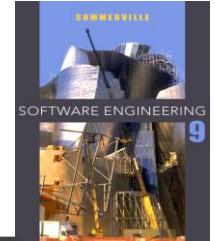
Chapter 3– Software Evolution

Lecture 5



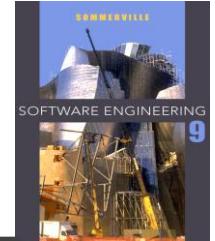
Topics covered

- ✧ Evolution processes
 - Change processes for software systems
- ✧ Program evolution dynamics
 - Understanding software evolution
- ✧ Software maintenance
 - Making changes to operational software systems
- ✧ Legacy system management
 - Making decisions about software change



Software change

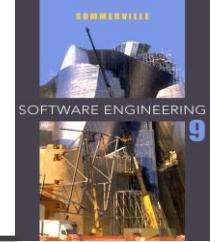
- ✧ Software change is inevitable (certain) when
 - New requirements emerge (appeared) when the software is used;
 - The business environment changes;
 - Errors must be repaired;
 - New computers and equipment is added to the system;
 - The performance or reliability of the system may have to be improved.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



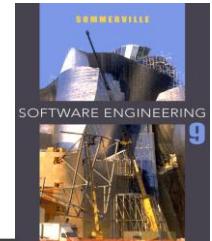
Change is inevitable

- ✧ The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- ✧ Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- ✧ Systems MUST be changed if they are to remain useful in an environment.

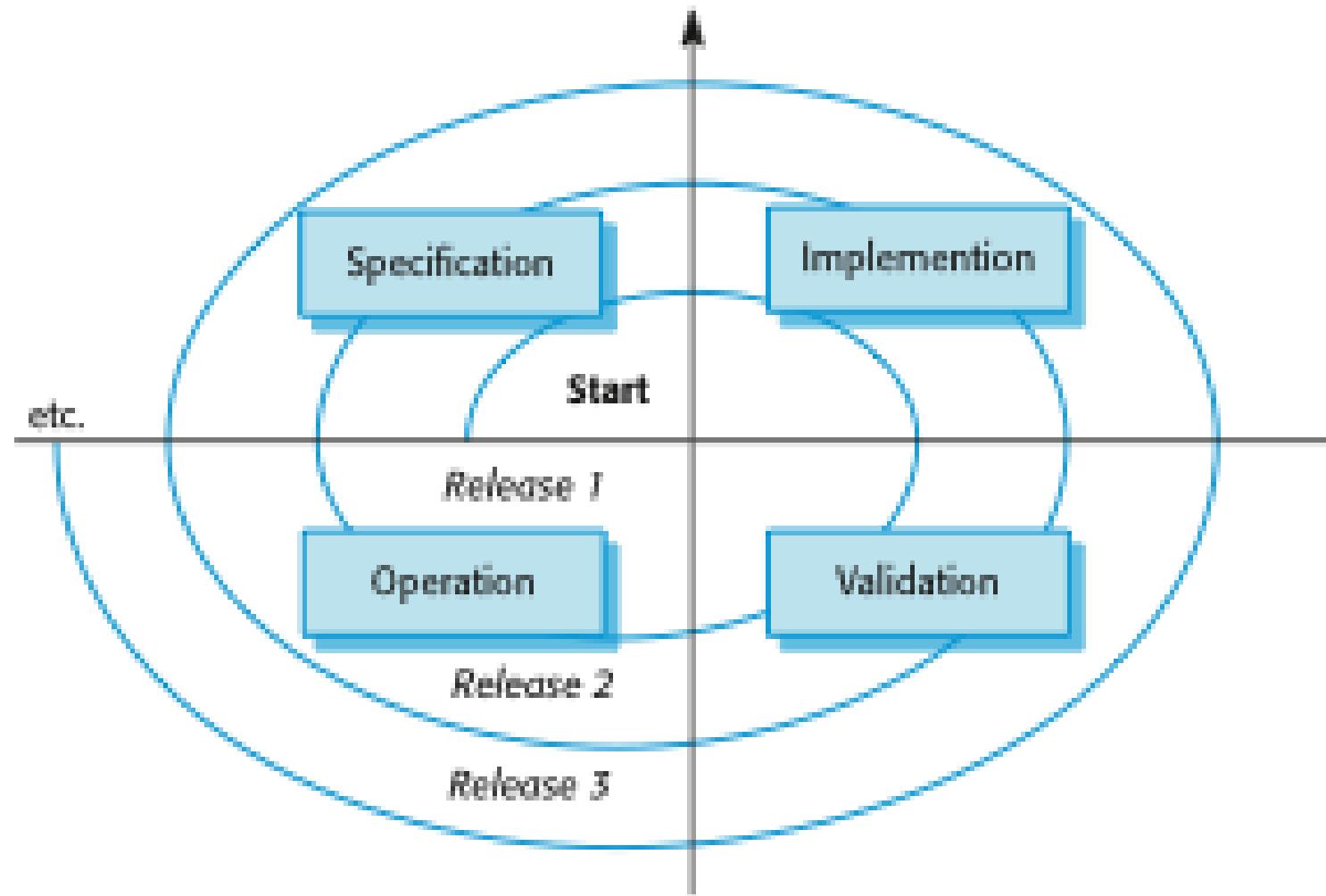
Importance of evolution



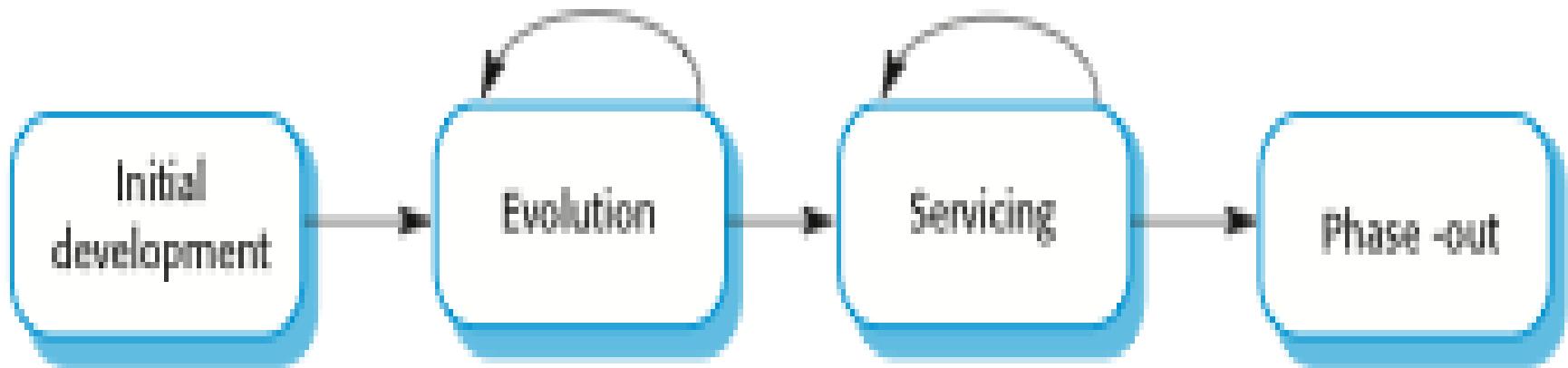
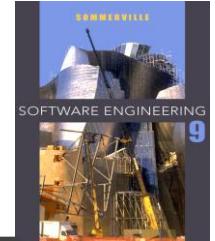
- ✧ Organizations have huge investments in their software systems - they are critical business assets.
- ✧ To maintain the value of these assets to the business, they must be changed and updated.
- ✧ The majority of the software budget in large companies is devoted to changing and evolving existing software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

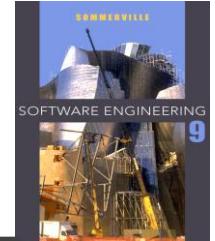


A spiral model of development and evolution



Evolution and servicing





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.

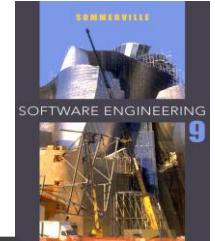
❖ Servicing

- 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.

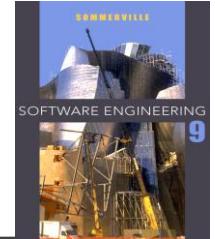
❖ Phase-out

- The software may still be used but no further changes are made to it.

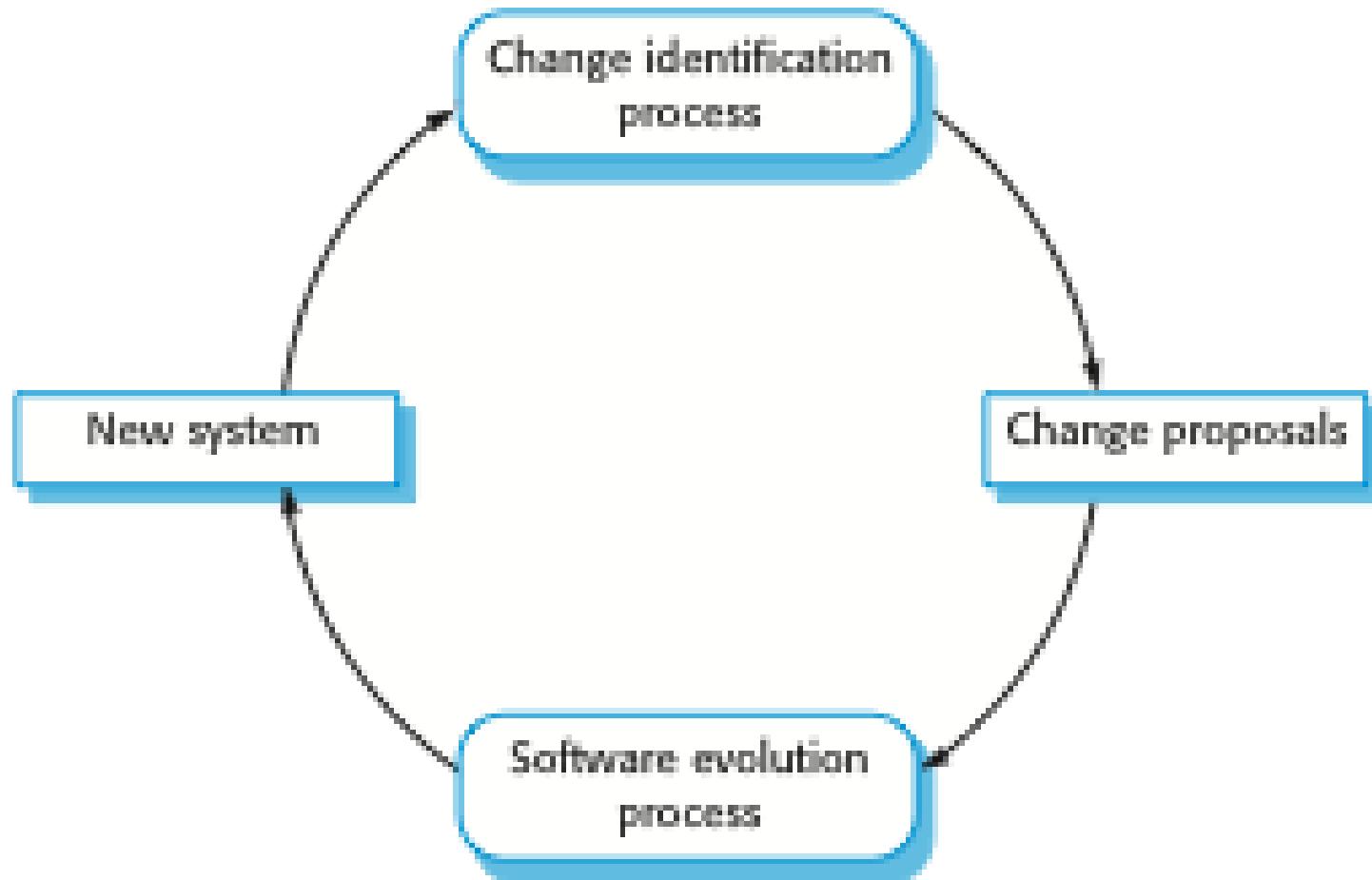
Evolution processes

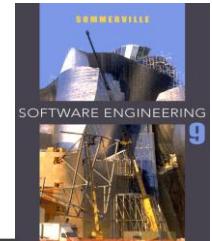


- ✧ Software evolution processes depend on
 - The type of software being maintained;
 - The development processes used;
 - The skills and experience of the people involved.
- ✧ Proposals for change are the driver for system evolution.
 - Should be linked with components that are affected by the change,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

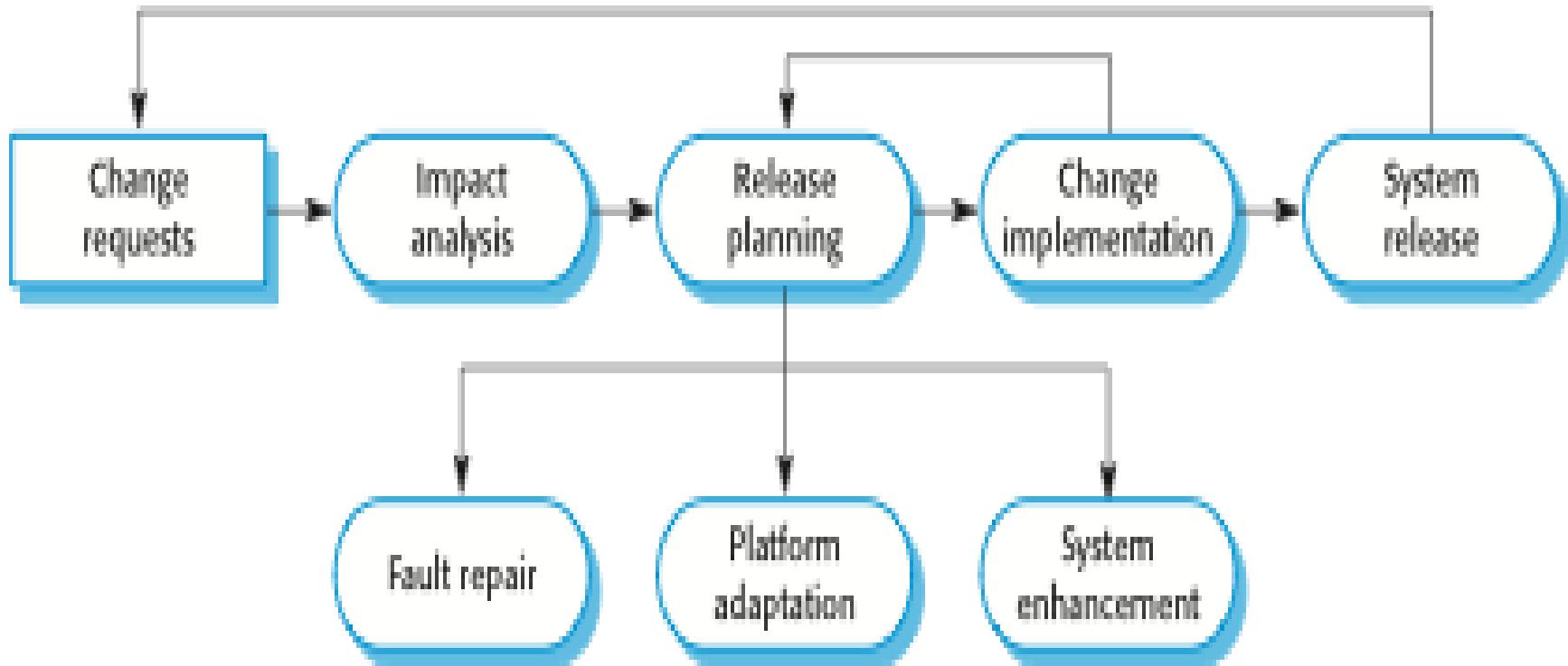


Change identification and evolution processes

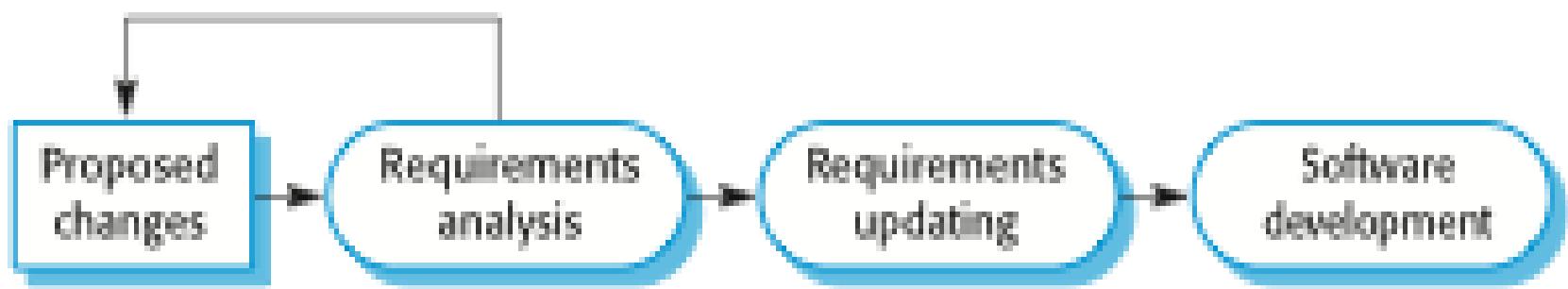
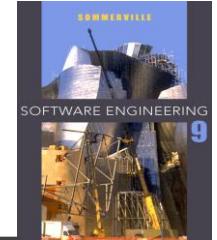


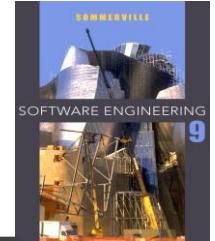


The software evolution process



Change implementation

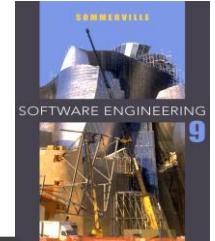




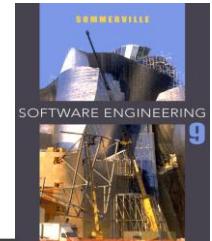
Change implementation

- ✧ Iteration of the development process where the revisions to the system are designed, implemented and tested.
- ✧ A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- ✧ During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

Urgent change requests



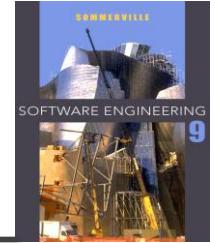
- ✧ Urgent changes may have to be implemented without going through all stages of the software engineering process
 - If a serious system fault has to be repaired to allow normal operation to continue;
 - If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
 - If there are business changes that require a very rapid response (e.g. the release of a competing product).



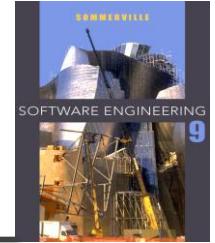
The emergency repair process



Agile methods and evolution



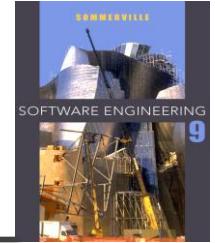
- ❖ Agile methods are based on incremental development so the transition from development to evolution is a seamless (smooth) one.
 - Evolution is simply a continuation of the development process based on frequent system releases.
- ❖ Automated regression testing is particularly valuable when changes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



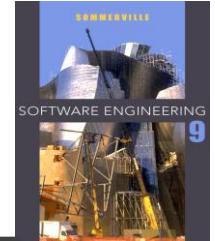
Handover problems

- ✧ 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.

Program evolution dynamics

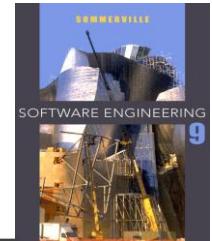


- ✧ Program evolution dynamics is the study of the processes of system change.
- ✧ After several major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved.
- ✧ There are sensible observations rather than laws. They are applicable to large systems developed by large organisations.
 - It is not clear if these are applicable to other types of software system.



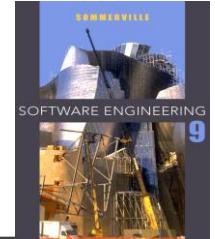
Lehman's laws

Law	Description
Continuing change	A <u>program</u> that is <u>used in a real-world environment</u> must <u>necessarily change</u> , or else become progressively less useful in that environment.
Increasing complexity	As an <u>evolving program changes</u> , its <u>structure</u> tends to <u>become more complex</u> . Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.



