

Software Engineering -1

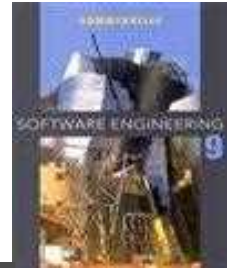
Dr. Mohammad Ahmad

Textbooks:

- 1. *Software Engineering*, by *Sommerville, Ian*, 9th Edition, or Above.**
- 2. *Software Engineering: A Practitioner's Approach*.
Roger S. Pressman. 6th edition or Above.**

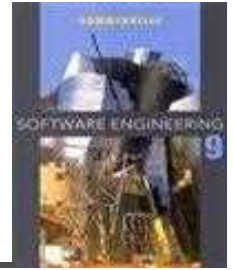
Lecture 1

Topics covered

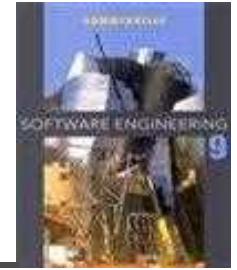


- ✧ Professional software development
 - What is meant by software engineering.
- ✧ Software engineering ethics
 - A brief introduction to ethical issues that affect software engineering.

Software engineering

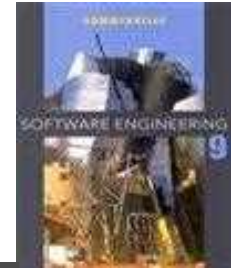


- ✧ The economies of ALL developed nations are dependent on software.
- ✧ More and more systems are software controlled
- ✧ Software engineering is concerned with theories, methods and tools for professional software development.
- ✧ Expenditure on software represents a significant fraction of GNP in all developed countries.



Software costs

- ✧ Software costs often dominate 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.



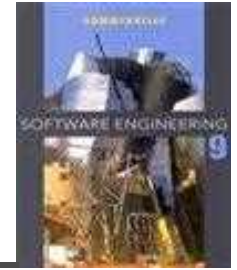
Software products

✧ 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

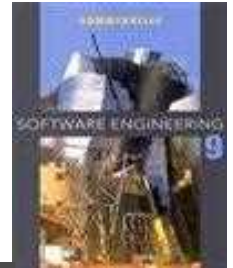
✧ Generic products

- 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

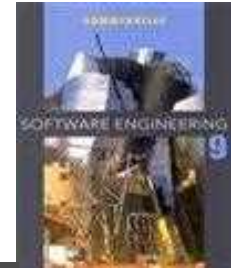
- The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.

Frequently asked questions about software engineering

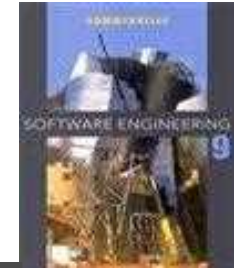


Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to 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.
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

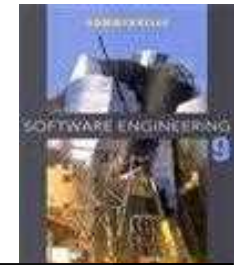


Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For 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.



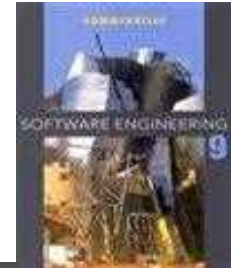
Essential attributes of good software

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.



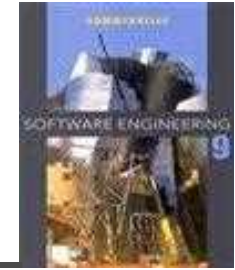
Characteristics of Software

Software	Hardware
logical system element	physical system element
developed/engineered	manufactured
Not ware out but deteriorate - no spare parts	ware out -yes, with spare parts
Usually custom-built	assembled from existing component



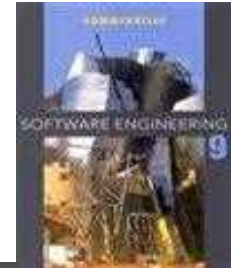
Software engineering

- ✧ 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.



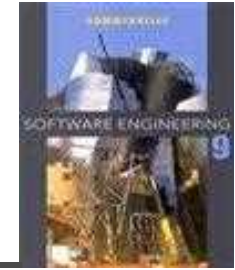
Importance of software engineering

- ✧ 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 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

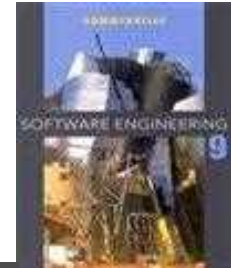
- Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

✧ 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

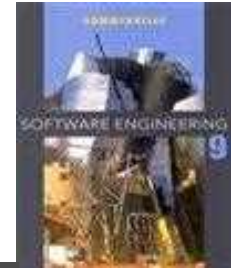
- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.



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, tools and procedures used depend on the type of application being developed, the requirements of the customer and the background of the development team.
 - Method: introduce a way to build SW
 - Tool: automatic, semi-auto support for methods
 - Procedure: define the sequence in which methods will be applied, the controls that help ensure quality and coordinate changes.

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.

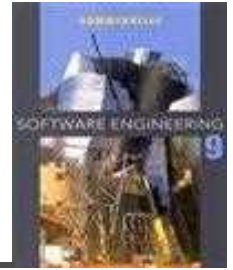
✧ Interactive transaction-based applications

- 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.

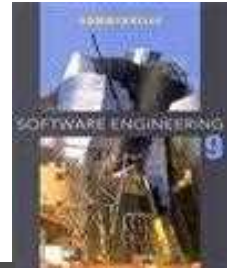
✧ Entertainment systems

- 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

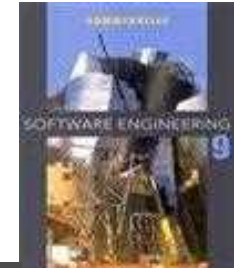


✧ Data collection systems

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

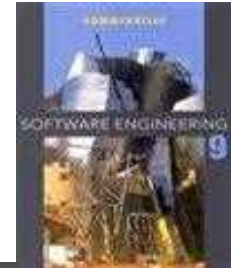
✧ Systems of systems

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



Software engineering fundamentals

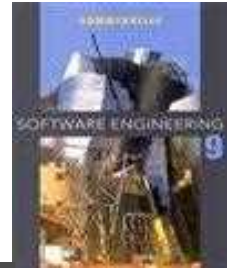
- ✧ Some fundamental principles apply to all types of software system, irrespective of the development techniques used:
 - Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.
 - Dependability and performance 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

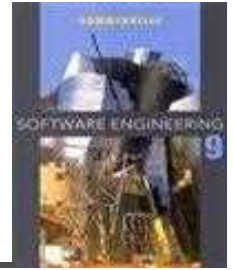
- ✧ The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- ✧ Web services allow application functionality to be accessed over the web.
- ✧ Cloud computing is an approach to the provision of computer services where applications run remotely on the 'cloud'.
 - Users do not buy software buy pay according to use.