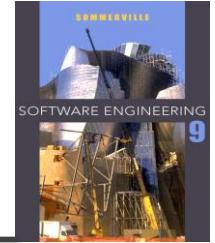
Lehman's laws

Law	Description
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multiage, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.



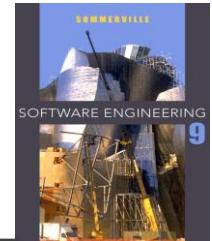
Applicability of Lehman's laws

- ✧ Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
 - Confirmed in early 2000's by work by Lehman on the FEAST project.
- ✧ It is not clear how they should be modified for
 - Shrink-wrapped software products;
 - Systems that incorporate a significant number of COTS (**Commercial off-the-shelf**) components;
 - Small organisations;
 - Medium sized systems.



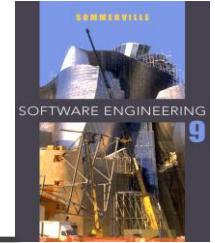
Key points

- ✧ 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.
- ✧ Lehman's laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.



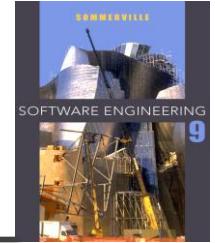
Chapter 3 – Software Evolution

Lecture 6



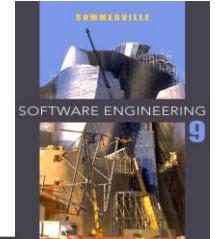
Software maintenance

- ✧ Modifying a program after it has been put into use.
- ✧ The term is mostly used for changing custom software.
- ✧ Generic software products are said to evolve to create new versions.
- ✧ Maintenance does not normally involve major changes to the system's architecture.
- ✧ Changes are implemented by modifying existing components and adding new components to the system.

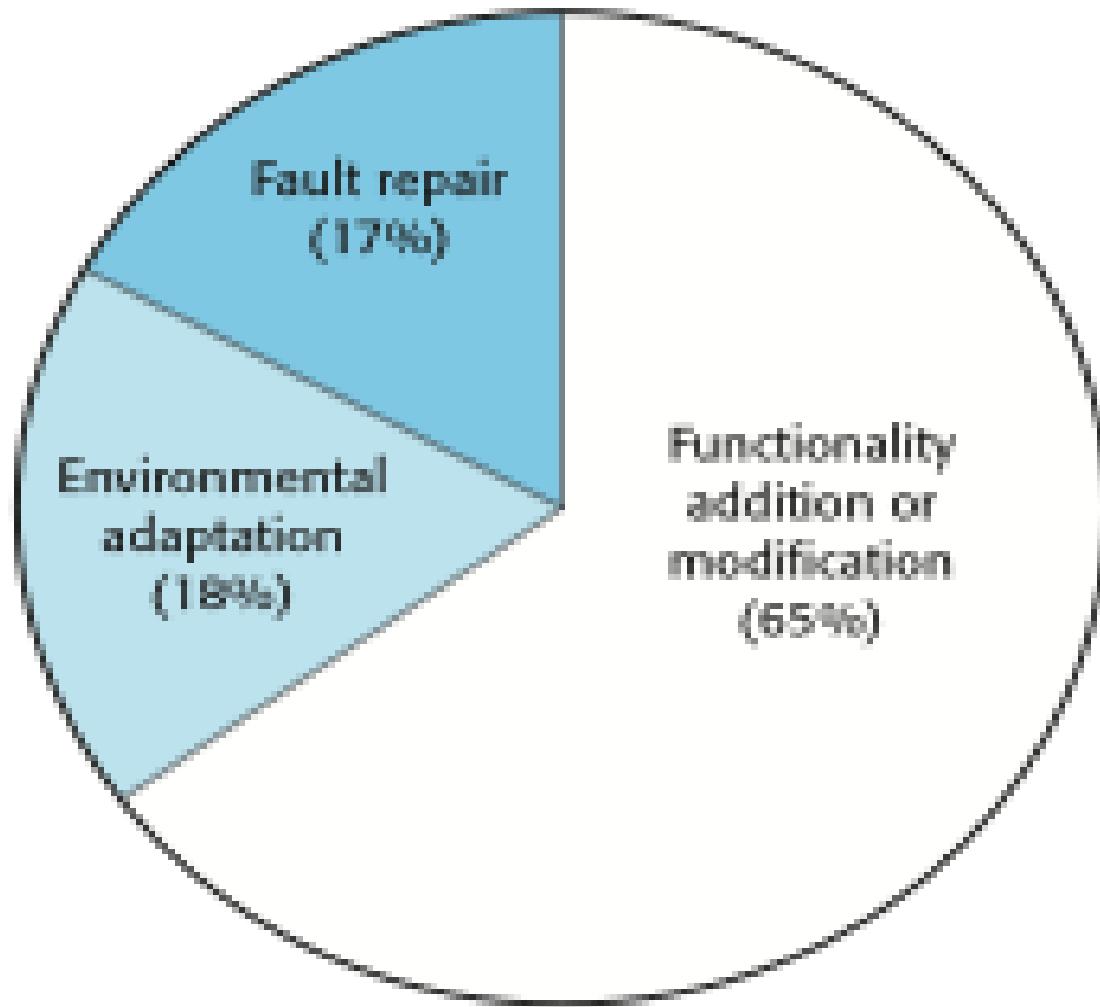


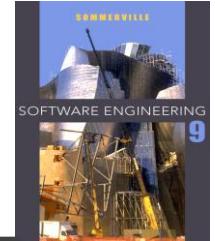
Types of maintenance

- ✧ Maintenance to repair software faults
 - Changing a system to correct deficiencies in the way it meets its requirements.
- ✧ 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.
- ✧ Maintenance to add to or modify the system's functionality
 - Modifying the system to satisfy new requirements.



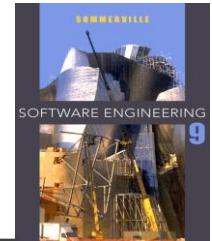
Maintenance effort distribution



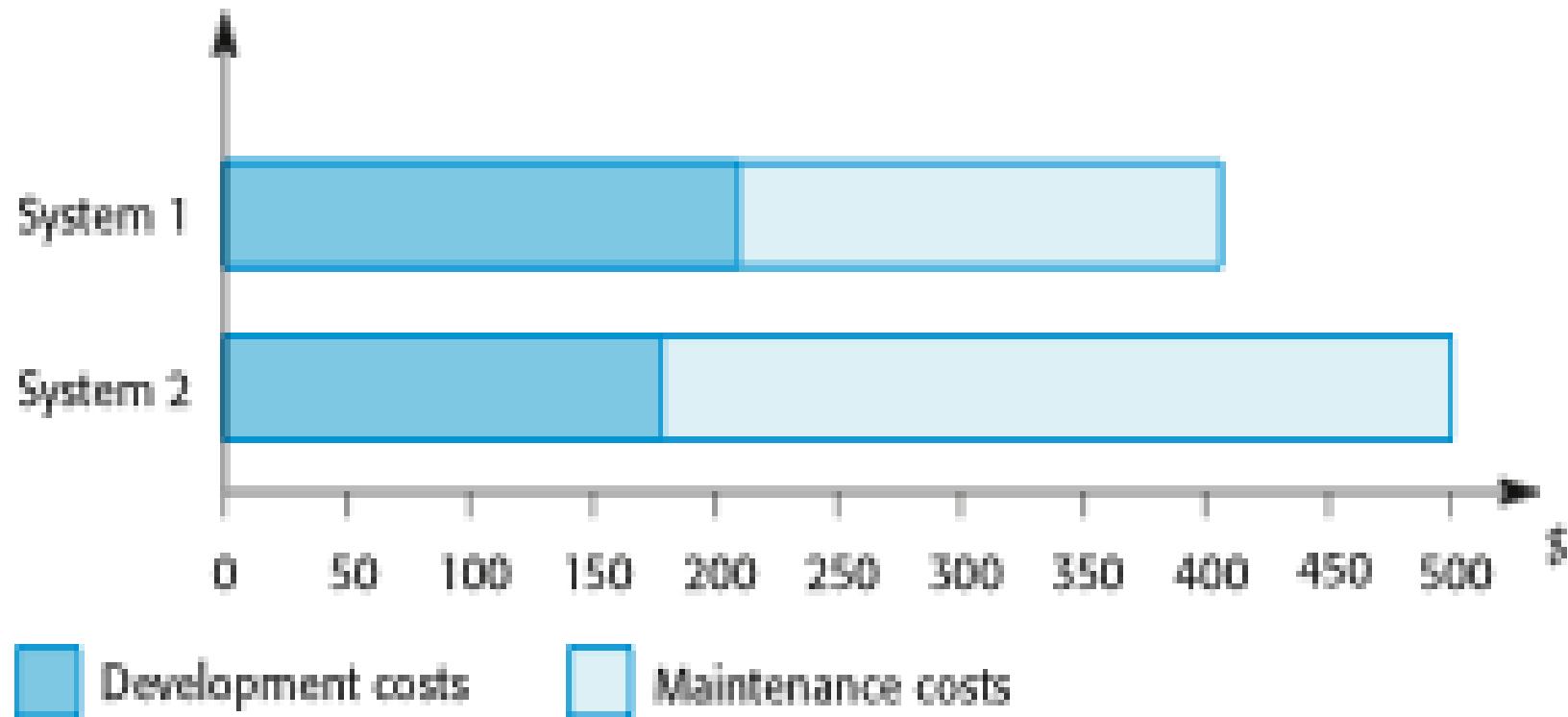


Maintenance costs

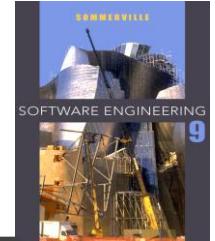
- ❖ Usually greater than development costs (2* to 100* depending on the application).
- ❖ Affected by both technical and non-technical factors.
- ❖ Increases as software is maintained.
Maintenance corrupts the software structure so makes further maintenance more difficult.
- ❖ Ageing software can have high support costs (e.g. old languages, compilers etc.).



Development and maintenance costs



Maintenance cost factors



✧ Team stability

- Maintenance costs are reduced if the same staff are involved with them for some time.

✧ Contractual responsibility

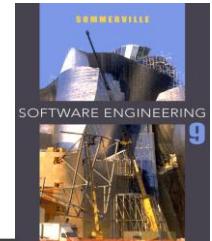
- The developers of a system may have no contractual responsibility for maintenance so there is no motivation to design for future change.

✧ Staff skills

- Maintenance staff are often inexperienced and have limited domain knowledge.

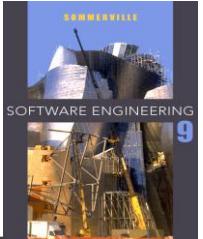
✧ Program age and structure

- As programs age, their structure is degraded and they become harder to understand and change.



Maintenance prediction

- ✧ Maintenance prediction is concerned with assessing (measuring) 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

What parts of the system are most likely to be affected by change requests?

Predicting maintainability

What parts of the system will be the most expensive to maintain?

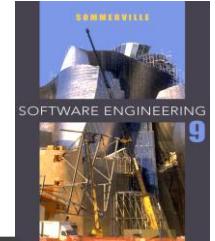
Predicting system changes

Predicting maintenance costs

What will be the lifetime maintenance costs of this system?

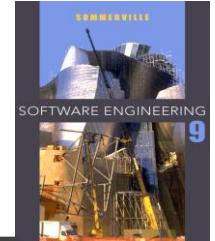
How many change requests can be expected?

What will be the costs of maintaining this system over the next year?



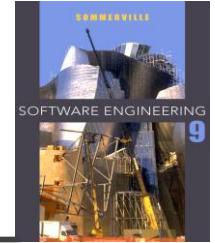
Change prediction

- ✧ 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

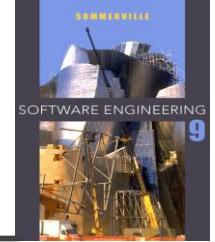
- ✧ Predictions of maintainability can be made by assessing the complexity of system components.
- ✧ Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- ✧ Complexity depends on
 - Complexity of control structures;
 - Complexity of data structures;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



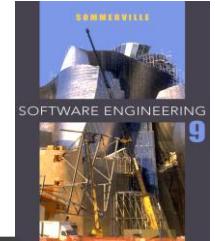
Process metrics

- ✧ Process metrics may be used to assess (evaluate) 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.

System re-engineering



- ✧ Re-structuring or re-writing 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.
- ✧ Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.



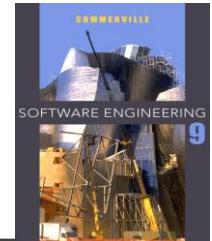
Advantages of re-engineering

✧ Reduced risk

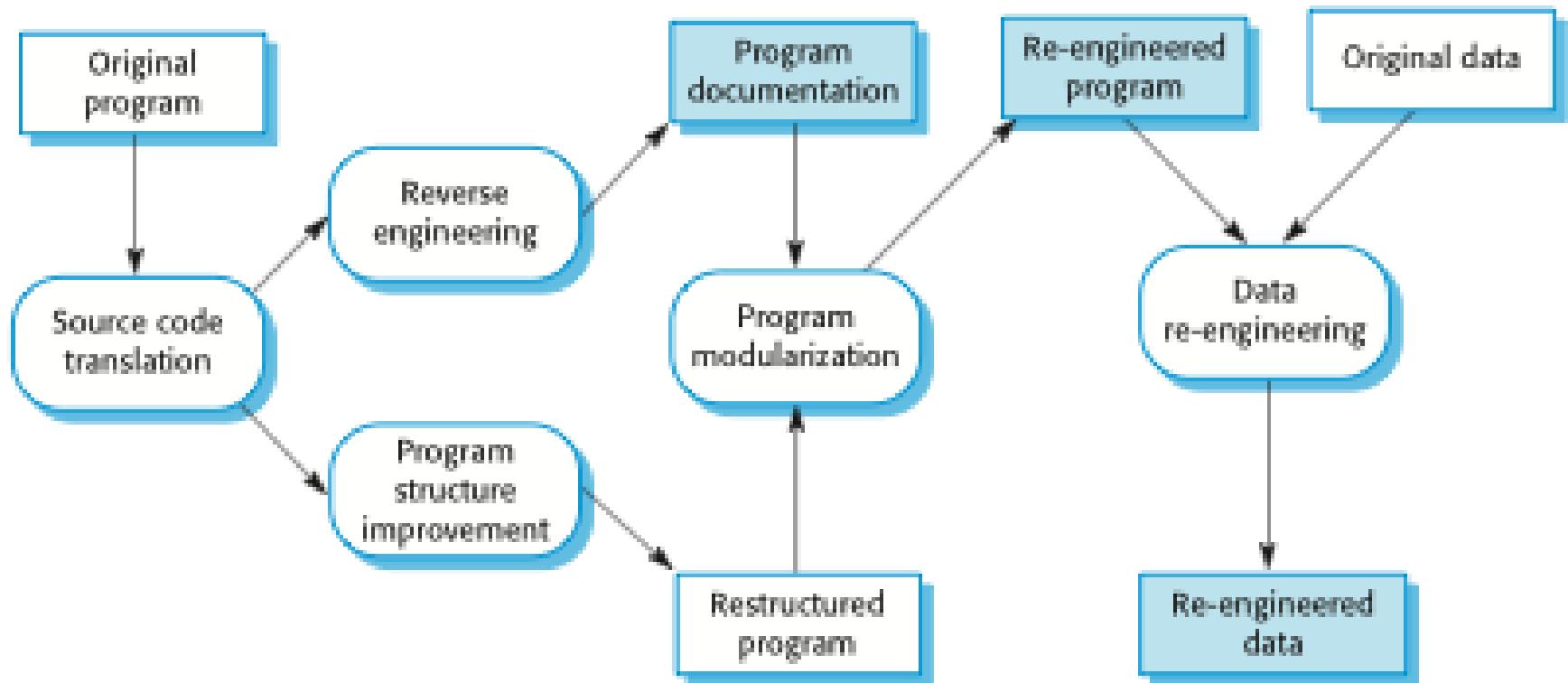
- There is a high risk in new software development. There may be development problems, staffing problems and specification problems.

✧ Reduced cost

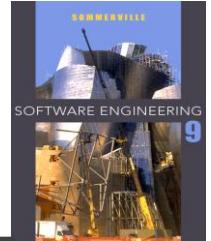
- The cost of re-engineering is often significantly less than the costs of developing new software.



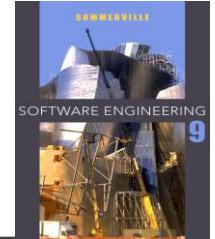
The re-engineering process



Re-engineering process activities



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



Reengineering approaches

Automated program
restructuring

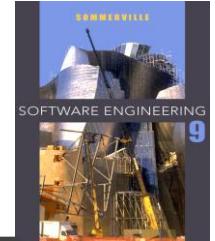
Program and data
restructuring

Automated source
code conversion

Automated restructuring
with manual changes

Restructuring plus
architectural changes

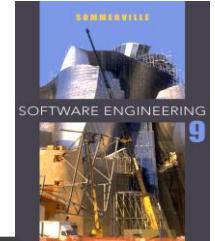
Increased cost



Reengineering cost factors

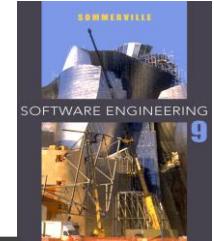
- ✧ The quality of the software to be reengineered.
- ✧ The tool support available for reengineering.
- ✧ The extent of the data conversion which is required.
- ✧ The availability of expert staff for reengineering.
 - This can be a problem with old systems based on technology that is no longer widely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

Preventative maintenance by refactoring

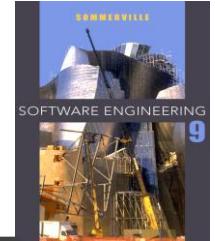


- ✧ Refactoring is the process of making improvements to a program to slow down degradation through change.
- ✧ You can think of refactoring as 'preventative maintenance' that reduces the problems of future change.
- ✧ Refactoring involves modifying a program to improve its structure, reduce its complexity 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..
- ✧ 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

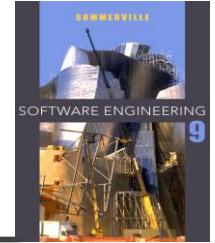
- If a method is too long, it should be redesigned as a number of shorter methods.

✧ 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.

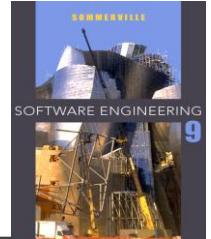
✧ Data clumping

- 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.



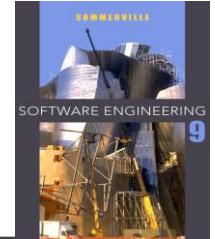
Legacy system management

- ✧ Organisations that rely on legacy systems must choose a strategy for evolving these systems
 - Scrap the system completely and modify business processes so that it is no longer required;
 - Continue maintaining the system;
 - Transform the system by re-engineering to improve its maintainability;
 - Replace the system with a new system.
- ✧ The strategy chosen should depend on the system quality and its business value.



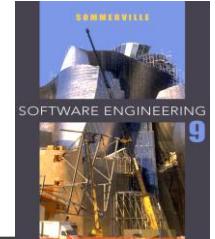
Legacy system categories

- ✧ Low - quality, low - business value
 - These systems should be scrapped.
- ✧ Low - quality, high - business value
 - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- ✧ High - quality, low - business value
 - Replace with COTS (Commercial off the shelf), scrap completely or maintain.
- ✧ High - quality, high - business value
 - Continue in operation using normal system maintenance.



System quality assessment

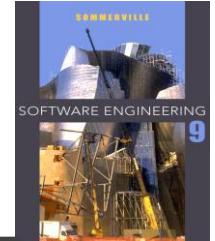
- ✧ Business process assessment
 - How well does the business process support the current goals of the business?
- ✧ Environment assessment
 - How effective is the system's environment and how expensive is it to maintain?
- ✧ Application assessment
 - What is the quality of the application software system?