Web software engineering



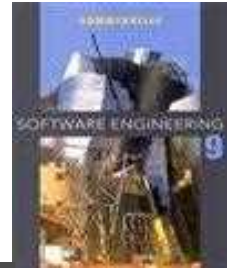
- ✧ Software reuse is the dominant approach for constructing web-based systems.
 - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- ✧ Web-based systems should be developed and delivered incrementally.
 - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- ✧ User interfaces are constrained by the capabilities of web browsers.
 - Technologies such as AJAX allow rich interfaces to be created within a web browser but are still difficult to use. Web forms with local scripting are more commonly used.

Web-based software engineering



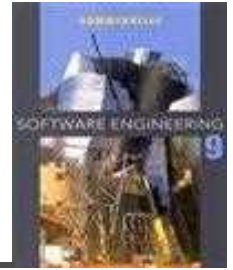
- ✧ Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- ✧ The fundamental ideas of software engineering, discussed in the previous section, apply to web-based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

What is CASE (Computer-Aided Software Engineering)

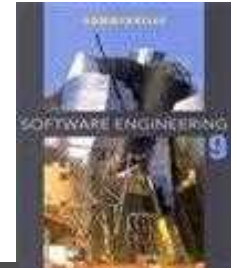


- Software systems that are intended to provide automated support for software process activities.
- CASE systems are often used for method support.
- Upper-CASE: Tools to support the early process activities of requirements and design;
- Lower-CASE: Tools to support later activities such as programming, debugging and testing.

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.



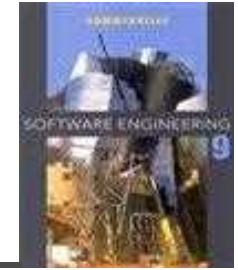
Issues of professional responsibility

✧ 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

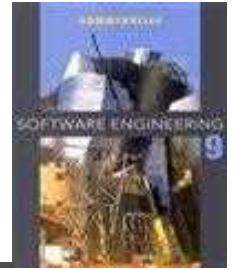
✧ Intellectual property rights

- 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

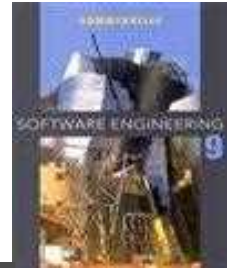
- Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Rationale for the code of ethics

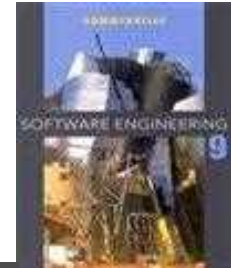


- *Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance 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.*

Additional software functionality

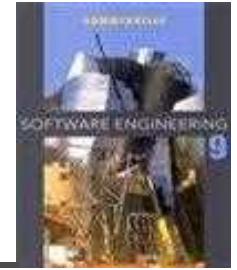


- ✧ Monitor the instruments, power and communication hardware and report faults to the management system.
- ✧ Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- ✧ Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.



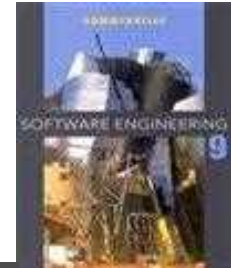
Key points

- ✧ 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.



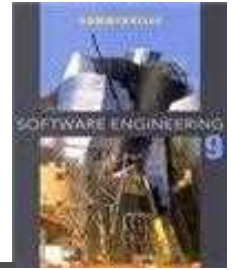
Key points

- ✧ 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.



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.



Chapter 3 – Agile Software Development

Lecture 3

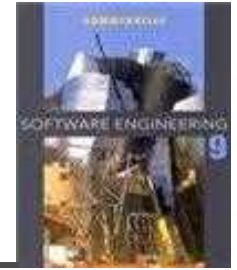
Dr. Mohammad Ahmad

Ph.D. in Software Engineering

MCA Master of Computer Application

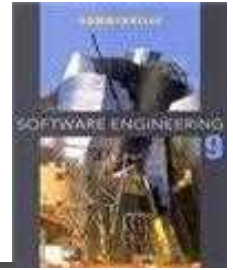
MBA-IT Master of Business Administration in Information Technology

Topics covered



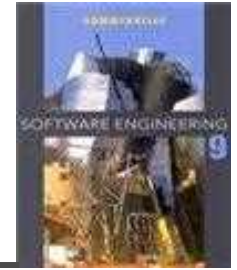
- ✧ Agile methods
- ✧ Plan-driven and agile development
- ✧ Extreme programming
- ✧ Agile project management
- ✧ Scaling agile methods

Rapid software development



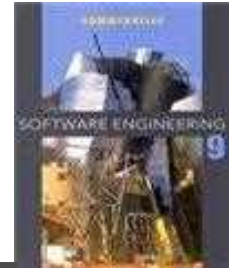
- ✧ Rapid development and delivery is now often the most important requirement for software systems
 - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
 - Software has to evolve quickly to reflect changing business needs.
- ✧ Rapid software development
 - Specification, design and implementation are inter-leaved
 - System is developed as a series of versions with stakeholders involved in version evaluation
 - User interfaces are often developed using an IDE and graphical toolset.

Agile methods



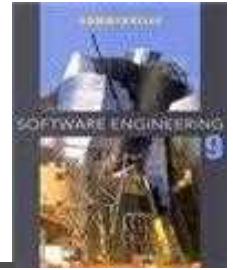
- ✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
 - Focus on the code rather than the design
 - Are based on an iterative approach to software development
 - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ✧ The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

The principles of agile methods

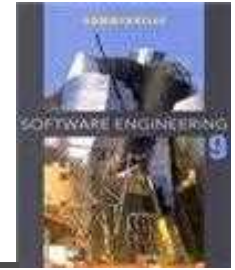


Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability



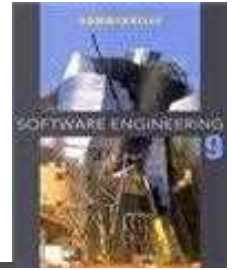
- ✧ Product development where a software company is developing a small or medium-sized product for sale.
- ✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- ✧ Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.



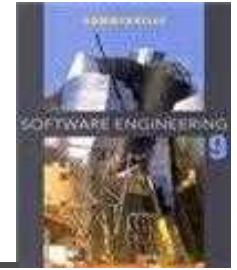
Problems with agile methods

- ✧ It can be difficult to keep the interest of customers who are involved in the process.
- ✧ Team members may be unsuited to the intense involvement that characterizes agile methods.
- ✧ Prioritizing changes can be difficult where there are multiple stakeholders.
- ✧ Maintaining simplicity requires extra work.
- ✧ Contracts may be a problem as with other approaches to iterative development.

Agile methods and software maintenance



- ✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- ✧ Two key issues:
 - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
 - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.