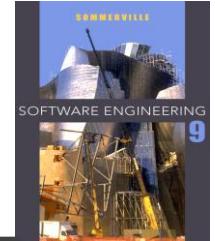
Factors used in environment assessment

Factor	Questions
Supplier stability	Is the supplier <u>still in existence</u> ? Is the <u>supplier financially stable</u> and likely to continue in existence? <u>If the supplier is no longer in business, does someone else maintain the systems?</u>
Failure rate	<u>Does the hardware have a high rate of reported failures?</u> Does the support software crash and force system restarts?
Age	<u>How old is the hardware and software?</u> 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

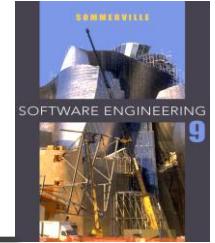


Factor	Questions
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licenses? Older hardware 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?



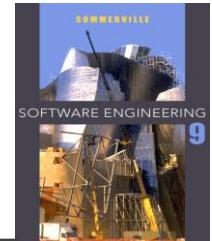
Factors used in application assessment

Factor	Questions
Understandability	<u>How difficult</u> is it to <u>understand the source code</u> of the current system? <u>How complex</u> are <u>the control structures</u> that are used? <u>Do variables have meaningful names</u> that reflect their function?
Documentation	What system documentation is available? Is the documentation <u>complete</u> , <u>consistent</u> , and <u>current</u> ?
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	<u>Is</u> the performance of the application <u>adequate</u> ? Do performance problems have a <u>significant</u> effect on <u>system users</u> ?



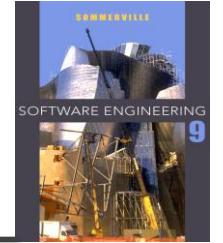
Factors used in application assessment

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



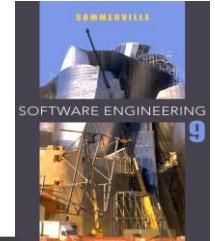
Key points

- ✧ There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- ✧ 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.
- ✧ 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.



Chapter 24 - Quality Management

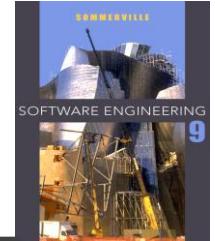
Lecture 7



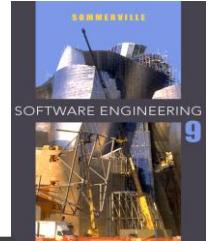
Topics covered

- ✧ Software quality
- ✧ Software standards
- ✧ Reviews and inspections
- ✧ Software measurement and metrics

Software quality management



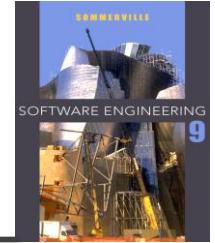
- ✧ Concerned with ensuring that the required level of quality is achieved in a software product.
- ✧ Three principal concerns:
 - At the **organizational level**, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software.
 - At the **project level**, quality management involves defining appropriate quality standards or processes and checking that these planned processes have been followed.
 - At the **project level**, quality management is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.



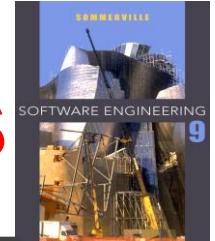
What is quality?

- ✧ Quality means that a product should meet its specification which there are:
 - Customer quality requirements (efficiency, reliability, etc.) and
 - Developer quality requirements (maintainability, reusability, etc.);
- ✧ Some quality requirements are difficult to specify in an unambiguous (definite) way;
- ✧ Software specifications are usually incomplete and often inconsistent.

Quality management activities



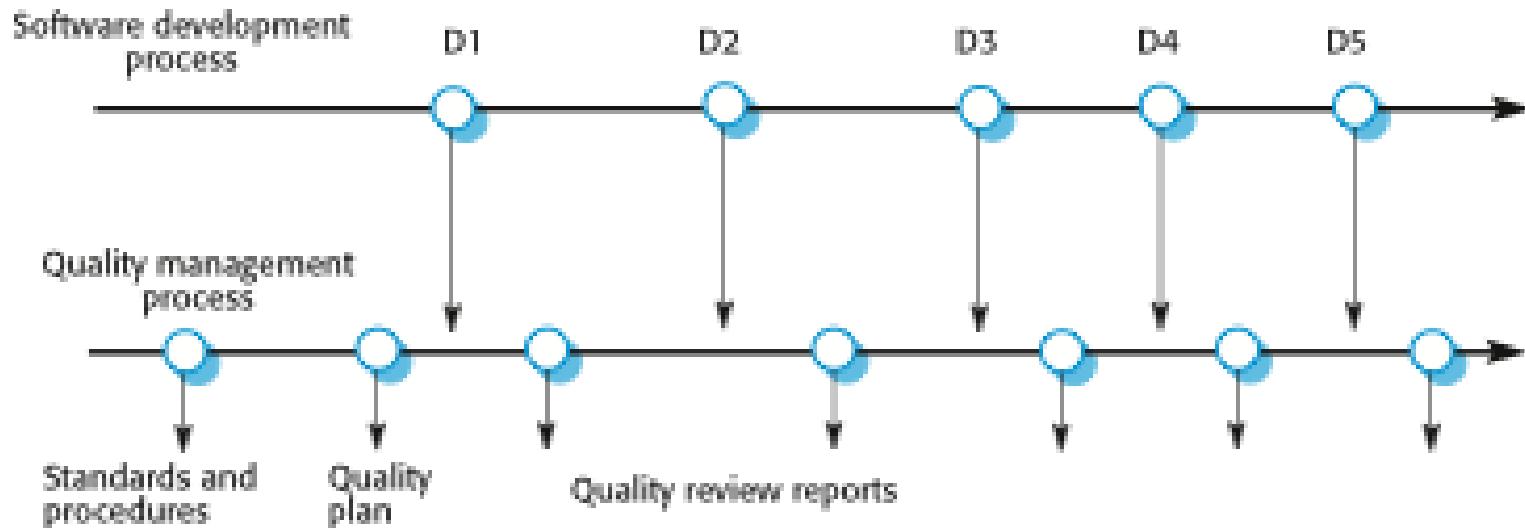
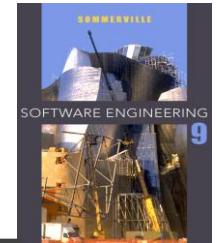
- ❖ Structured into three main activities.
 - Quality assurance: Establish organizational procedures and standards for quality.
 - Quality planning: Select applicable procedures and standards for a particular project and modify these as required.
 - Quality control: Ensure that procedures and standards are followed by the software development team.
- ❖ Quality management should be separate from project management to ensure independence.

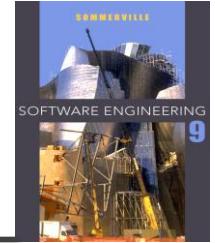


Quality management activities

- ✧ Quality management provides an independent check on the software development process.
- ✧ The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals
- ✧ The quality team should be independent from the development team so that they can take an objective view of the software. This allows them to report on software quality without being influenced by software development issues.

Quality management and software development

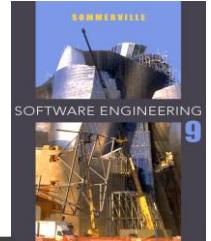




Quality planning

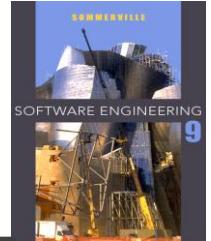
❖ A quality plan

- Sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- Define the quality assessment process.
- Set out which organisational standards should be applied and, where necessary, define new standards to be used.



Quality plans

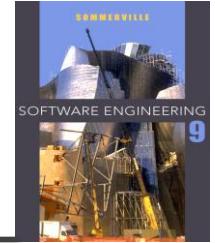
- ✧ Quality plan structure
 - Product introduction;
 - Product plans;
 - Process descriptions;
 - Quality goals;
 - Risks and risk management.
- ✧ Quality plans should be short documents
 - If they are too long, no-one will read them.



Software quality

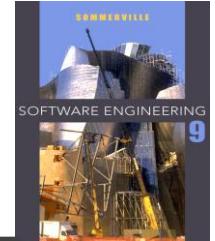
- ✧ Quality, simply, means that a product should meet its specification.
- ✧ This is uncertain for software systems
 - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
 - Some quality requirements are difficult to specify in an unambiguous (clear) way;
 - Software specifications are usually incomplete and often inconsistent.
- ✧ The focus may be 'fitness for purpose' rather than specification conformance.

Software fitness for purpose



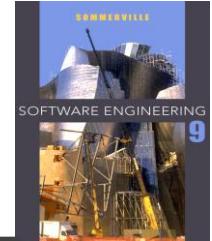
- ❖ Have programming and documentation standards been followed in the development process?
- ❖ Has the software been properly tested?
- ❖ Is the software sufficiently dependable to be put into use?
- ❖ Is the performance of the software acceptable for normal use?
- ❖ Is the software usable?
- ❖ Is the software well-structured and understandable?

Software quality attributes

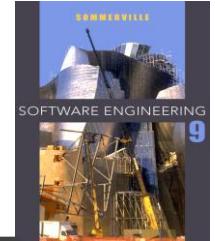


<u>Safety</u>	<u>Understandability</u>	<u>Portability</u>
<u>Security</u>	<u>Testability</u>	<u>Usability</u>
<u>Reliability</u>	<u>Adaptability</u>	<u>Reusability</u>
<u>Resilience</u>	<u>Modularity</u>	<u>Efficiency</u>
<u>Robustness</u>	<u>Complexity</u>	<u>Learnability</u>

Quality conflicts

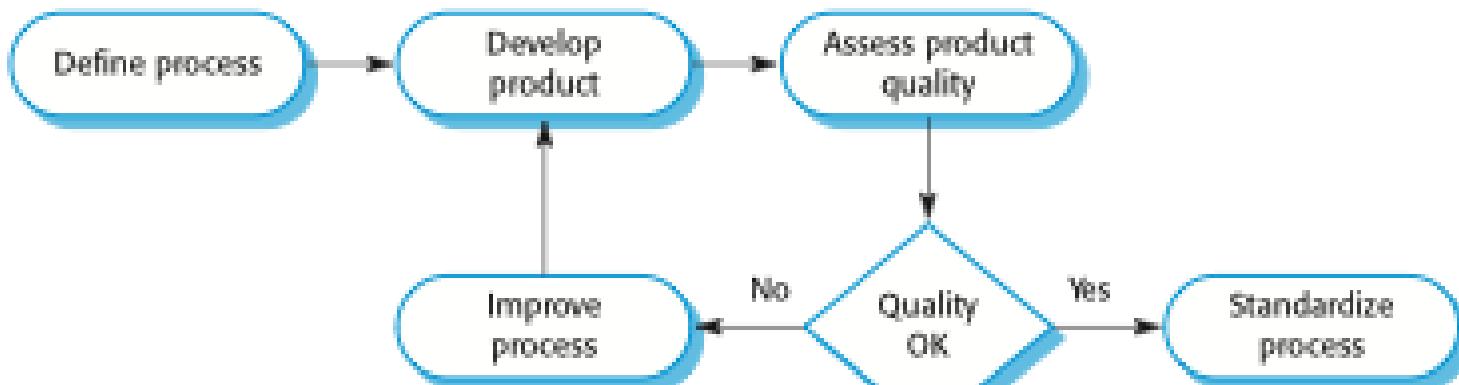


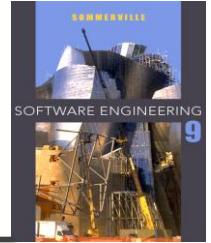
- ❖ It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance.
- ❖ The quality plan should therefore define the most important quality attributes for the software that is being developed.
- ❖ The plan should also include a definition of the quality assessment process, an agreed way of assessing whether some quality, such as Maintainability or robustness, is present in the product.



Process and product quality

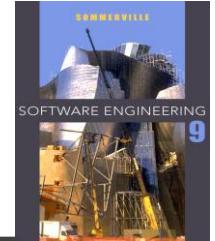
- ✧ The quality of a developed product is influenced by the quality of the production process.
- ✧ This is important in software development as some product quality attributes are hard to assess.
- ✧ However, there is a very complex and poorly understood relationship between software processes and product quality.





Software standards

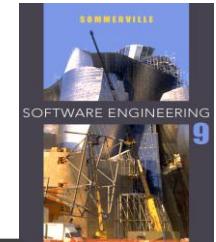
- ❖ Standards define the required attributes of a product or process. They play an important role in quality management.
- ❖ Standards may be international, national, organizational or project standards.
- ❖ Product standards define characteristics that all software components should exhibit e.g. a common programming style.
- ❖ Process standards define how the software process should be enacted (put into practice).



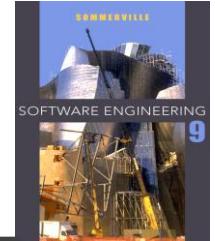
Importance of standards

- ✧ Encapsulation of best practice- avoids repetition of past mistakes.
- ✧ They are a framework for defining what quality means in a particular setting i.e. that organization's view of quality.
- ✧ They provide continuity - new staff can understand the organisation by understanding the standards that are used.

Product and process standards

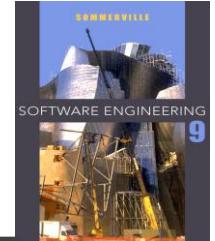


Product standards	Process standards
Design review form	Design review conduct (manner)
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process



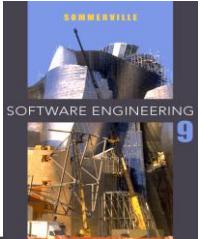
Problems with standards

- ✧ They may not be seen as relevant (related) and up-to-date by software engineers.
- ✧ They often involve too much bureaucratic form filling.
- ✧ If they are unsupported by software tools, tedious form filling work is often involved to maintain the documentation associated with the standards.



ISO 9001 standards framework

- ❖ An international set of standards that can be used as a basis for developing quality management systems.
- ❖ ISO 9001, the most general of these standards, applies to organizations that design, develop and maintain products, including software.
- ❖ The ISO 9001 standard is a framework for developing software standards.
 - It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.



ISO 9001 core processes

Product delivery processes

Business acquisition

Design and development

Test

Production and delivery

Service and support

Supporting processes

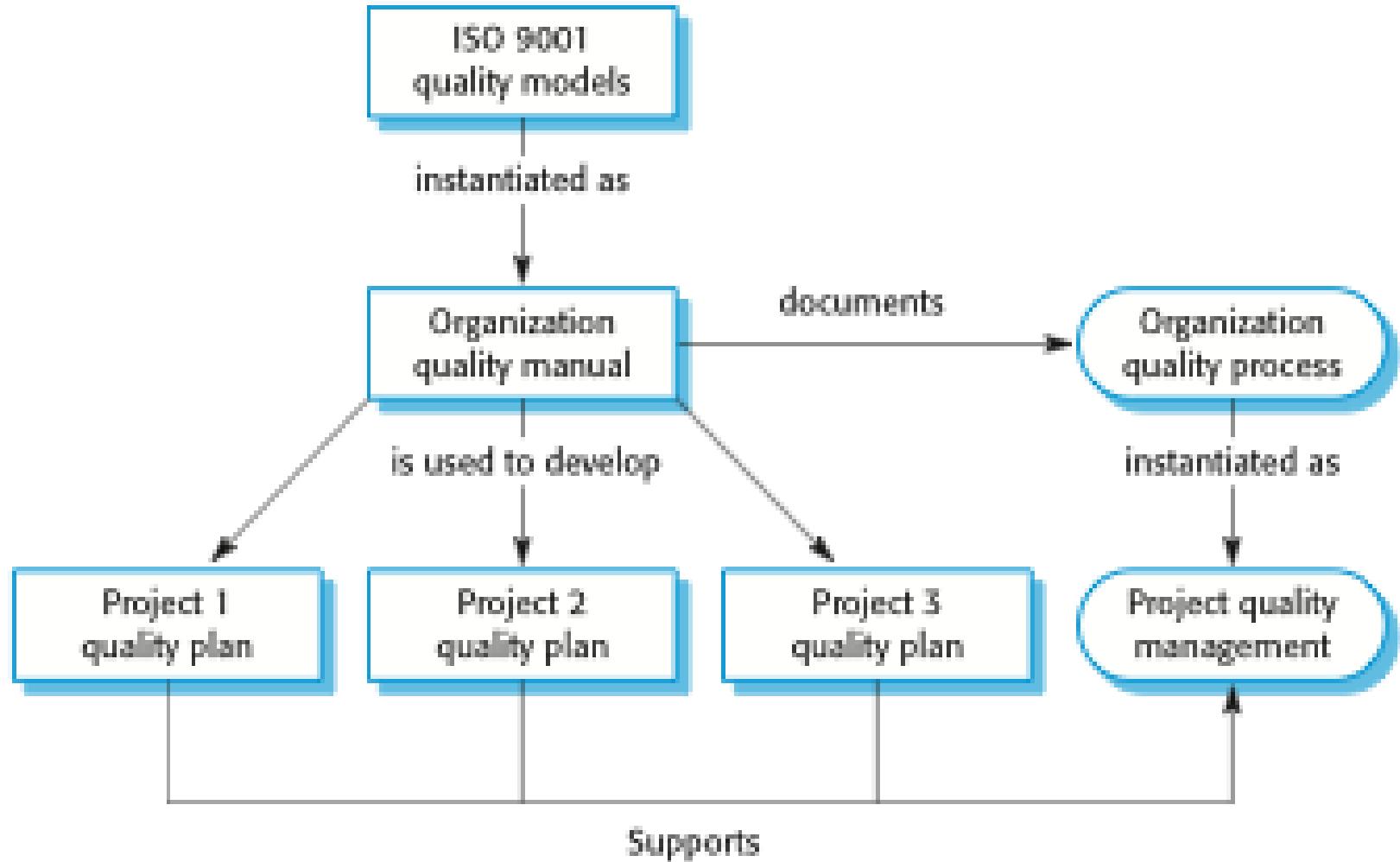
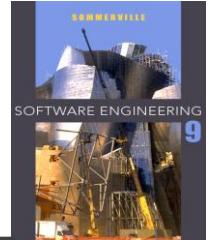
Business management

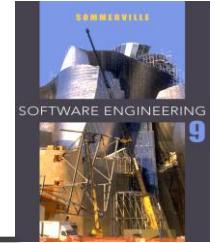
Supplier management

Inventory management

Configuration management

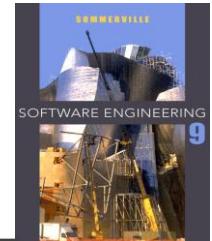
ISO 9001 and quality management





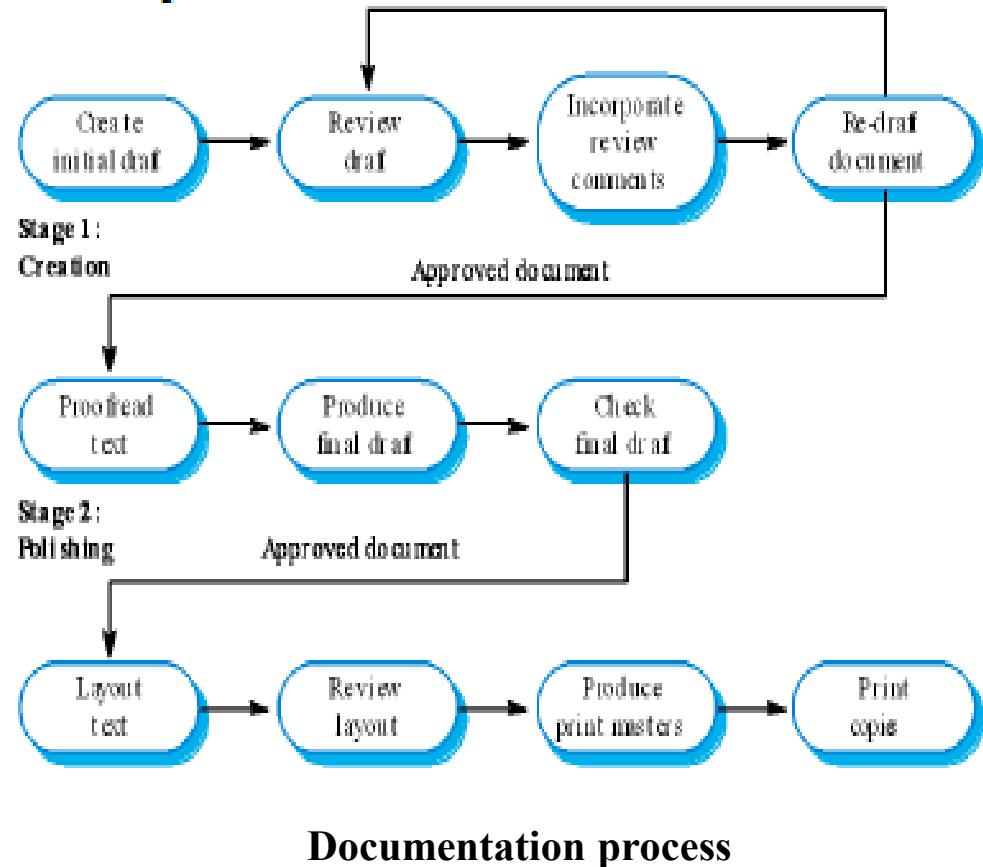
ISO IEC 90003:2004

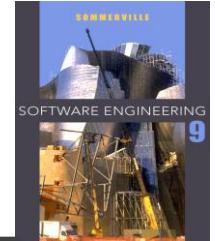
- ❖ ISO IEC 90003 2004 is really nothing more than ISO 9001 2000 applied to computer software and related services. It doesn't add to or change the ISO 9001 requirements in any way, it just explains and describes how you can meet these requirements if you're in the software business.
- ❖ ISO 90003 = ISO 9001 + ADVICE ON HOW TO APPLY ISO 9001



Documentation standards

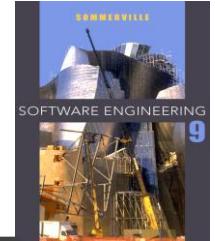
- ✧ Particularly important - documents are the tangible way of representing the software.
- ✧ Documentation process standards: Concerned with how documents should be developed, validated and maintained.
- ✧ Document standards: Concerned with document contents, structure, and appearance.
- ✧ Document interchange standards: Concerned with the compatibility of electronic documents.





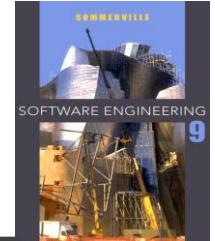
Documentation standards

- ✧ Document identification standards: How documents are uniquely identified.
- ✧ Document structure standards: Standard structure for project documents.
- ✧ Document presentation standards: Define fonts and styles, use of logos, etc.
- ✧ Document update standards: Define how changes from previous versions are reflected in a document.



Documentation interchange standards

- ❖ Interchange standards allow electronic documents to be exchanged, mailed, etc.
- ❖ Documents are produced using different systems and on different computers. Even when standard tools are used, standards are needed to define conventions for their use e.g. use of style sheets and macros.
- ❖ Need for archiving. The lifetime of word processing systems may be much less than the lifetime of the software being documented. An archiving standard may be defined to ensure that the document can be accessed in future.



Key points

- ✧ Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability and so on.
- ✧ SQM includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- ✧ Software standards are important for quality assurance as they represent an identification of 'best practice'.
- ✧ Quality management procedures may be documented in an organizational quality manual, based on the generic model for a quality manual suggested in the ISO 9001 standard.

Chapter 24 - Quality Management

Lecture 8

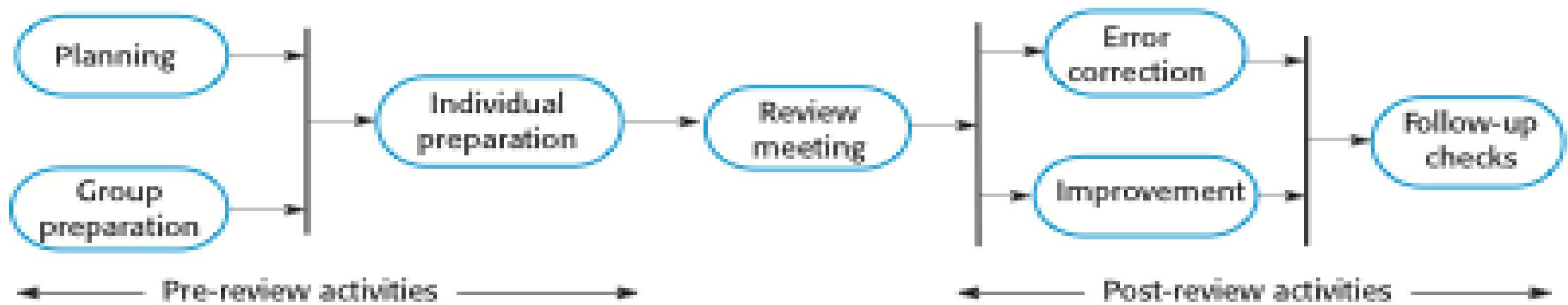
Quality control

- This involves checking the software development process to ensure that procedures and standards are being followed.
- There are two approaches to quality control:
 - Quality reviews;
 - Automated software assessment and software measurement.

Quality reviews

- ❑ This is the principal method of validating the quality of a process or of a product.
- ❑ A group of people carefully examine part or all of a software system and its associated cementation (support).
- ❑ Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- ❑ Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.
- ❑ There are different types of review with different objectives
 - Inspections for defect removal (product);
 - Reviews for progress assessment (product and process);
 - Quality reviews (product and standards).

The software review process



Program inspections

- These are peer reviews where engineers examine the source of a system with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design , configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

An inspection checklist

Fault class	Inspection check
Data faults	<ul style="list-style-type: none">• Are all program variables initialized before their values are used?• Have all constants been named?• Should the upper bound of arrays be equal to the size of the array or Size -1?• If character strings are used, is a delimiter explicitly assigned?• Is there any possibility of buffer overflow?
Control faults	<ul style="list-style-type: none">• For each conditional statement, is the condition correct?• Is each loop certain to terminate?• Are compound statements correctly bracketed?• In case statements, are all possible cases accounted for?• If a break is required after each case in case statements, has it been included?
Input/output faults	<ul style="list-style-type: none">• Are all input variables used?• Are all output variables assigned a value before they are output?• Can unexpected inputs cause corruption?

An inspection checklist

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none">• Do all function and method calls have the correct number of parameters?• Do formal and actual parameter types match?• Are the parameters in the right order?• If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	<ul style="list-style-type: none">• If a linked structure is modified, have all links been correctly reassigned?• If dynamic storage is used, has space been allocated correctly?• Is space explicitly deallocated after it is no longer required?
Exception management faults	<ul style="list-style-type: none">• Have all possible error conditions been taken into account?

Agile methods and inspections

- ❑ Agile processes rarely use formal inspection or peer review processes.
- ❑ Rather, they rely on team members cooperating to check each other's code, and informal guidelines, such as 'check before check-in', which suggest that programmers should check their own code.
- ❑ Extreme programming practitioners argue that pair programming is an effective substitute for inspection as this is, in effect, a continual inspection process.
- ❑ Two people look at every line of code and check it before it is accepted.

Software measurement and metrics

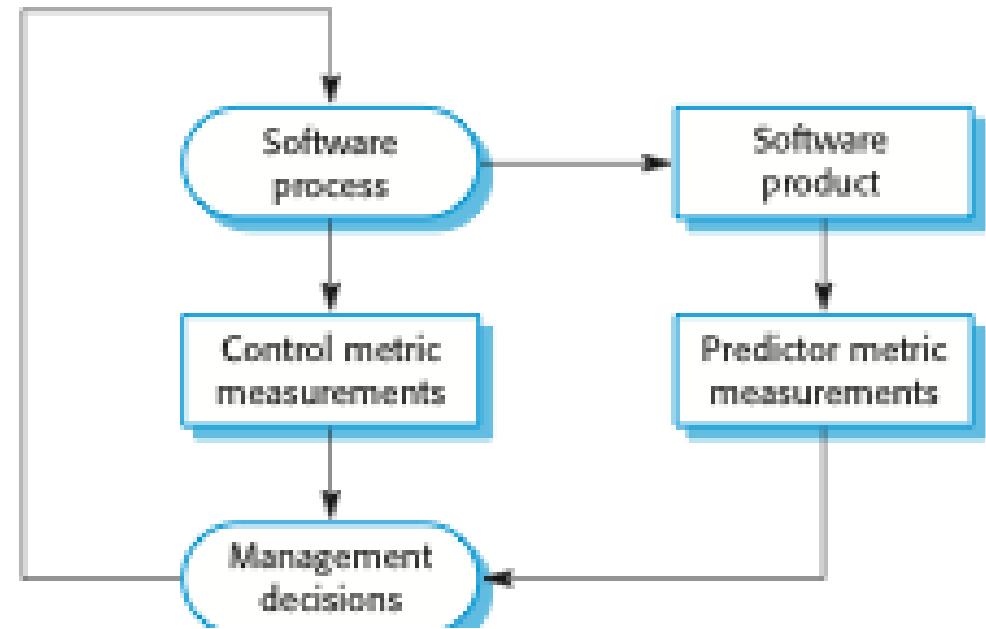
- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- There are few established standards in this area.

Software metric

- Any type of measurement which relates to a software system, process or related documentation
 - Lines of code in a program, the Fog index, number of person-days required to develop a component.
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

Predictor and control measurements

- S/W metrics may be either control metrics or predictor metrics.
- Control metrics are usually associated with S/W processes;
- Predictor metrics are associated with S/W products.



- Examples of control or process metrics are the average effort and time required to repair reported defects.
- Examples of Predictor metrics are the complexity of a module, the average length of identifiers in a program, and the number of attributes and operations associated with objects in design.

Use of measurements

- To assign a value to system quality attributes
 - By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.

- To identify the system components whose quality is sub-standard
 - Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

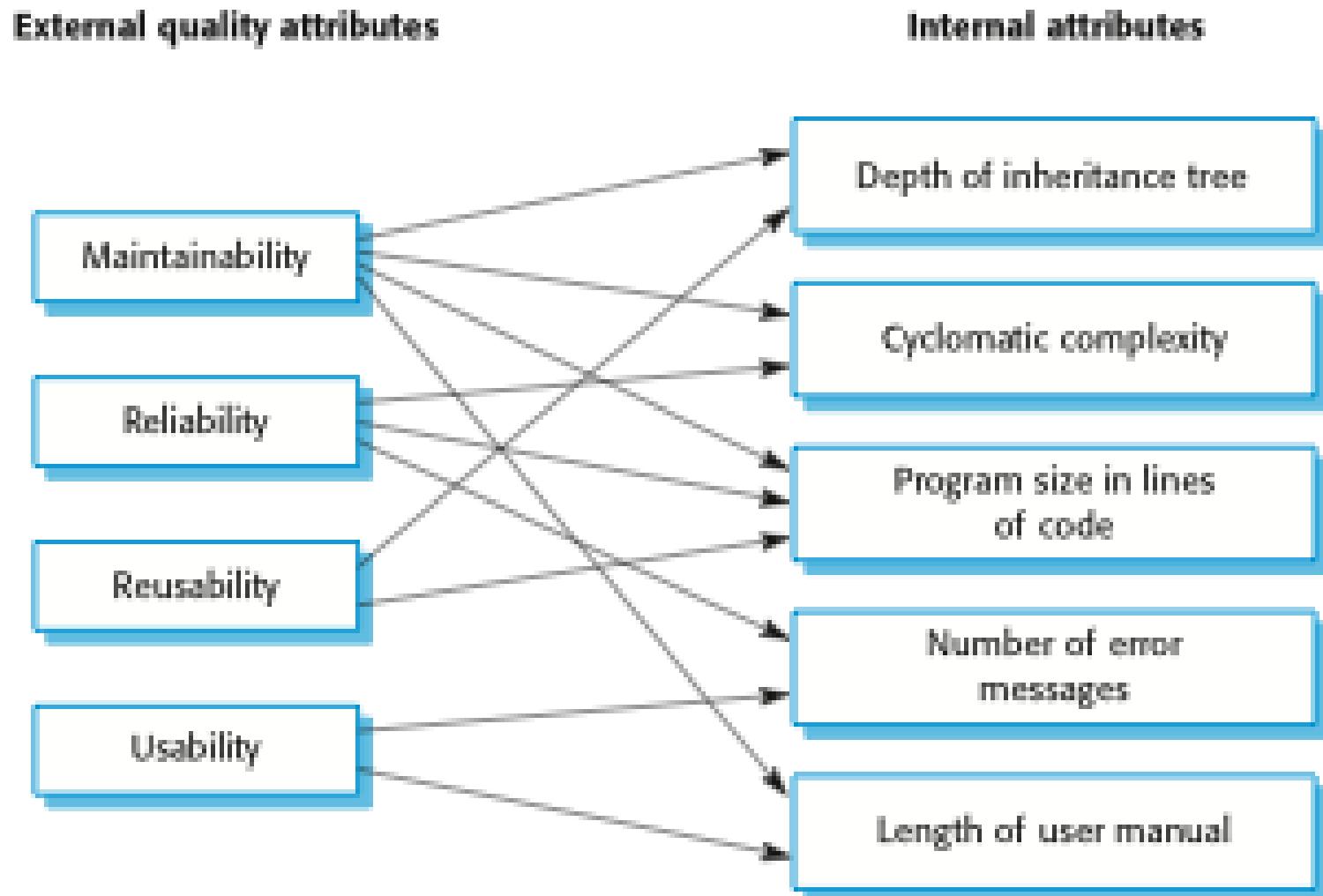
The measurement process

- A software measurement process may be part of a quality control process.
- Data collected during this process should be maintained as an organizational resource.
- Once a measurement database has been established, comparisons across projects become possible.

Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalised and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

Relationships between internal and external software



Product metrics

- A quality metric should be a predictor of product quality.
- Classes of product metric
 - Dynamic metrics which are collected by measurements made of a program in execution;
 - Static metrics which are collected by measurements made of the system representations;
 - Dynamic metrics help assess efficiency and reliability
 - Static metrics help assess complexity, understand ability and Maintainability.

Dynamic and static metrics

- Dynamic metrics are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- Static metrics have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

Static software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

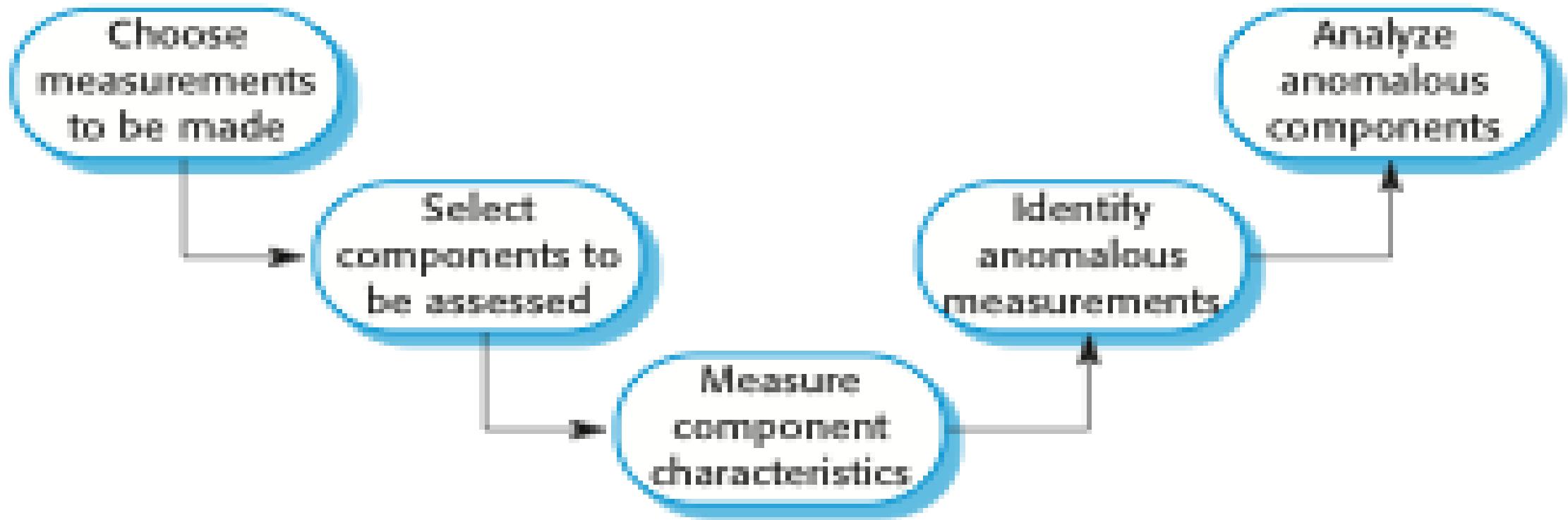
Static software product metrics

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Software component analysis

- System component can be analyzed separately using a range of metrics.
- The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

The process of product measurement



Measurement surprises

- Reducing the number of faults in a program leads to an increased number of help desk calls
 - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
 - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

Key points

- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed.
- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions
- Software measurement can be used to gather data about software and software processes.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems.

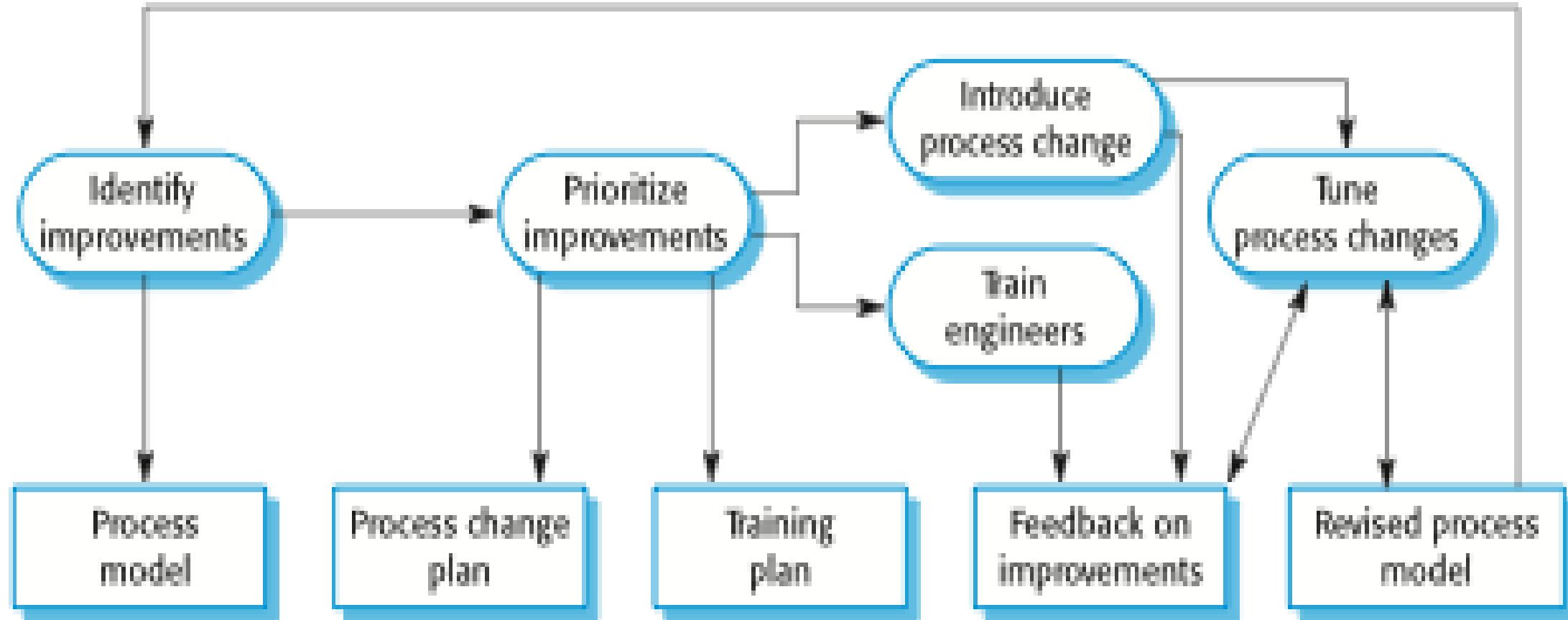
Chapter 26 – Process improvement

Lecture 10

Process change

- Involves making modifications to existing processes.
- This may involve:
 - Introducing new practices (performs), methods or processes;
 - Changing the ordering of process activities;
 - Introducing or removing deliverables;
 - Introducing new roles or responsibilities.
- Change should be driven by measurable goals.