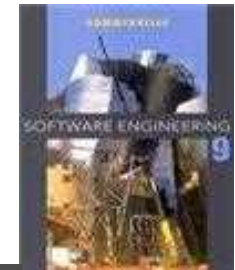
Plan-driven and agile development

✧ Plan-driven development

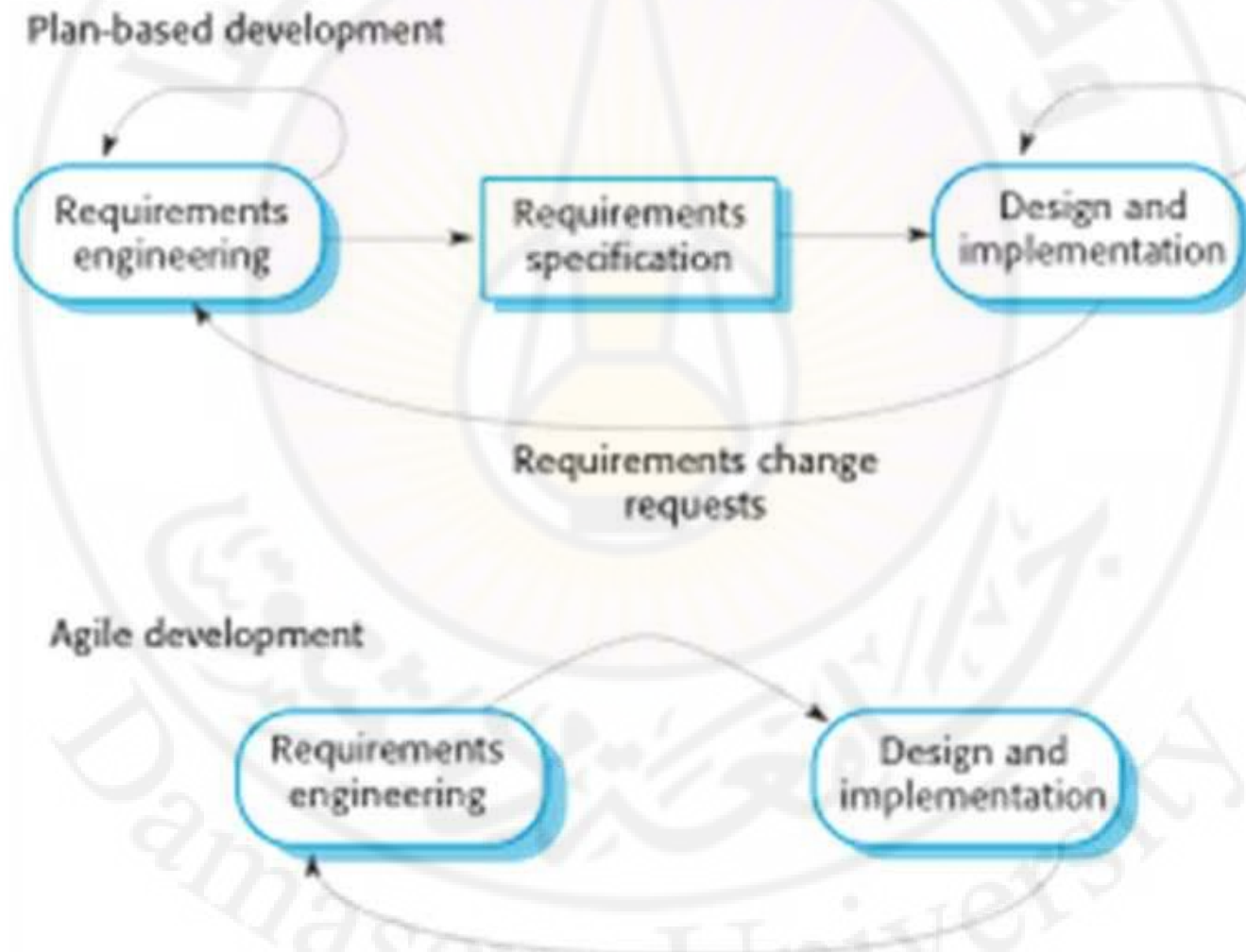
- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

✧ Agile development

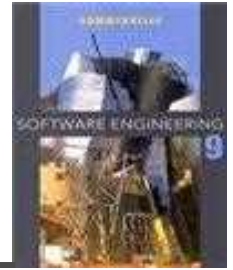
- Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.



Plan-driven and agile specification

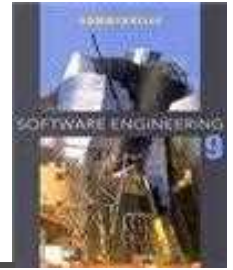


Technical, human, organizational issues



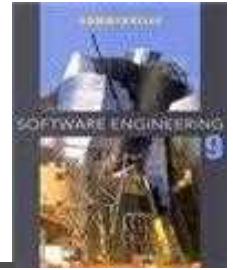
- ✧ Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
 - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

Technical, human, organizational issues

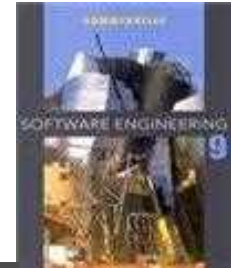


- What type of system is being developed?
 - Plan-driven approaches may be required for systems that require a lot of analysis before implementation (e.g. real-time system with complex timing requirements).
- What is the expected system lifetime?
 - Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.
- What technologies are available to support system development?
 - Agile methods rely on good tools to keep track of an evolving design
- How is the development team organized?
 - If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.

Technical, human, organizational issues



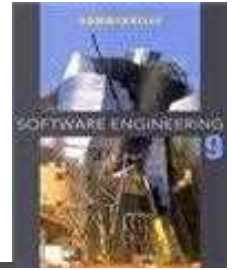
- Are there cultural or organizational issues that may affect the system development?
 - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good are the designers and programmers in the development team?
 - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code
- Is the system subject to external regulation?
 - If a system has to be approved by an external regulator (e.g. the FAA approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.



Extreme programming

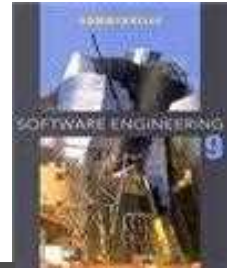
- ✧ Perhaps the best-known and most widely used agile method.
- ✧ Extreme Programming (XP) takes an 'extreme' approach to iterative development.
 - New versions may be built several times per day;
 - Increments are delivered to customers every 2 weeks;
 - All tests must be run for every build and the build is only accepted if tests run successfully.

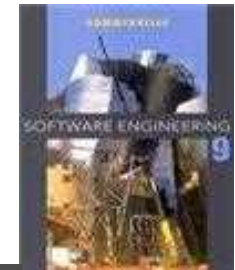
XP and agile principles



- ✧ Incremental development is supported through small, frequent system releases.
- ✧ Customer involvement means full-time customer engagement with the team.
- ✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ✧ Change supported through regular system releases.
- ✧ Maintaining simplicity through constant refactoring of code.

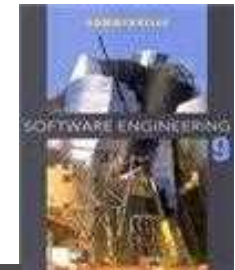
The extreme programming release cycle





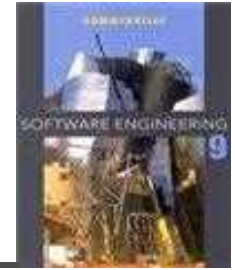
Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.