The process change process



Process change stages

- Improvement identification
 - This stage is concerned with using the results of the process analysis , schedule bottlenecks or cost inefficiencies that have been identified during process analysis.
- Improvement prioritization
 - When many possible changes have been identified, it is usually impossible to introduce them all at once, and you must decide which are the most important.
- Process change introduction
 - Process change introduction means putting new procedures, methods and tools into place and integrating them with other process activities.

Process change stages

- Process change training
 - Without training, it is not possible to gain the full benefits of process changes. The engineers involved need to understand the changes that have been proposed and how to perform the new and changed processes.
- Change tuning
 - Proposed process changes will never be completely effective as soon as they are introduced. You need a tuning phase where minor problems can be discovered, and modifications to the process can be proposed and introduced.

Process change problems

- Resistance to change
 - Team members or project managers may resist the introduction of process changes and propose reasons why changes will not work, or delay the introduction of changes. They may, in some cases, purposely obstruct process changes and interpret data to show the ineffectiveness of proposed process change.
- Change persistence
 - While it may be possible to introduce process changes initially, it is common for process innovations to be discarded after a short time and for the processes to revert to their previous state.

Resistance to change

- Project managers often resist process change because any innovation has unknown risks associated with it.
 - Project managers are judged according to whether or not their project produces software on time and to budget. They may prefer an inefficient but predictable process to an improved process that has organizational benefits, but which has short-term risks associated with it.
- Engineers may resist the introduction of new processes for similar reasons, or because they see these processes as threatening their professionalism.
 - That is, they may feel that the new pre-defined process gives them less discretion and does not recognize the value of their skills and experience.

Change persistence

- The problem of changes being introduced then subsequently discarded is a common one.
 - Changes may be proposed by an ‘evangelist’ who believes strongly that the changes will lead to improvement. He or she may work hard to ensure the changes are effective and the new process is accepted.
 - If the ‘evangelist’ leaves, then the people involved may therefore simply revert to the previous ways of doing things.
- Change institutionalization is important
 - This means that process change is not dependent on individuals but that the changes become part of standard practice in the company, with company-wide support and training.

The CMMI process improvement framework

- The Capability Maturity Model Integration (CMMI) framework is the current stage of work on process assessment and improvement that started at the Software Engineering Institute in the 1980s.
- The CMMI principal is that “the quality of a system or product is highly influenced by the process used to develop and maintain it”.
- CMMI was developed by a group from industry, government, and the Software Engineering Institute (SEI).
- The SEI’s mission is to promote software technology transfer particularly to US defence contractors.

The SEI capability maturity model levels

- Initial
 - Essentially uncontrolled
- Repeatable
 - Product management procedures defined and used
- Defined
 - Process management procedures and strategies defined and used
- Managed
 - Quality management strategies defined and used
- Optimising
 - Process improvement strategies defined and used

Process capability assessment

- Intended as a means to assess the extent to which an organization's processes follow best practice (performs).
- By providing a means for assessment, it is possible to identify areas of weakness for process improvement.
- There have been various process assessment and improvement models but the SEI work has been most powerful.

The CMMI model

- An integrated capability model that includes software and systems engineering capability assessment.
- The model has two instantiations (forms)
 - Staged where the model is expressed in terms of capability levels;
 - Continuous where a capability rating is computed.

CMMI model components

- Process areas
 - 24 process areas that are relevant to process capability and improvement are identified. These are organized into 4 groups.
- Goals
 - Goals are descriptions of desirable organizational states. Each process area has associated goals.
- Practices
 - Practices are ways of achieving a goal - however, they are advisory and other approaches to achieve the goal may be used.

Process areas in the CMMI

Category	Process area
Process management	Organizational process definition (OPD)
	Organizational process focus (OPF)
	Organizational training (OT)
	Organizational process performance (OPP)
	Organizational innovation and deployment (OID)
Project management	Project planning (PP)
	Project monitoring and control (PMC)
	Supplier agreement management (SAM)
	Integrated project management (IPM)
	Risk management (RSKM)
	Quantitative project management (QPM)

Process areas in the CMMI

Category	Process area
Engineering	Requirements management (REQM)
	Requirements development (RD)
	Technical solution (TS)
	Product integration (PI)
	Verification (VER)
	Validation (VAL)
Support	Configuration management (CM)
	Process and product quality management (PPQA)
	Measurement and analysis (MA)
	Decision analysis and resolution (DAR)
	Causal analysis and resolution (CAR)

Goals and associated practices in the CMMI

Goal	Associated practices
The requirements are analyzed and validated, and a definition of the required functionality is developed.	Analyze derived requirements systematically to ensure that they are necessary and sufficient.
	Validate requirements to ensure that the resulting product will perform as intended in the user's environment, using multiple techniques as appropriate.
Root causes of defects and other problems are systematically determined.	Select the critical defects and other problems for analysis.
	Perform causal analysis of selected defects and other problems and propose actions to address them.
The process is institutionalized as a defined process.	Establish and maintain an organizational policy for planning and performing the requirements development process.

Examples of goals in the CMMI

Goal	Process area
Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan.	Project monitoring and control (specific goal)
Actual performance and progress of the project are monitored against the project plan.	Project monitoring and control (specific goal)
The requirements are analyzed and validated, and a definition of the required functionality is developed.	Requirements development (specific goal)
Root causes of defects and other problems are systematically determined.	Causal analysis and resolution (specific goal)
The process is institutionalized as a defined process.	Generic goal

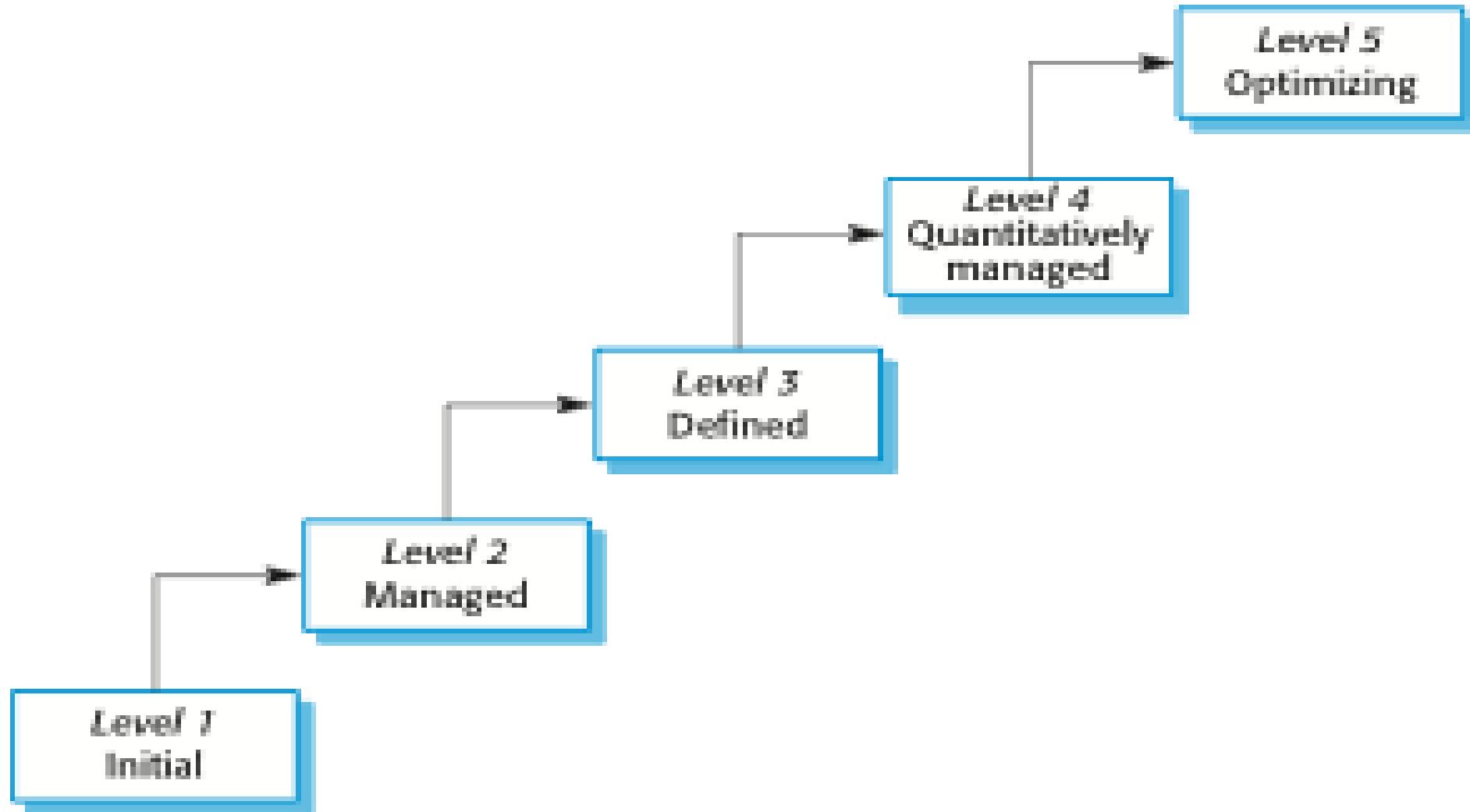
CMMI assessment

- Examines the processes used in an organization and assesses their maturity in each process area.
- Based on a 6-point scale:
 - Not performed;
 - Performed;
 - Managed;
 - Defined;
 - Quantitatively managed;
 - Optimizing.

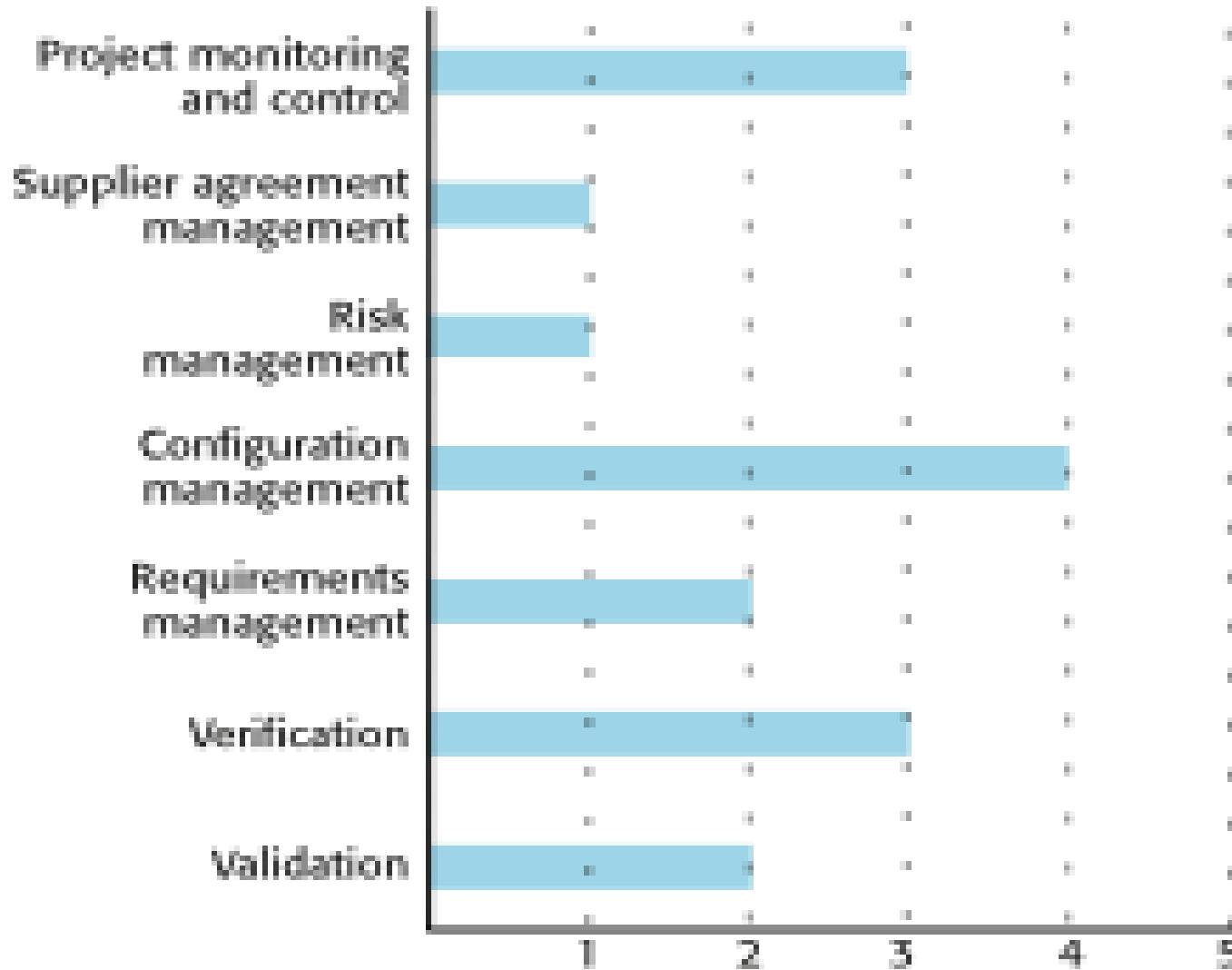
The staged CMMI model

- Comparable with the software CMM.
- Each maturity level has process areas and goals. For example, the process area associated with the managed level include:
 - Requirements management;
 - Project planning;
 - Project monitoring and control;
 - Supplier agreement management;
 - Measurement and analysis;
 - Process and product quality assurance.

The CMMI staged maturity model

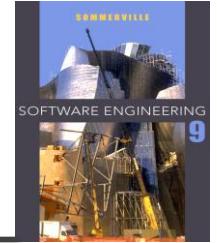


A process capability profile



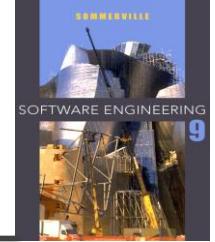
Key points

- The CMMI process maturity model is an integrated process improvement model that supports both staged and continuous process improvement.
- Process improvement in the CMMI model is based on reaching a set of goals related to good software engineering practice and describing, standardizing and controlling the practices used to achieve these goals.
- The CMMI model includes recommended practices that may be used, but these are not obligatory.



Chapter 26 – Process improvement

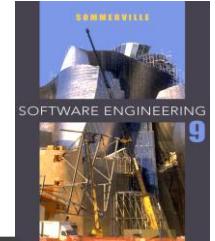
Lecture 9



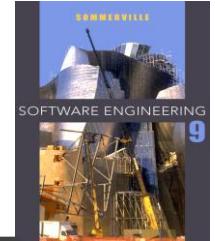
Topics covered

- ✧ The process improvement process
- ✧ Process measurement
- ✧ Process analysis
- ✧ Process change
- ✧ The CMMI process improvement framework

Process improvement



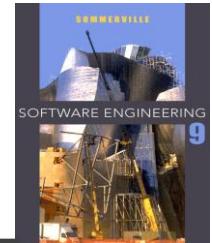
- ✧ Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- ✧ Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.



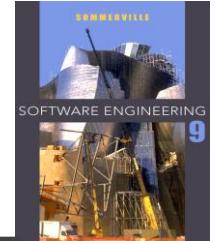
Approaches to improvement

- ✧ The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
 - The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
- ✧ The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
 - The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

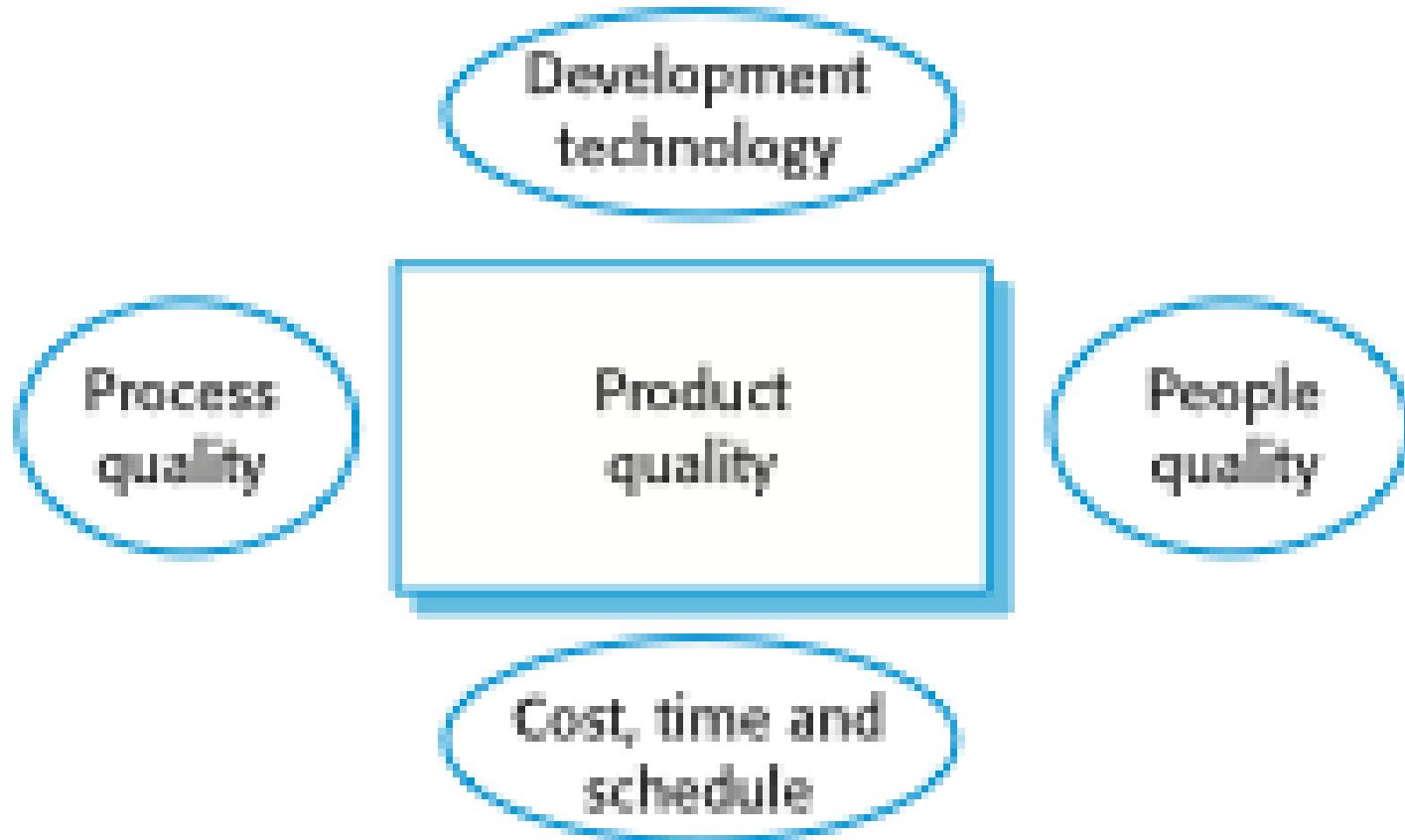
Process and product quality

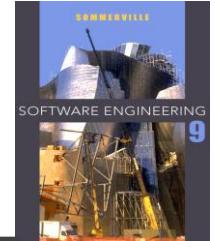


- ✧ Process quality and product quality are closely related and process improvement benefits arise because the quality of the product depends on its development process.
- ✧ A good process is usually required to produce a good product.
- ✧ For manufactured goods, process is the principal quality determinant.
- ✧ For design-based activities, other factors are also involved, especially the capabilities of the designers.



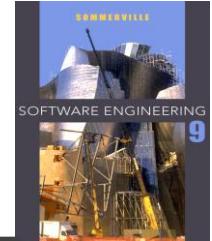
Factors affecting software product quality





Quality factors

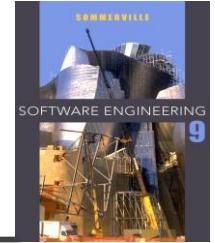
- ✧ For large projects with ‘average’ capabilities, the development process determines product quality.
- ✧ For small projects, the capabilities of the developers is the main determinant.
- ✧ The development technology is particularly significant for small projects.
- ✧ In all cases, if an unrealistic schedule is imposed then product quality will suffer.



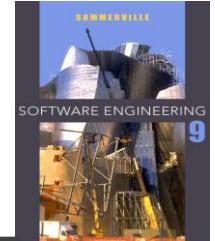
Process improvement process

- ✧ There is no such thing as an ‘ideal’ or ‘standard’ software process that is applicable in all organizations or for all software products of a particular type.
 - You will rarely be successful in introducing process improvements if you simply attempt to change the process to one that is used elsewhere.
 - You must always consider the local environment and culture and how this may be affected by process change proposals.
- ✧ Each company has to develop its own process depending on its size, the background and skills of its staff, the type of software being developed, customer and market requirements, and the company culture.

Improvement attributes

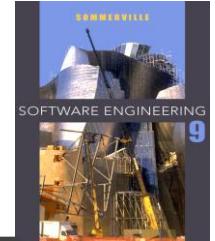


- ✧ You have to consider what aspects of the process that you want to improve.
- ✧ Your goal might be to improve software quality and so you may wish to introduce new process activities that change the way software is developed and tested.
- ✧ You may be interested in improving some attribute of the process itself (such as development time) and you have to decide which process attributes are the most important to your company.



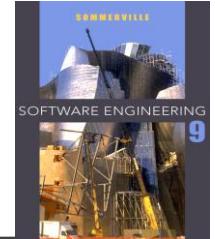
Process attributes

Process characteristic	Key issues
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Standardization	To what extent is the process based on a standard generic process? This may be important for some customers who require conformance with a set of defined process standards. To what extent is the same process used in all parts of a company?
Visibility	Do the process activities culminate in clear results, so that the progress of the process is externally visible?
Measurability	Does the process include data collection or other activities that allow process or product characteristics to be measured?
Supportability	To what extent can software tools be used to support the process activities?



Process attributes

Process characteristic	Key issues
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organizational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?



Process improvement stages

❖ Process measurement

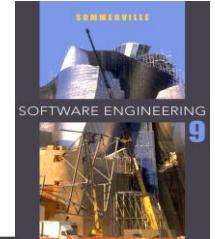
- Attributes of the current process are measured. These are a baseline for assessing improvements.

❖ Process analysis and model

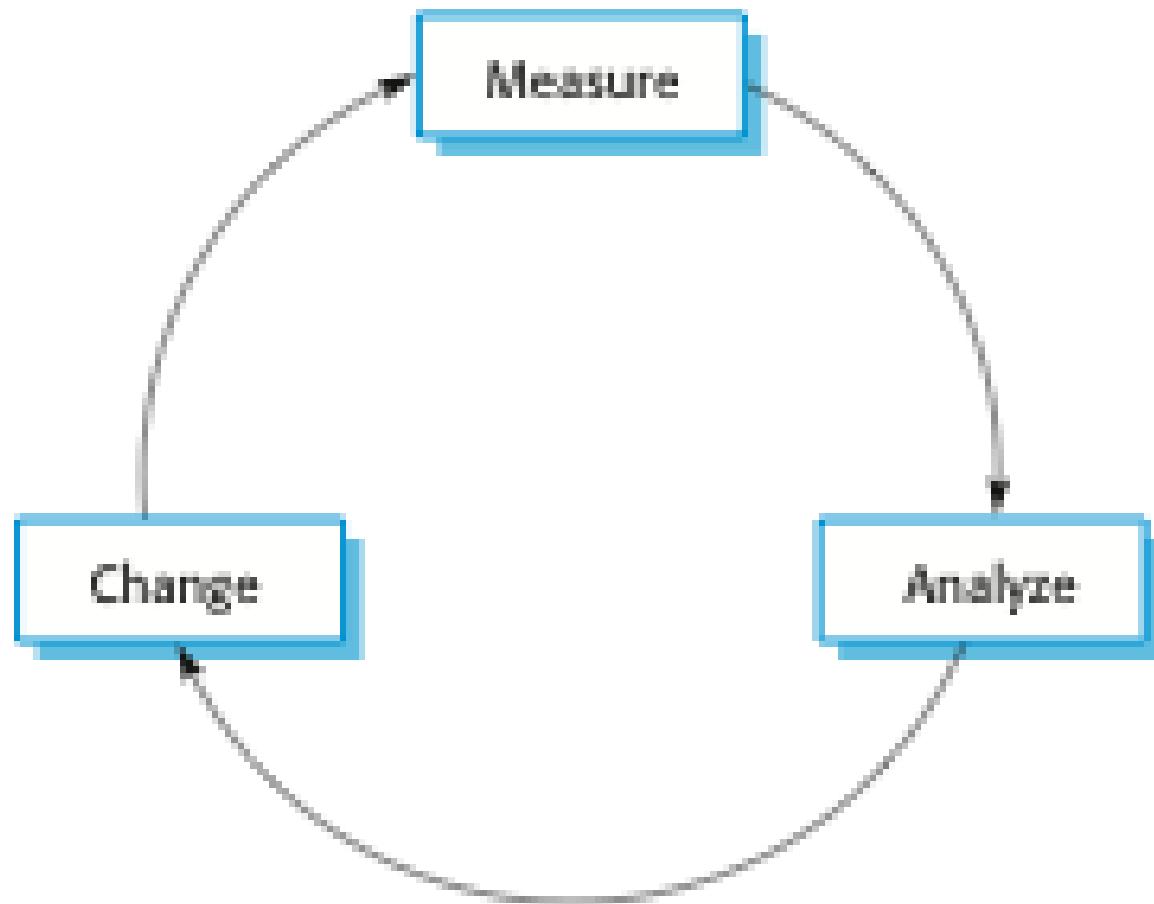
- The current process is assessed and bottlenecks and weaknesses are identified.

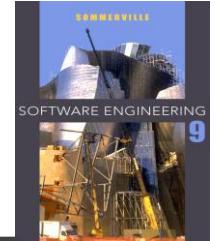
❖ Process change

- Changes to the process that have been identified during the analysis are introduced.



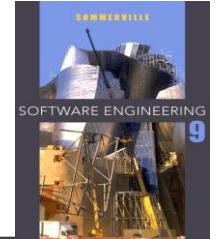
The process improvement cycle





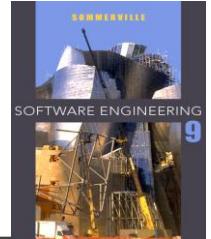
Process measurement

- ✧ Wherever possible, quantitative process data should be collected
 - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
 - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.



Process metrics

- ✧ Time taken for process activities to be completed
 - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
 - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
 - E.g. Number of defects discovered.



Goal-Question-Metric Paradigm

✧ Goals

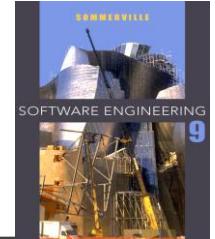
- What is the organisation trying to achieve? The objective of process improvement is to satisfy these goals.

✧ Questions

- Questions about areas of uncertainty related to the goals. You need process knowledge to derive these.

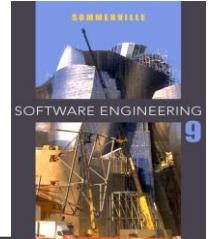
✧ Metrics

- Measurements to be collected to answer the questions.

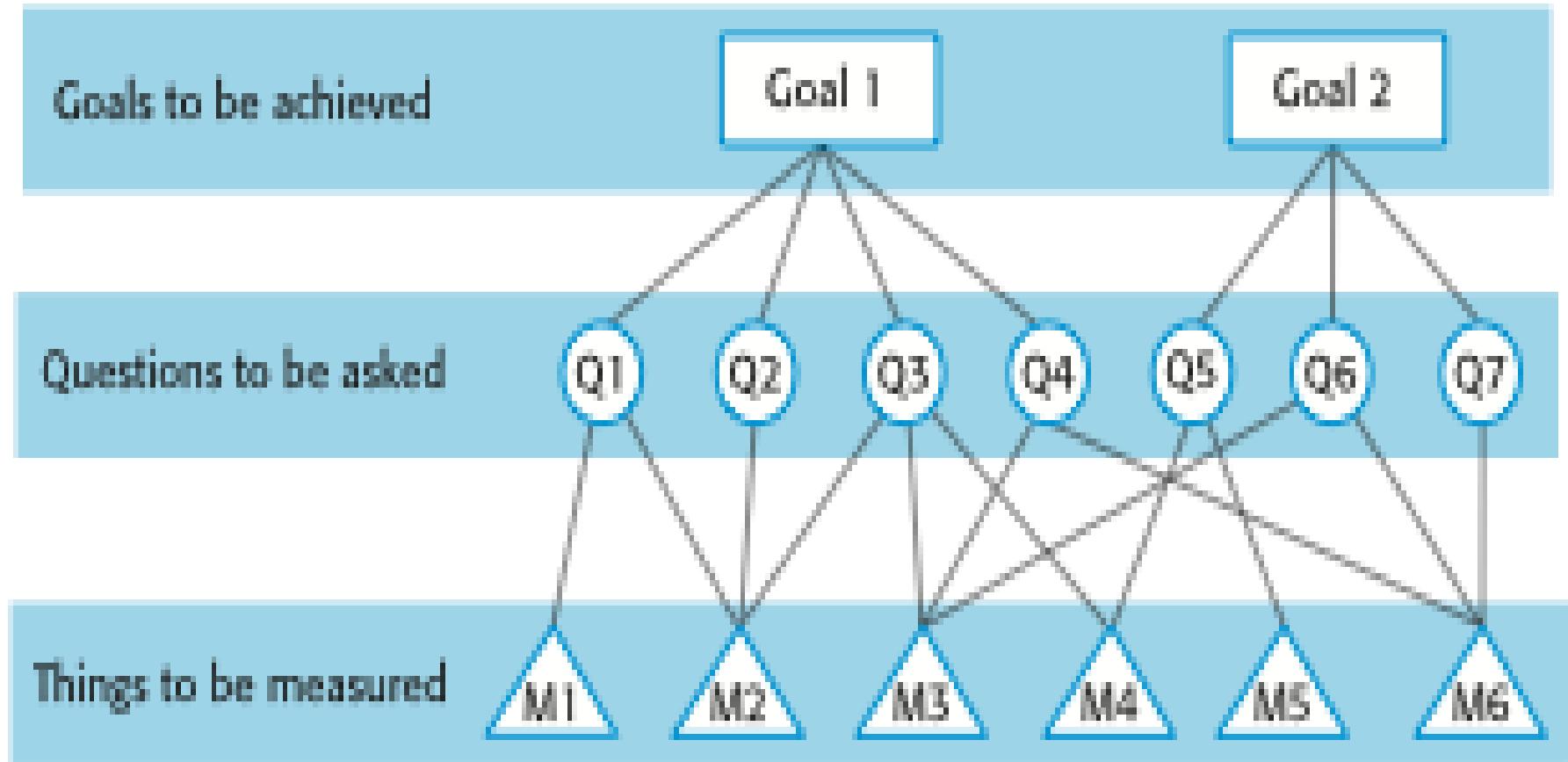


GQM questions

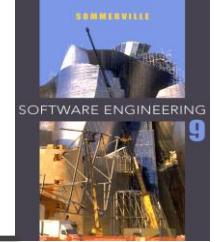
- ✧ The GQM paradigm is used in process improvement to help answer three critical questions:
 - Why are we introducing process improvement?
 - What information do we need to help identify and assess improvements?
 - What process and product measurements are required to provide this information?



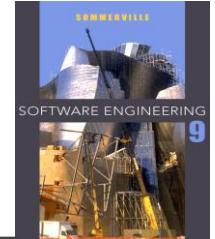
The GQM paradigm



Process analysis

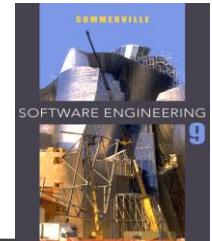


- ❖ The study of existing processes to understand the relationships between parts of the process and to compare them with other processes.
- ❖ Process analysis and process measurement are intertwined.
- ❖ You need to carry out some analysis to know what to measure, and, when making measurements, you inevitably develop a deeper understanding of the process being measured.



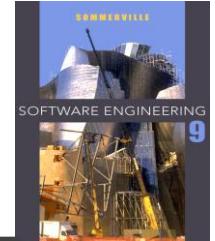
Process analysis objectives

- ✧ To understand the activities involved in the process and the relationships between these activities.
- ✧ To understand the relationships between the process activities and the measurements that have been made.
- ✧ To relate the specific process or processes that you are analyzing to comparable processes elsewhere in the organization, or to idealized processes of the same type.



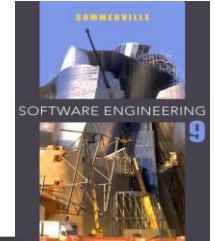
Process analysis techniques

- ✧ Published process models and process standards
 - It is always best to start process analysis with an existing model. People then may extend and change this.
- ✧ Questionnaires and interviews
 - Must be carefully designed. Participants may tell you what they think you want to hear.
- ✧ Ethnographic (habits) analysis
 - Involves assimilating process knowledge by observation. Best for in-depth analysis of process fragments rather than for whole-process understanding.



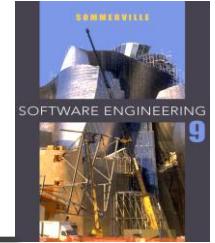
Aspects of process analysis

Process aspect	Questions
Adoption and standardization	Is the process documented and standardized across the organization? If not, does this mean that any measurements made are specific only to a single process instance? If processes are not standardized, then changes to one process may not be transferable to comparable processes elsewhere in the company.
Software engineering practice	Are there known, good software engineering practices that are not included in the process? Why are they not included? Does the lack of these practices affect product characteristics, such as the number of defects in a delivered software system?
Organizational constraints	What are the organizational constraints that affect the process design and the ways that the process is performed? For example, if the process involves dealing with classified material, there may be activities in the process to check that classified information is not included in any material due to be released to external organizations. Organizational constraints may mean that possible process changes cannot be made.



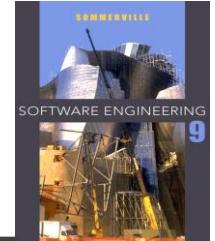
Aspects of process analysis

Process aspect	Questions
Communications	How are communications managed in the process? How do communication issues relate to the process measurements that have been made? Communication problems are a major issue in many processes and communication bottlenecks are often the reasons for project delays.
Introspection	Is the process reflective (i.e., do the actors involved in the process explicitly think about and discuss the process and how it might be improved)? Are there mechanisms through which process actors can propose process improvements?
Learning	How do people joining a development team learn about the software processes used? Does the company have process manuals and process training programs?
Tool support	What aspects of the process are and aren't supported by software tools? For unsupported areas, are there tools that could be deployed cost-effectively to provide support? For supported areas, are the tools effective and efficient? Are better tools available?



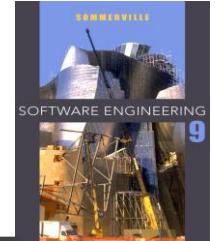
Process models

- ✧ Process models are a good way of focusing attention on the activities in a process and the information transfer between these activities.
- ✧ Process models do not have to be formal or complete
- ✧ Their purpose is to provoke (needle) discussion rather than document the process in detail.
- ✧ Model-oriented questions can be used to help understand the process e.g.
 - What activities take place in practice but are not shown in the model?
 - Are there process activities, shown in the model, that you (the process actor) think are inefficient?



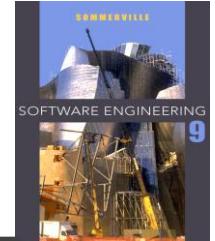
Process exceptions

- ❖ Software processes are complex and process models cannot effectively represent how to handle exceptions:
 - Several key people becoming ill just before a critical review;
 - A violations of security that means all external communications are out of action for several days;
 - Organisational reorganisation;
 - A need to respond to an unanticipated request for new proposals.
- ❖ Under these circumstances, the model is suspended and managers use their initiative to deal with the exception.



Key points

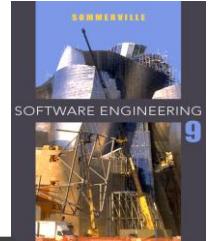
- ✧ The goals of process improvement are higher product quality, reduced process costs and faster delivery of software.
- ✧ The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- ✧ The process improvement cycle involves process measurement, process analysis and modeling, and process change.
- ✧ Measurement should be used to answer specific questions about the software process used. These questions should be based on organizational improvement goals.



Chapter 25

Configuration Management

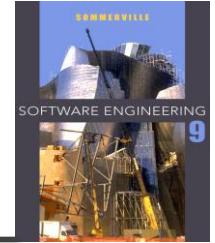
Lecture 11



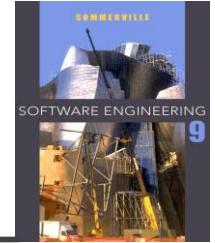
Topics covered

- ✧ Change management
- ✧ Version management
- ✧ System building
- ✧ Release management

Configuration management and why



- ❖ Because software changes frequently, systems, can be thought of as a set of versions, each of which has to be maintained and managed.
- ❖ Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems.
- ❖ Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems.
- ❖ You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.



CM activities

✧ Change management

Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.

✧ Version management

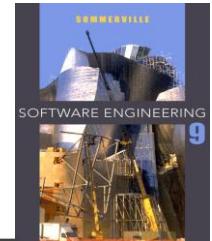
Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.

✧ System building

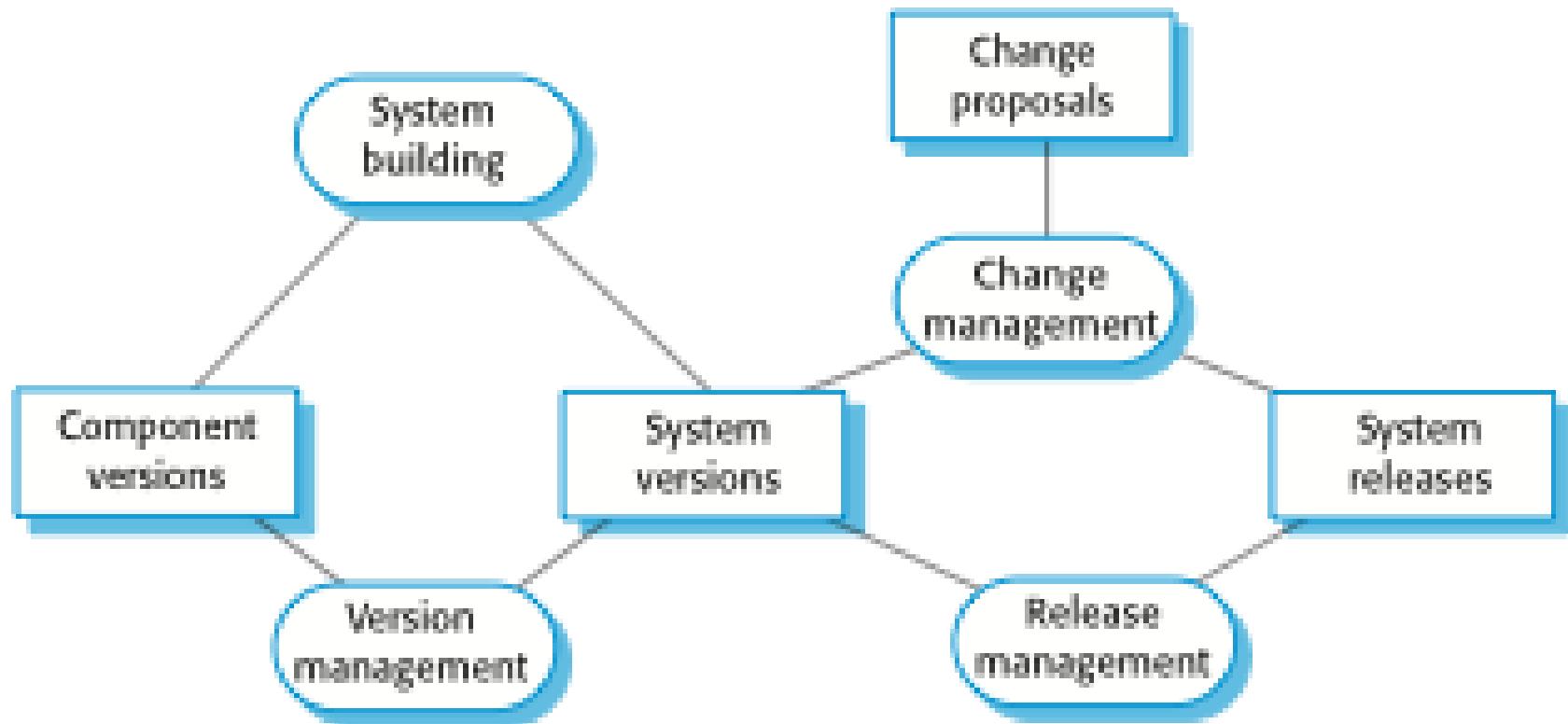
The process of assembling program components, data and libraries, then compiling these to create an executable system.