Extreme programming practices (b)

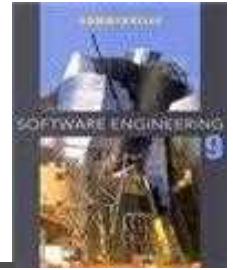
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.



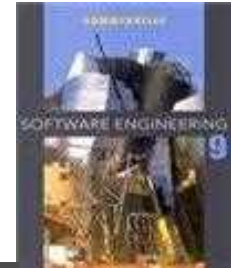
Requirements scenarios

- ✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- ✧ User requirements are expressed as scenarios or user stories.
- ✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- ✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

XP and change

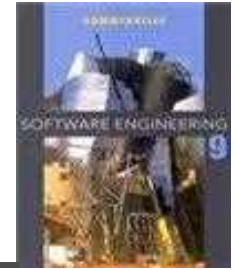


- ✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.



Refactoring

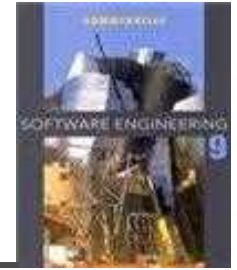
- ✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ✧ This improves the understandability of the software and so reduces the need for documentation.
- ✧ Changes are easier to make because the code is well-structured and clear.
- ✧ However, some changes requires architecture refactoring and this is much more expensive.



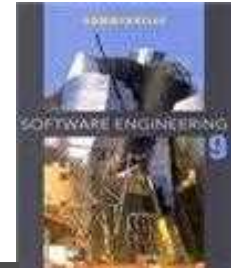
Examples of refactoring

- ✧ Re-organization of a class hierarchy to remove duplicate code.
- ✧ Tidying up and renaming attributes and methods to make them easier to understand.
- ✧ The replacement of inline code with calls to methods that have been included in a program library.

Testing in XP

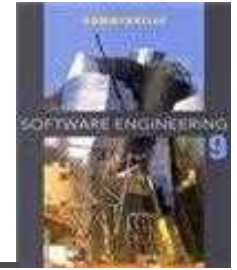


- ✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
- ✧ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.



Test-first development

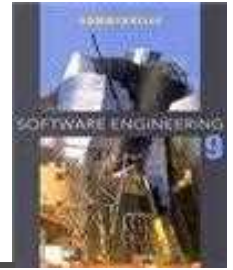
- ✧ Writing tests before code clarifies the requirements to be implemented.
- ✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as Junit.
- ✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.



Customer involvement

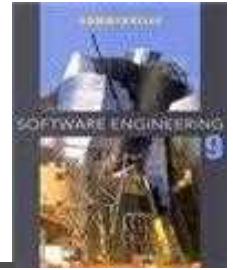
- ✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- ✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test automation

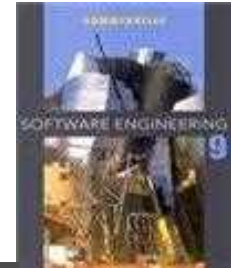


- ✧ Test automation means that tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

XP testing difficulties

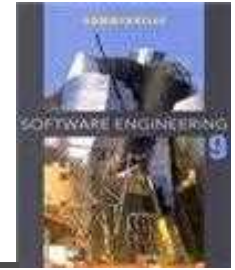


- ✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ✧ Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.



Pair programming

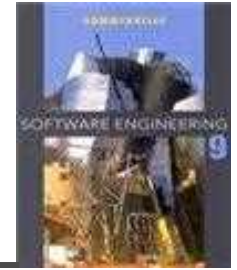
- ✧ In XP, programmers work in pairs, sitting together to develop code.
- ✧ This helps develop common ownership of code and spreads knowledge across the team.
- ✧ It serves as an informal review process as each line of code is looked at by more than 1 person.
- ✧ It encourages refactoring as the whole team can benefit from this.
- ✧ Measurements suggest that development productivity with pair programming is similar to that of two people working independently.



Pair programming

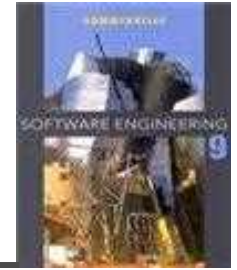
- ✧ In pair programming, programmers sit together at the same workstation to develop the software.
- ✧ Pairs are created dynamically so that all team members work with each other during the development process.
- ✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- ✧ Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

Advantages of pair programming



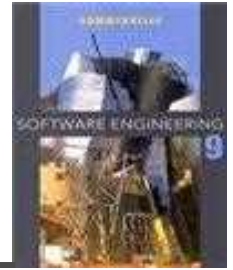
- ✧ It supports the idea of collective ownership and responsibility for the system.
 - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- ✧ It acts as an informal review process because each line of code is looked at by at least two people.
- ✧ It helps support refactoring, which is a process of software improvement.
 - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Agile project management



- ✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- ✧ The standard approach to project management is plan-driven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.
- ✧ Agile project management requires a different approach, which is adapted to incremental development and the particular strengths of agile methods.

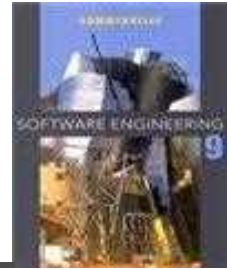
Scrum



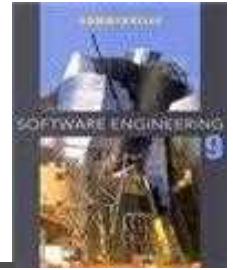
- ✧ The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
 - The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
 - This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
 - The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.



The Scrum process

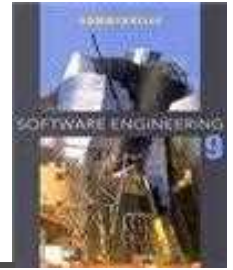


The Sprint cycle



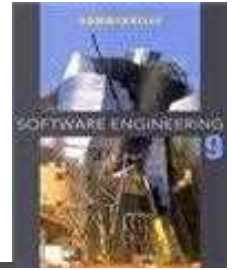
- ✧ Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.

The Sprint cycle

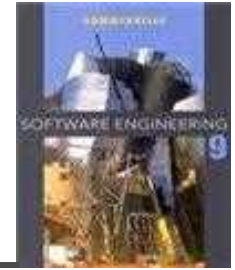


- ✧ Once these are agreed, the team organize themselves to develop the software. During this stage the team is isolated from the customer and the organization, with all communications channelled through the so-called 'Scrum master'.
- ✧ The role of the Scrum master is to protect the development team from external distractions.
- ✧ At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

Teamwork in Scrum

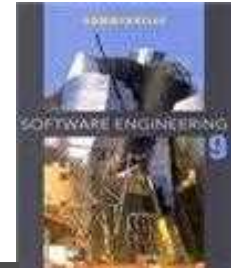


- ✧ The 'Scrum master' is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- ✧ The whole team attends short daily meetings where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
 - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.



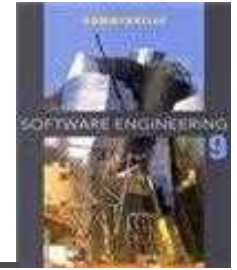
Scrum benefits

- ✧ The product is broken down into a set of manageable and understandable chunks.
- ✧ Unstable requirements do not hold up progress.
- ✧ The whole team have visibility of everything and consequently team communication is improved.
- ✧ Customers see on-time delivery of increments and gain feedback on how the product works.
- ✧ Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.