✧ Release management

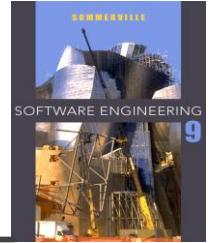
Preparing software for external release and keeping track of the system versions that have been released for customer use.



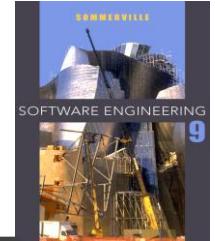
Configuration management activities



CM terminology

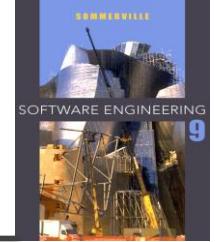


Term	Explanation
<u>Configuration item or software configuration item (SCI)</u>	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
<u>Configuration control</u>	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
<u>Version</u>	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
<u>Baseline</u>	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to recreate a baseline from its constituent components.
<u>Codeline</u>	A codeline is a set of versions of a software component and other configuration items on which that component depends.



CM terminology

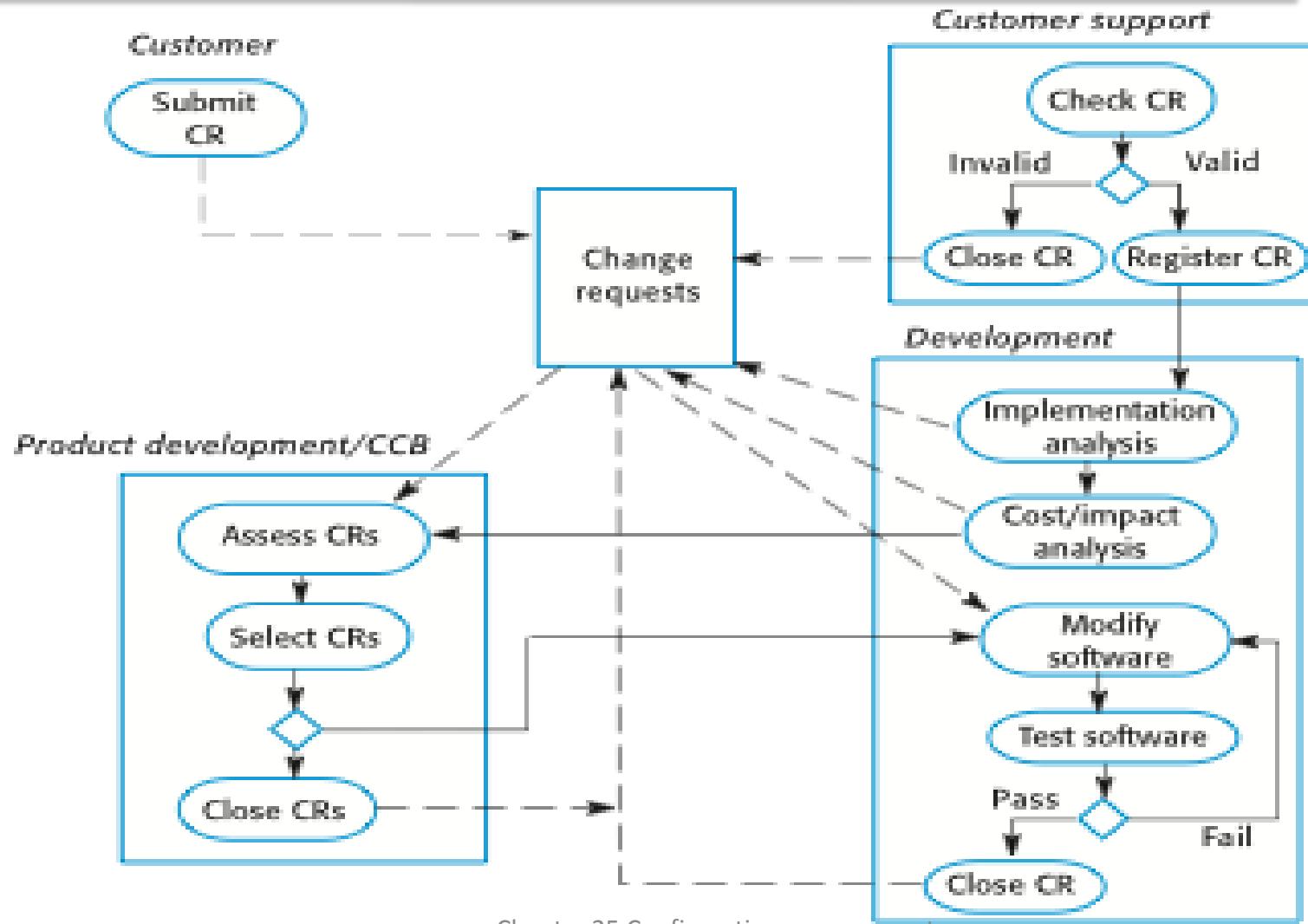
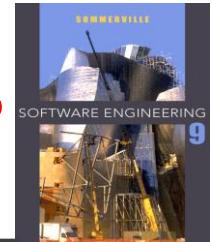
Term	Explanation
<u>Mainline</u>	A <u>sequence of baselines representing different versions of a system.</u>
<u>Release</u>	A <u>version of a system that has been released to customers (or other users in an organization) for use.</u>
<u>Workspace</u>	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
<u>Branching</u>	The <u>creation of a new codeline from a version in an existing codeline.</u> The new codeline and the existing codeline may then develop independently.
<u>Merging</u>	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
<u>System building</u>	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

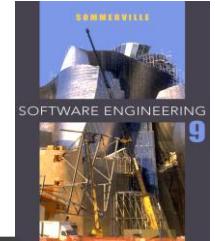


Change management

- ❖ The change management process is concerned with analyzing the costs and benefits of proposed changes, keeping track of these changes and ensuring that they are implemented in the most cost-effective way.
- ❖ Change requests: From users – From developers – From market forces.
- ❖ Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.

The change management process





The change management process

Request change by completing a change request form

Analyze change request

if change is valid then

 Assess how change might be implemented

 Assess change cost

 Submit request to change control board

 if change is accepted then

 repeat

 make changes to software

 submit changed software for quality approval

 until software quality is adequate

 create new system version

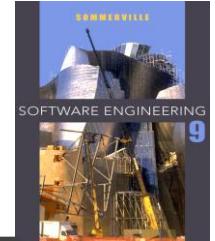
 else

 reject change request

else

 reject change request

A partially completed change request form (a)



Change Request Form

Project: SICSA/AppProcessing

Number: 23/02

Change requester: I. Sommerville

Date: 20/01/09

Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

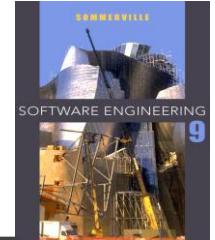
Change analyzer: R. Looek

Analysis date: 25/01/09

Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

A partially completed change request form (b)



Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium

Change implementation:

Estimated effort: 2 hours

Date to SGA app. team: 28/01/09

CCB decision date: 30/01/09

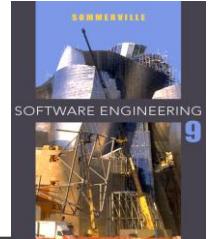
Decision: Accept change. Change to be implemented in Release 1.2

Change implementor: **Date of change:**

Date submitted to QA: **QA decision:**

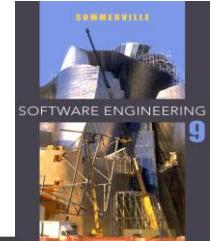
Date submitted to CM:

Comments:



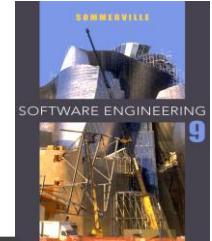
Factors in change analysis

- ✧ The consequences (penalties) of not making the change
- ✧ The benefits of the change
- ✧ The number of users affected by the change
- ✧ The costs of making the change
- ✧ The product release cycle



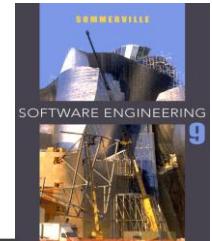
Version management

- ❖ Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.
- ❖ It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- ❖ Therefore version management can be thought of as the process of managing codelines and baselines.



Codelines and Baselines

- ❖ A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- ❖ Codelines normally apply to components of systems so that there are different versions of each component.
- ❖ A baseline is a definition of a specific system.
- ❖ The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.



Codelines and baselines

Codeline (A)



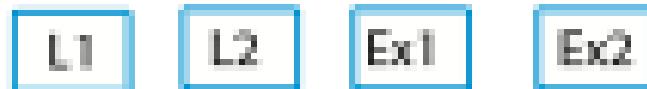
Codeline (B)



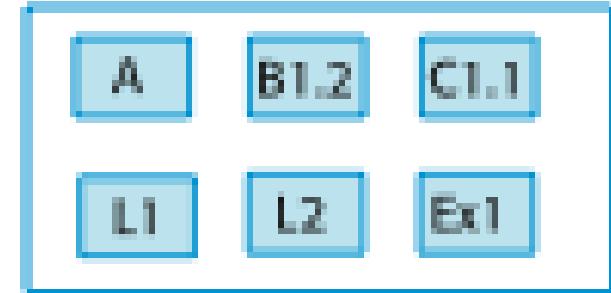
Codeline (C)



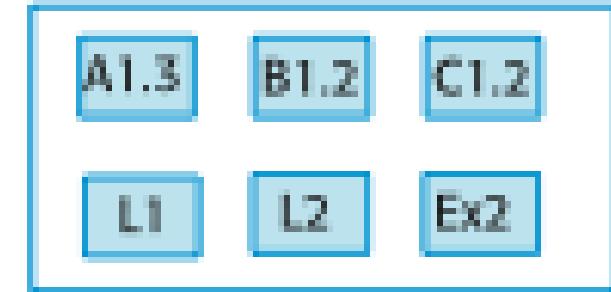
Libraries and external components



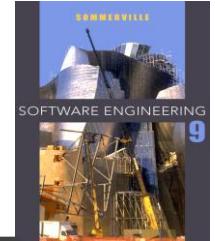
Baseline - V1



Baseline - V2

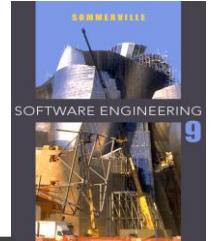


Mainline



Baselines

- ✧ Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- ✧ Baselines are important because you often have to recreate a specific version of a complete system.
 - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.



Version management systems

✧ Version and release identification

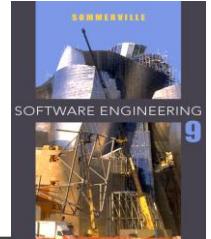
- Managed versions are assigned identifiers when they are submitted to the system.

✧ Storage management

- To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities.

✧ Change history recording

- All of the changes made to the code of a system or component are recorded and listed.



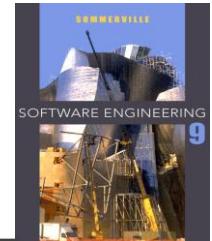
Version management systems

✧ Independent development

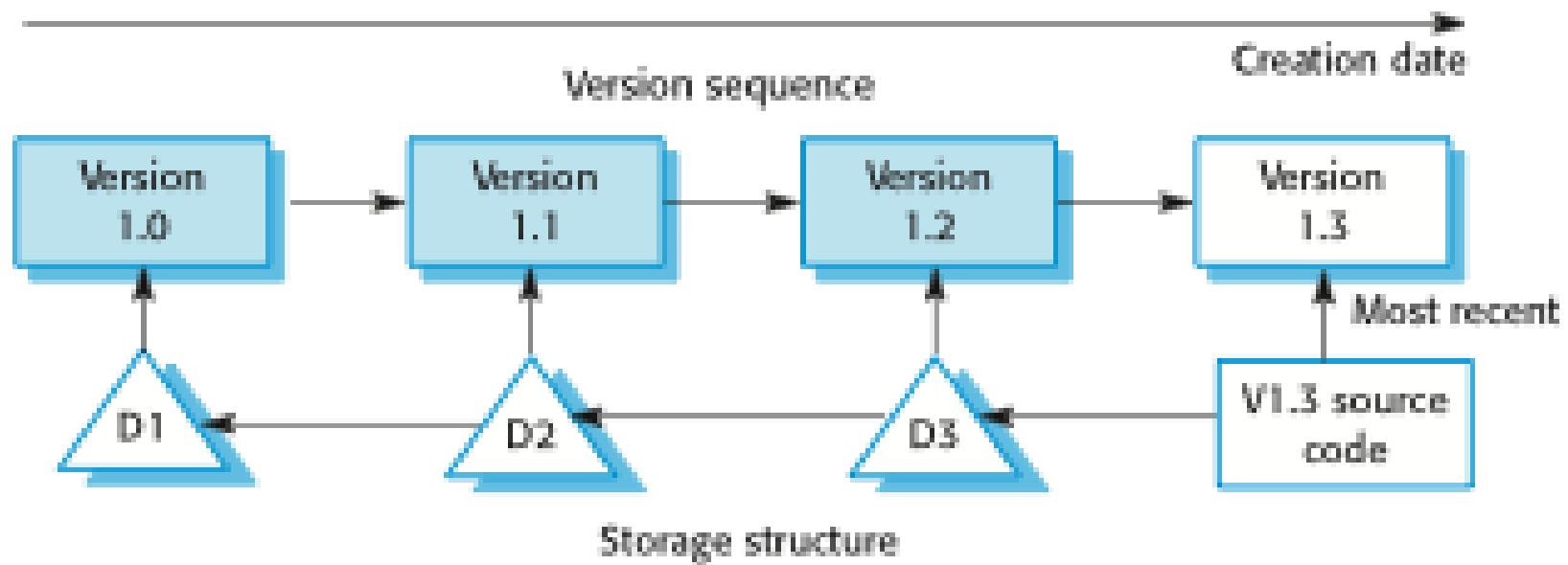
- The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

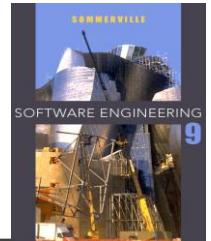
✧ Project support

- A version management system may support the development of several projects, which share components.

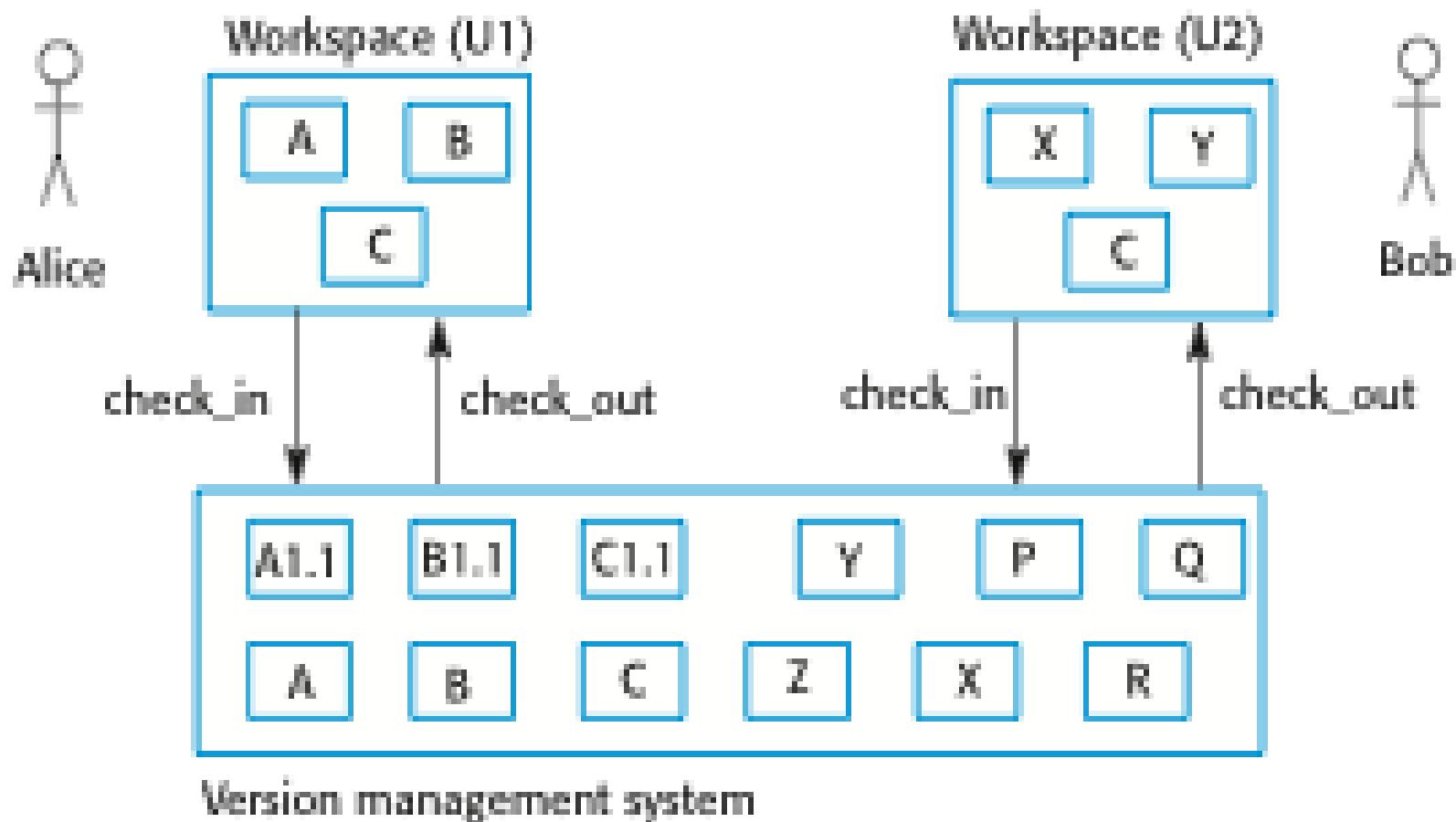


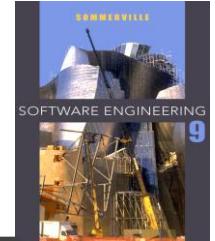
Storage management using deltas





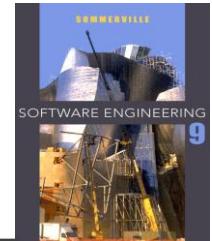
Check-in and check-out from a version repository



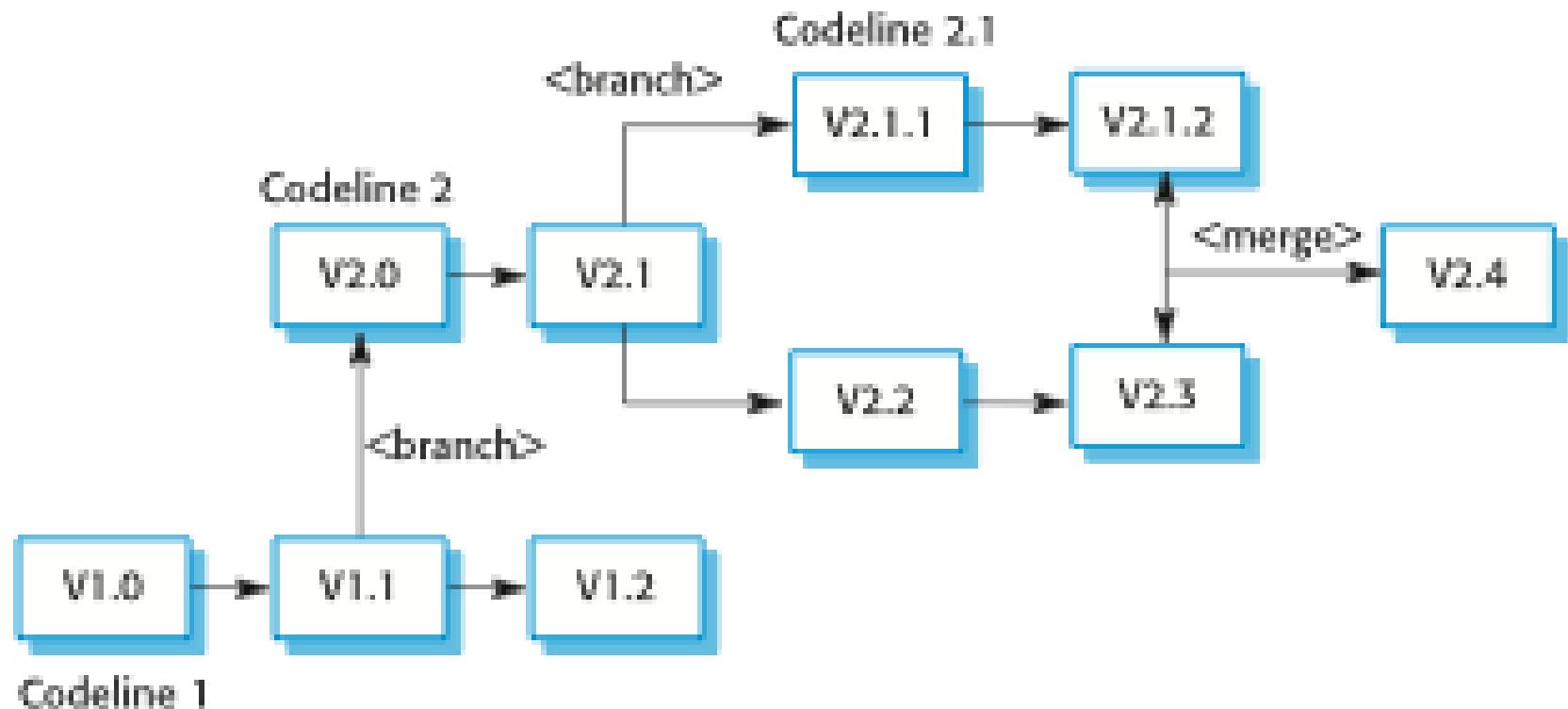


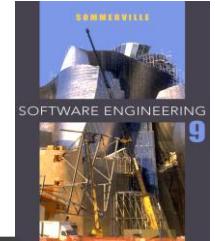
Codeline branches

- ✧ Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.
 - This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- ✧ At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
 - If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.