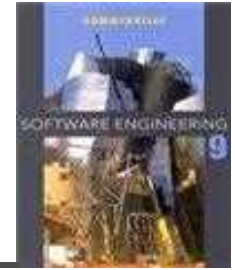
Scaling agile methods

- ✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- ✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- ✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.



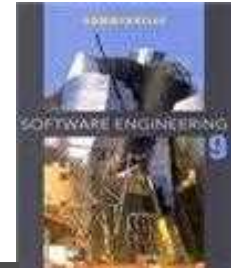
Large systems development

- ✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- ✧ Large systems are 'brownfield systems', that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.
- ✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.



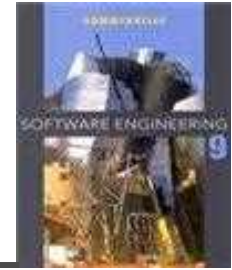
Large system development

- ✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- ✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- ✧ Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.



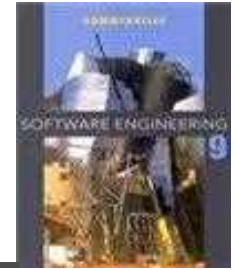
Scaling out and scaling up

- ✧ ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ✧ ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- ✧ When scaling agile methods it is essential to maintain agile fundamentals
 - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.



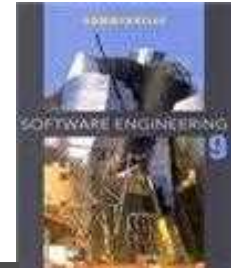
Scaling up to large systems

- ✧ For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation
- ✧ Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress.
- ✧ Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.



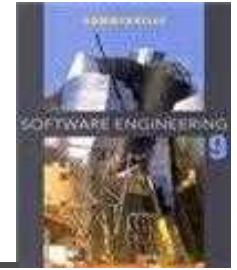
Scaling out to large companies

- ✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- ✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- ✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- ✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.



Key points

- ✧ Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads and producing high-quality code. They involve the customer directly in the development process.
- ✧ The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.
- ✧ Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement and customer participation in the development team.



Key points

- ✧ A particular strength of extreme programming is the development of automated tests before a program feature is created. All tests must successfully execute when an increment is integrated into a system.
- ✧ The Scrum method is an agile method that provides a project management framework. It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- ✧ Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation.

The background of the slide features a large, faint watermark of the Damascus University logo. The logo is circular, with the university's name in Arabic at the top and 'Damascus University' in English at the bottom. In the center is a stylized emblem with a central point and radiating lines.

Object Oriented Design, Design Pattern & Implementation Issues

Dr. Mohammad Ahmad

Lecture 8

Objectives

1. Outline the concepts and principles underpinning design
 - Abstraction and Refinement
 - Modularity
 - Cohesion and Coupling
 - Information Hiding
2. Present some criteria for good design
3. Describe the Object Oriented design process:
 - Subsystem Design
 - Class and Object Design
 - Message Design
 - Responsibilities Design
4. Detail the UML models produced during OO design
 - Specification Class Diagrams
 - Package Diagrams

Overview of Design

- What is it? A meaningful engineering representation of something that is to be built.
- Who does it? Software engineers with a variety of skills, ranging from human ergonomics to computer architecture
- Why is it important? A house would never be built without a blueprint. Why should software? Without design the system may fail with small changes, is difficult to test and cannot be assessed for quality
- What is the work product? A design specification

Designing Quality Software

- Design is the stage where quality is instilled
- Design is assessed by formal review or walkthrough
- **Characteristics of a good design:**
 1. Must implement explicit requirements and accommodate implicit requirements
 2. Must be a readable and understandable guide for coding and testing
 3. Should provide a complete picture of the software from an implementation perspective

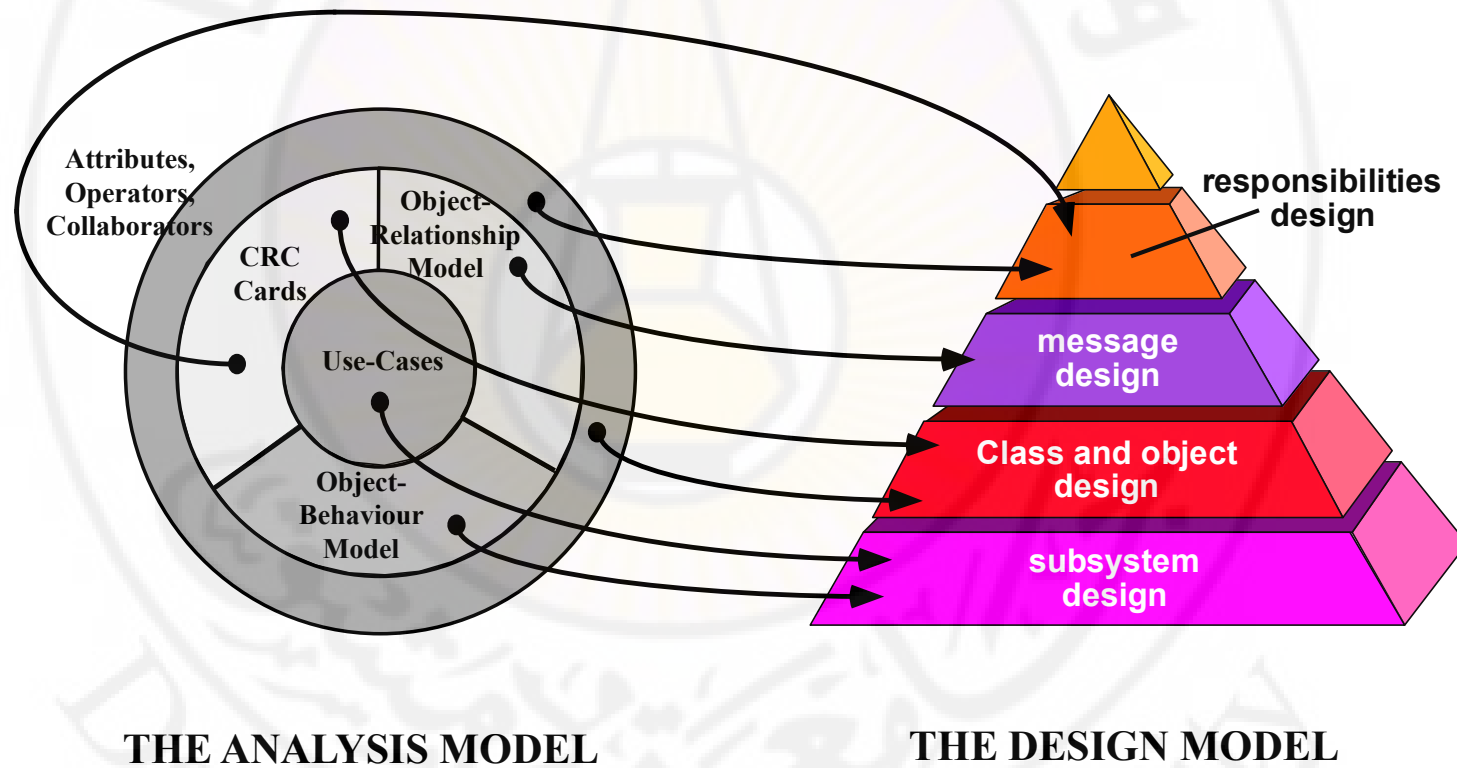
Generic Design Process

- All design methods have the following characteristics:
 - A mechanism for the translation of the analysis model into a design representation
 - A notation for representing functional components and their interfaces
 - Heuristics for refinement and partitioning
 - Guidelines for quality assessment