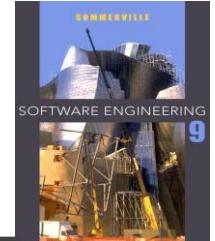
Branching and merging





Key points

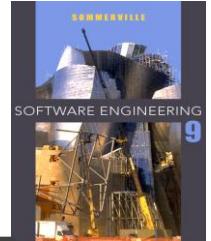
- ✧ Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- ✧ The main configuration management processes are change management, version management, system building and release management.
- ✧ Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- ✧ Version management involves keeping track of the different versions of software components as changes are made to them.



Chapter 25

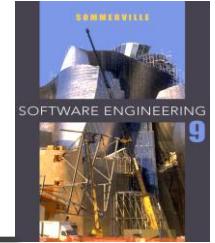
Configuration Management

Lecture 12



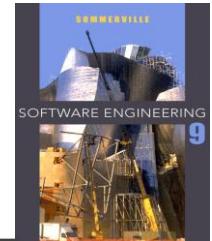
System building

- ✧ System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.
- ✧ System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- ✧ The configuration description used to identify a baseline is also used by the system building tool.

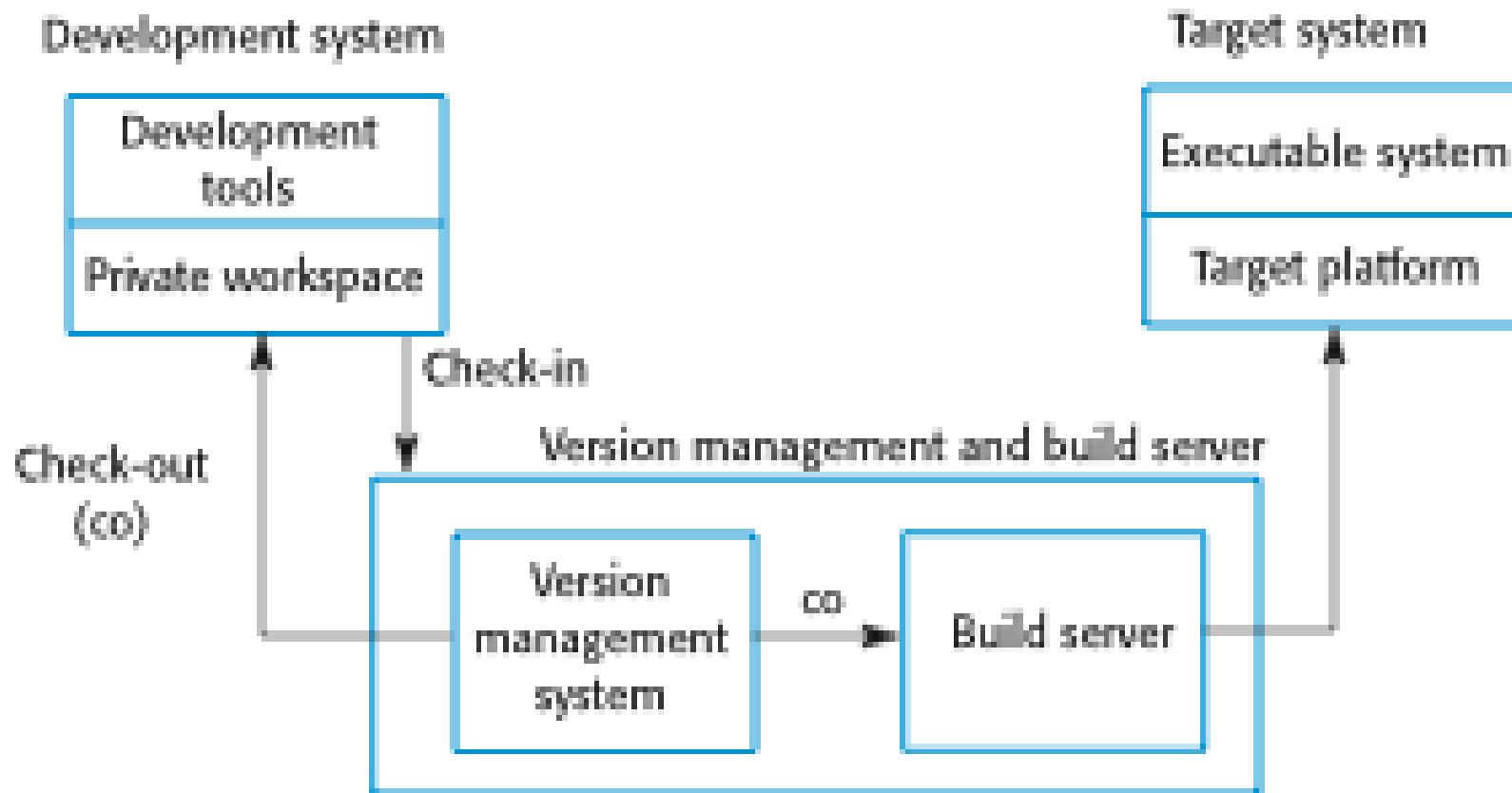


Build platforms

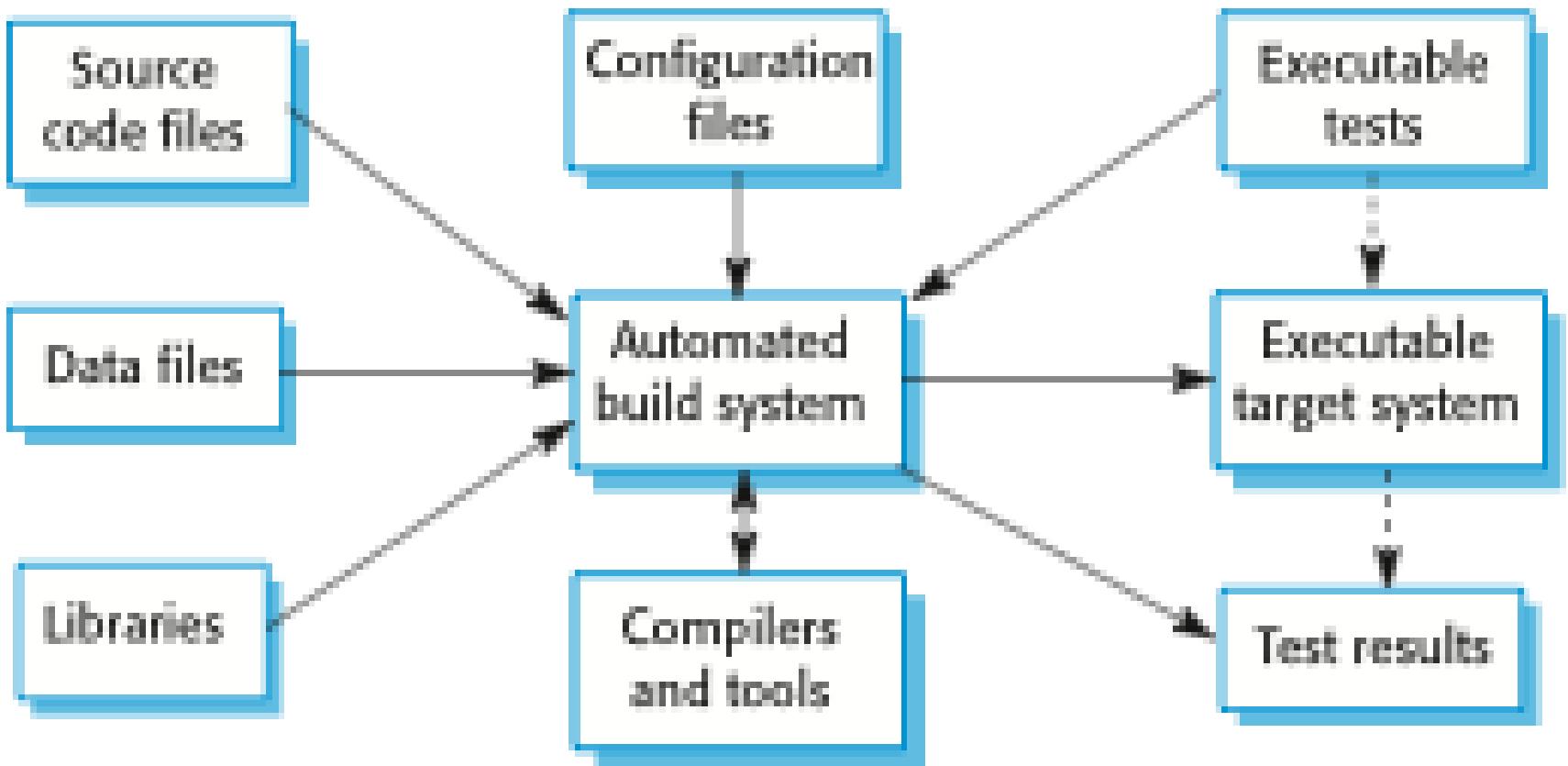
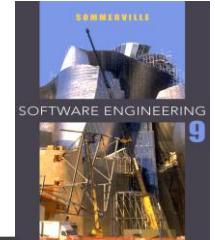
- ✧ The development system, which includes development tools such as compilers, source code editors, etc.
 - Developers check out code from the version management system into a private workspace before making changes to the system.
- ✧ The build server, which is used to build definitive, executable versions of the system.
 - Developers check-in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.
- ✧ The target environment, which is the platform on which the system executes.

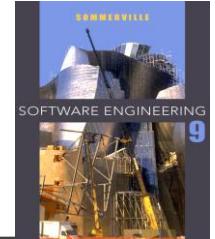


Development, build, and target platforms



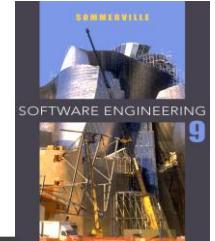
System building





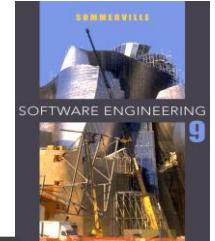
Build system functionality

- ✧ Build script generation
- ✧ Version management system integration
- ✧ Minimal re-compilation
- ✧ Executable system creation
- ✧ Test automation
- ✧ Reporting
- ✧ Documentation generation



Minimizing recompilation

- ✧ Tools to support system building are usually designed to minimize the amount of compilation that is required.
- ✧ They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- ✧ A unique signature identifies each source and object code version and is changed when the source code is edited.
- ✧ By comparing the signatures on the source and object code files, it is possible to decide if the source code was used to generate the object code component.



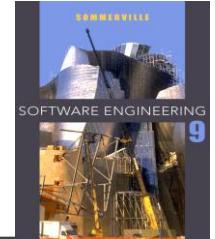
File identification

❖ Modification timestamps

- The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

❖ Source code checksums

- The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by 1 character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.



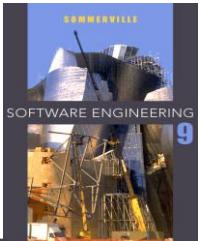
Timestamps vs checksums

❖ Timestamps

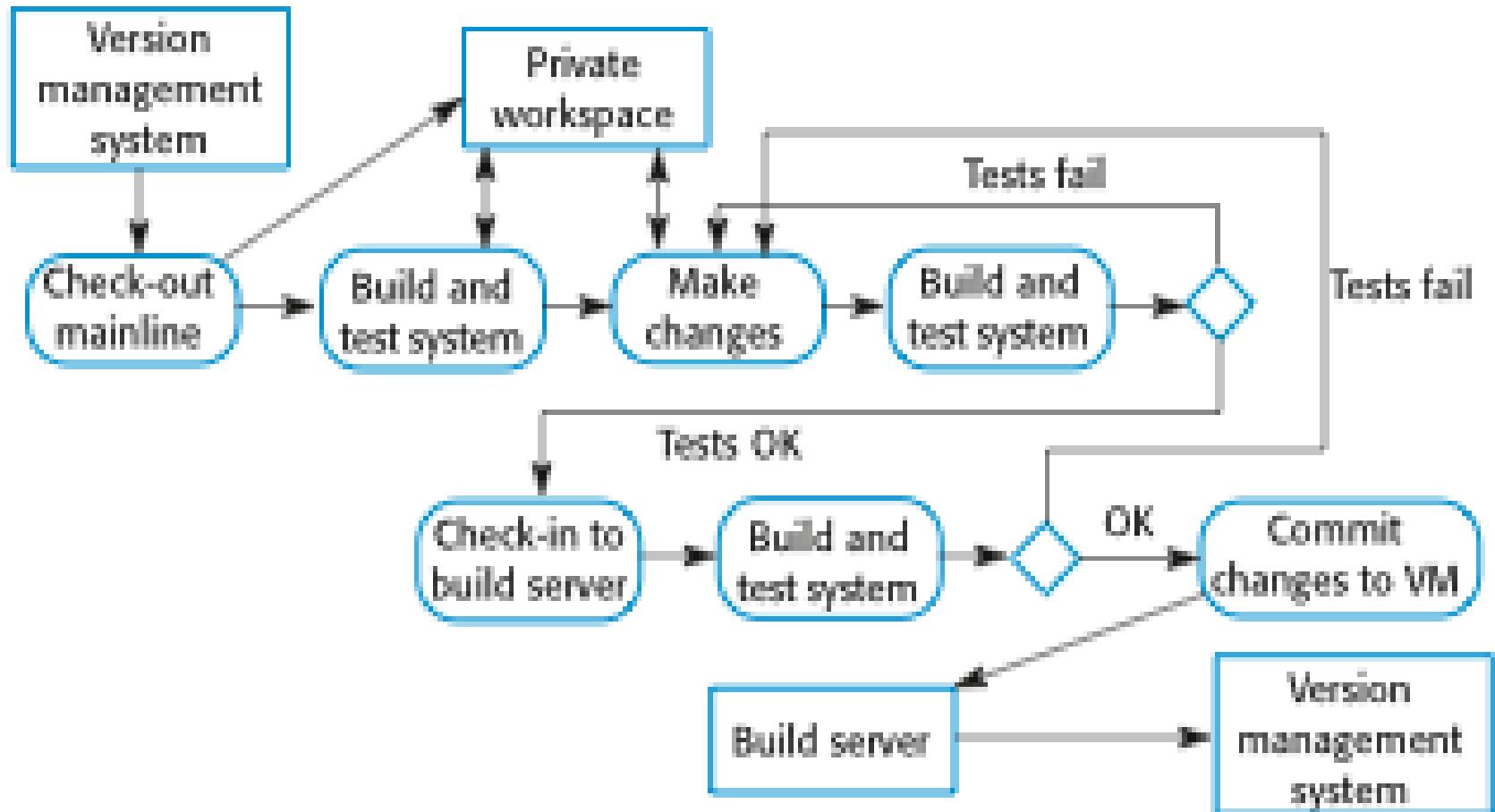
- Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.

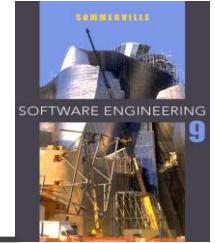
❖ Checksums

- When you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.



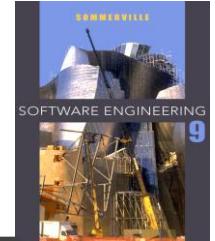
Continuous integration





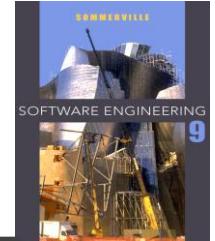
Daily building

- ✧ The development organization sets a delivery time (say 2 p.m.) for system components.
 - If developers have new versions of the components that they are writing, they must deliver them by that time.
 - A new version of the system is built from these components by compiling and linking them to form a complete system.
 - This system is then delivered to the testing team, which carries out a set of predefined system tests
 - Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.



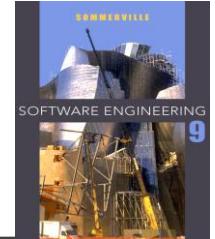
Release management

- ✧ A system release is a version of a software system that is distributed to customers.
- ✧ For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.
- ✧ For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.



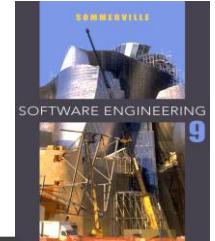
Release tracking

- ❖ In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.
- ❖ When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- ❖ This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines.
 - Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.



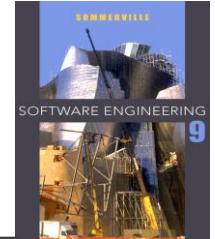
Release reproduction

- ✧ To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- ✧ You must keep copies of the source code files, corresponding executables and all data and configuration files.
- ✧ You should also record the versions of the operating system, libraries, compilers and other tools used to build the software.



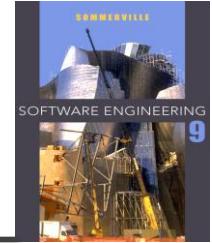
Release planning

- ✧ As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- ✧ Release timing
 - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
 - If system releases are too infrequent, market share may be lost as customers move to alternative systems.



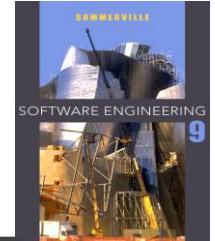
Release components

- ✧ As well as the executable code of the system, a release may also include:
 - configuration files defining how the release should be configured for particular installations;
 - data files, such as files of error messages, that are needed for successful system operation;
 - an installation program that is used to help install the system on target hardware;
 - electronic and paper documentation describing the system;
 - packaging and associated publicity that have been designed for that release.



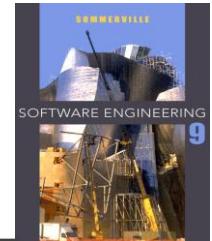
Factors influencing system release planning

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law (see Chapter 9)	This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance.



Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Customer change proposals	For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.



Key points

- ✧ System building is the process of assembling system components into an executable program to run on a target computer system.
- ✧ Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- ✧ System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.