The Challenge of OO Design

- To create re-usable OO software a designer must:
 - Find pertinent objects
 - Factor them into classes of the right granularity
 - Define class interfaces and inheritance hierarchies
- The design should be specific to the problem at hand but general enough to address future problems and requirements.
- All design methods strive for:
 - Abstraction
 - Information hiding
 - High cohesion; Low coupling
 - Modularity
- But only OOD can achieve all four without complexity or compromise.

OO Design



Design Summary

Process	Model Output
1. Subsystem Design: partition the system into components	A Package Diagram
2. Class and Object Design: define class hierarchies	Specification Class Diagram
3. Message Design: convert the object-relationship model into a set of class messages	Message Descriptions
4. Responsibility Design: specify the internal structure of classes	Specification Class Diagram with full attribute and method.



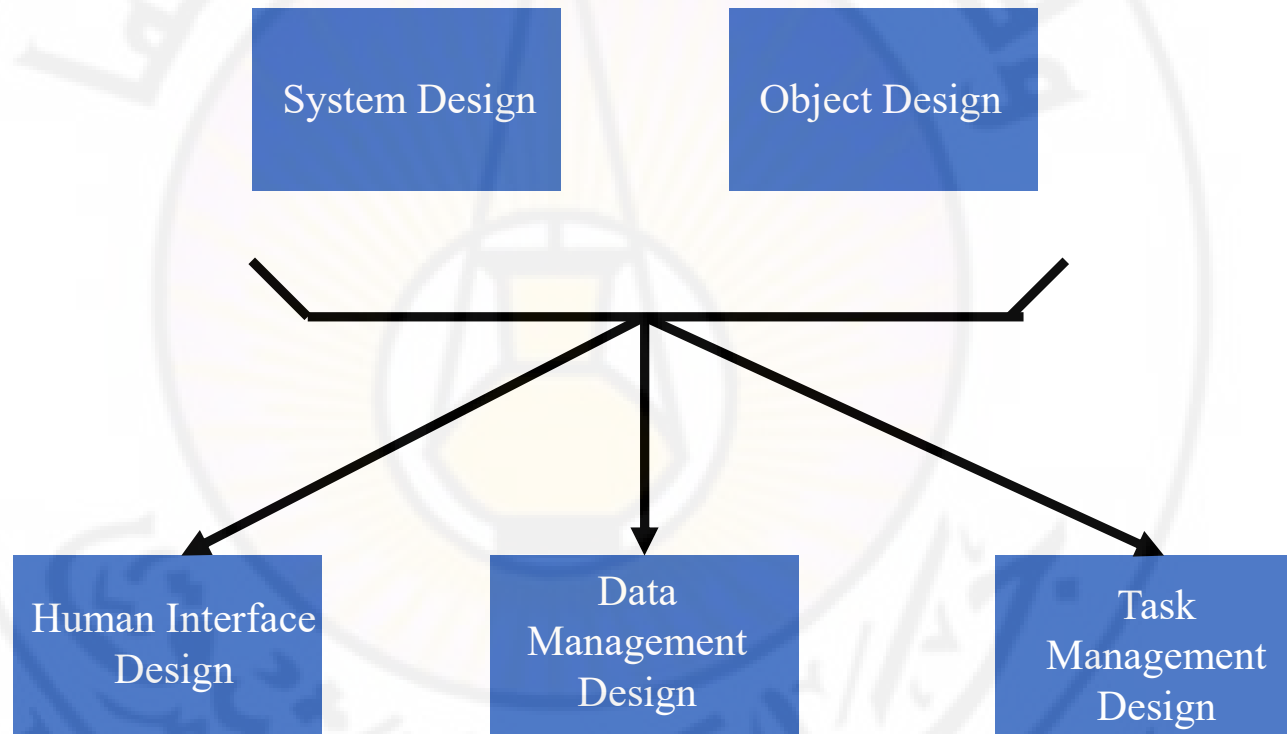
OO Design Criteria

- **Decomposability**
 - The design method helps the designer to decompose a large problem into subproblems that are easier to solve
- **Composability**
 - The design method ensures that program components (classes), once designed and built, can be reused to create other systems
- **Understandability**
 - The ease with which a program component can be understood without reference to other information or other modules
- **Continuity**
 - The ability to make small changes in a program and have these changes manifest themselves with corresponding changes in just one or a very few classes
- **Protection**
 - An architectural characteristic that will reduce the propagation of side effects if an error does occur in a given class

Generic Components for OOD

- Problem Domain Component
 - The subsystems that are responsible for implementing customer requirements directly
- Human Interaction Component
 - The subsystems that implement the user interface (this included reusable GUI subsystems)
- Task Management Component
 - The subsystems that control concurrent tasks that may be packaged within a subsystem or among different subsystems
- Data Management Component
 - The subsystem that is responsible for the storage and retrieval of persistent data

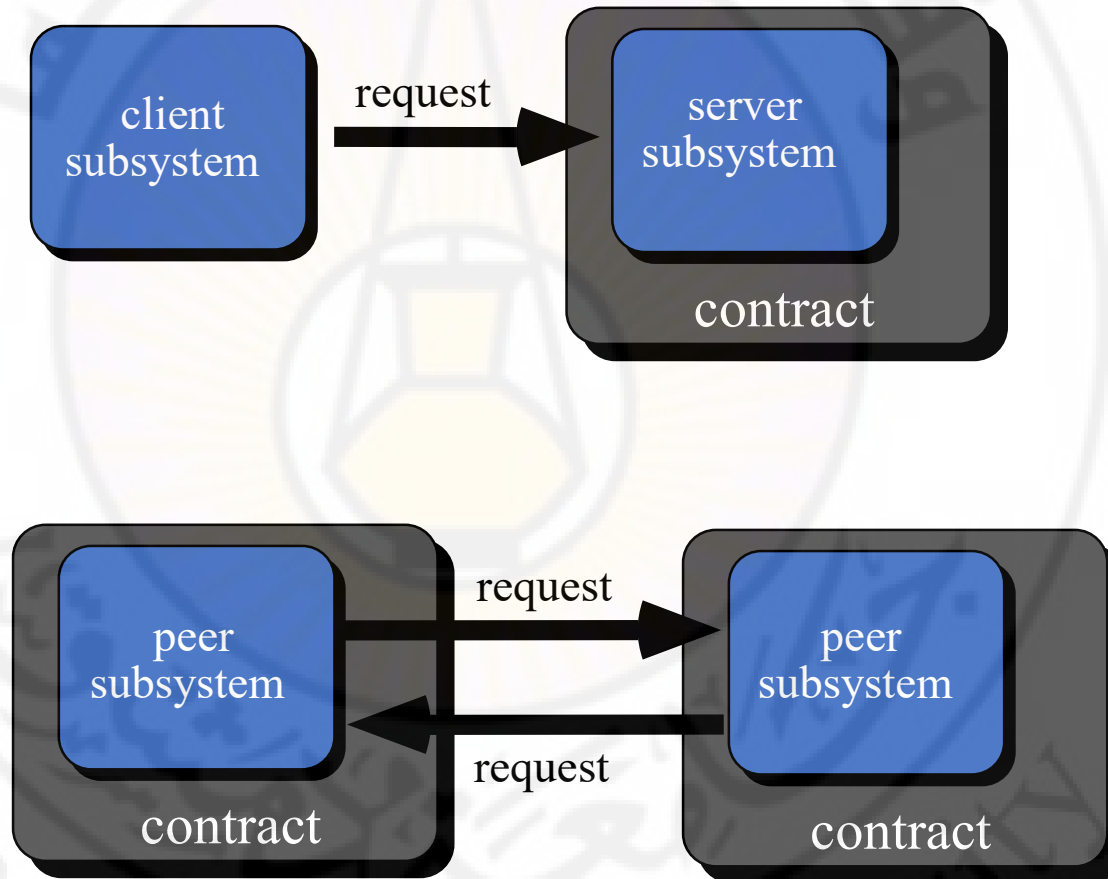
OOD Process



System Design Process

- Represent the conceptual software architecture.
- Activities (from Rambaugh et al.):
 - A. Partitioning - Partition the analysis model into subsystems
 - B. Concurrency - Identify concurrency that is dictated by the problem and allocate subsystems to processors and tasks
 - C. User Interface - Develop a design for the user interface
 - D. Data Management - Choose a basic strategy for implementing data management. Identify global resources and the control mechanisms required to access them
 - E. Task Management - Design an appropriate control mechanism for the system, including task management
 - F. Intersubsystem Communication - Define the collaborations that exist between subsystems
 - G. Consider how boundary conditions should be handled
 - H. Review and consider trade-offs.

Subsystem Collaboration Types



Subsystem Collaboration Table

Contract	Type	Collaborators	Class(es)	Operation(s)	Message Format

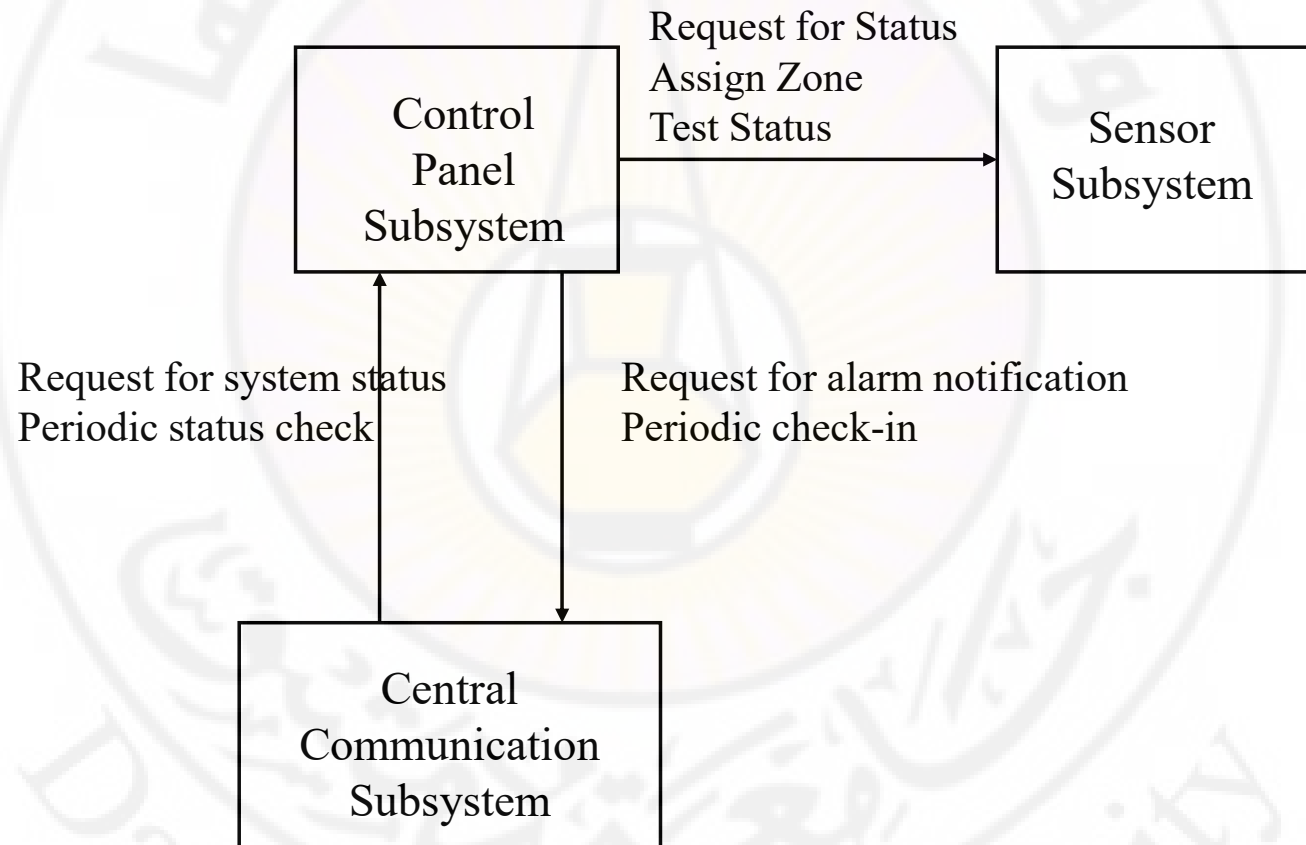
The diagram shows three arrows pointing upwards from labels below to the bottom of the table columns. The first arrow points to the 'Contract' column. The second arrow points to the 'Collaborators' column. The third arrow points to the 'Class(es)' column. The fourth arrow points to the 'Operation(s)' column. The fifth arrow points to the 'Message Format' column.

client/server
or peer to peer

other
subsystems
party to the
contract

subsystem classes (+ relevant
methods and formats) to support
contract services

Example: Subsystem Collaboration Diagram

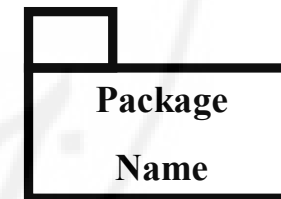


UML Outputs

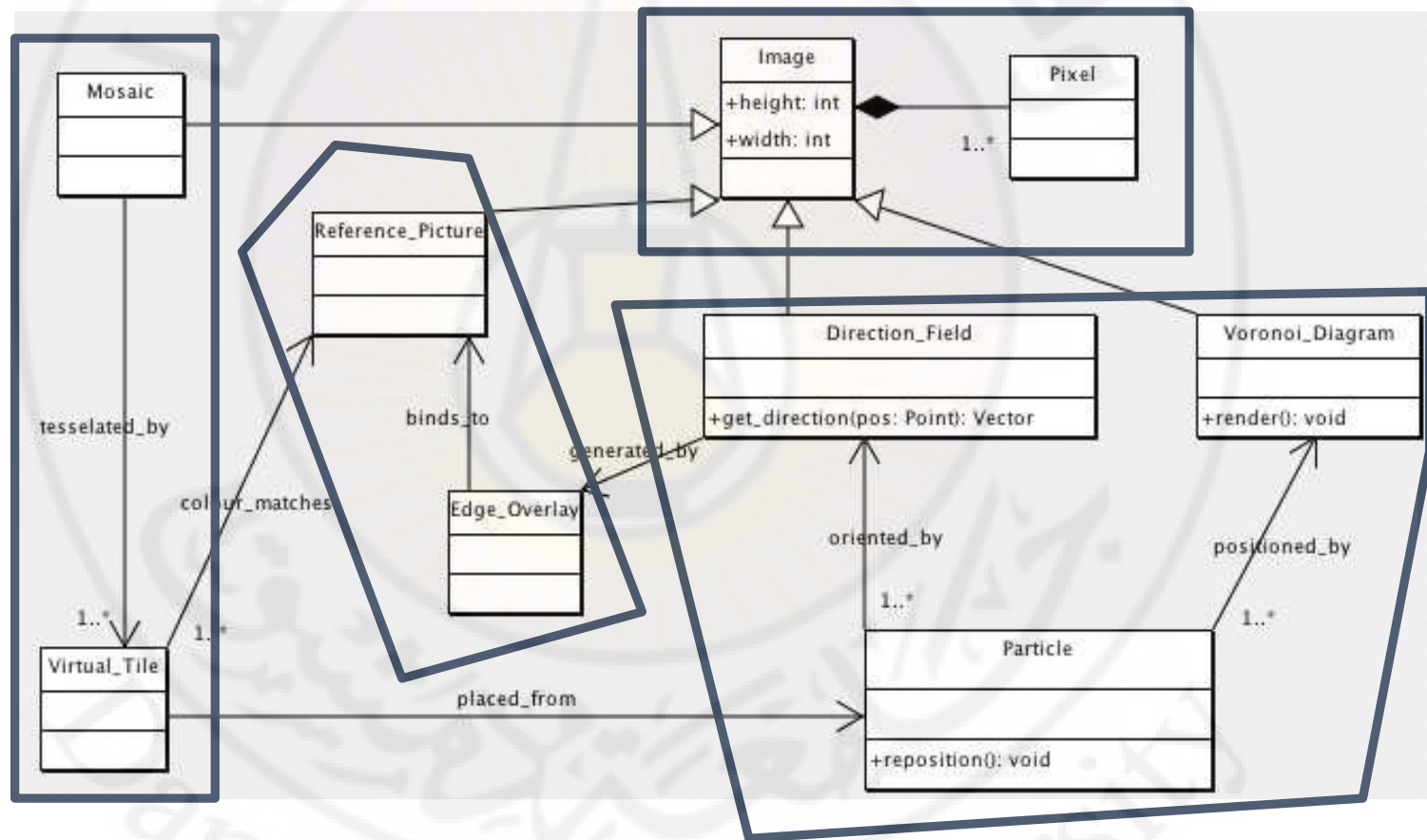
- The system design process expands on models produced during analysis
- Package diagrams (high-level class diagrams containing only package references) are popular for large complex systems
- More detail is added to the Conceptual Class Diagrams to create Specification Class Diagrams
- Remember to use the principles of Design:
 - Refinement and Abstraction
 - Modularity driven by Coupling and Cohesion
 - Information Hiding

UML Package Diagrams

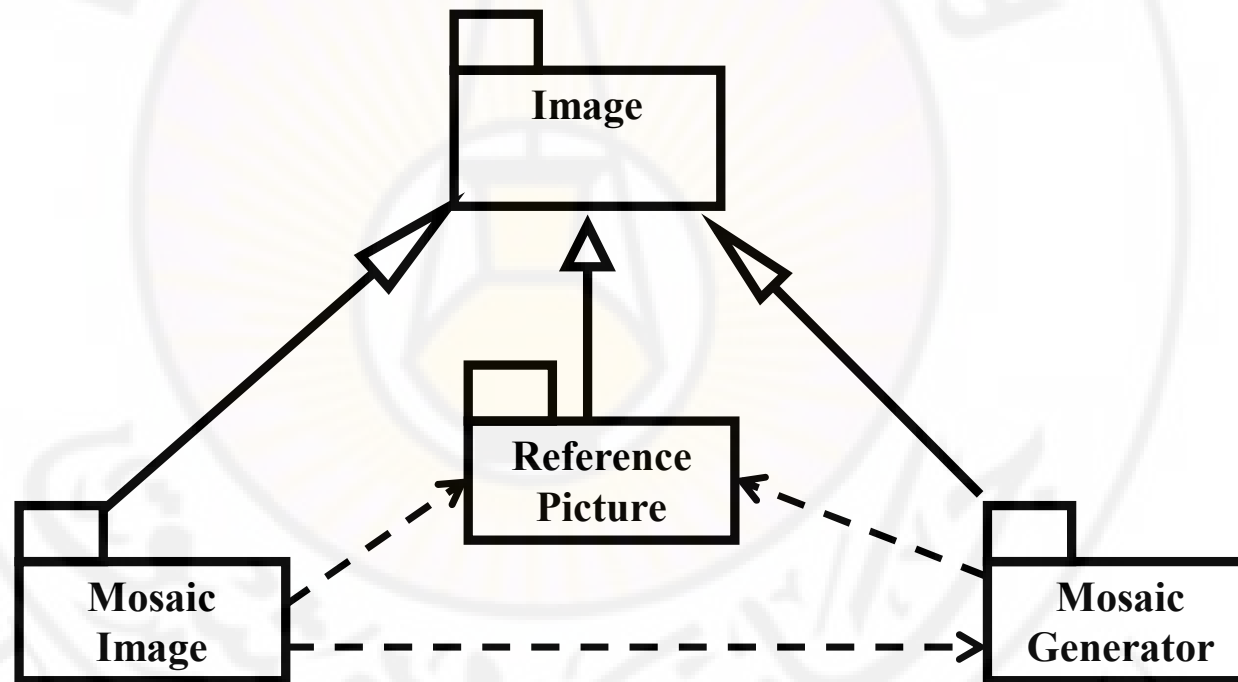
- Mechanism for dividing and grouping model elements (e.g. classes, use-cases)
- Group into Packages:
 1. Classes in the same inheritance hierarchy
 2. Classes related via aggregation or composition
 3. Classes with lots of collaboration
- Notation:
 - Package (a named collection of classes)
 - Dependency (reflects a relationship between any two classes in the participating packages)



Example: Packaging a Class Diagram



Example: Final Package Diagram



UML Specification Class Diagrams

- Detail is added to the attribute and methods areas of the class
- The class diagram may also be refined
- Attributes:

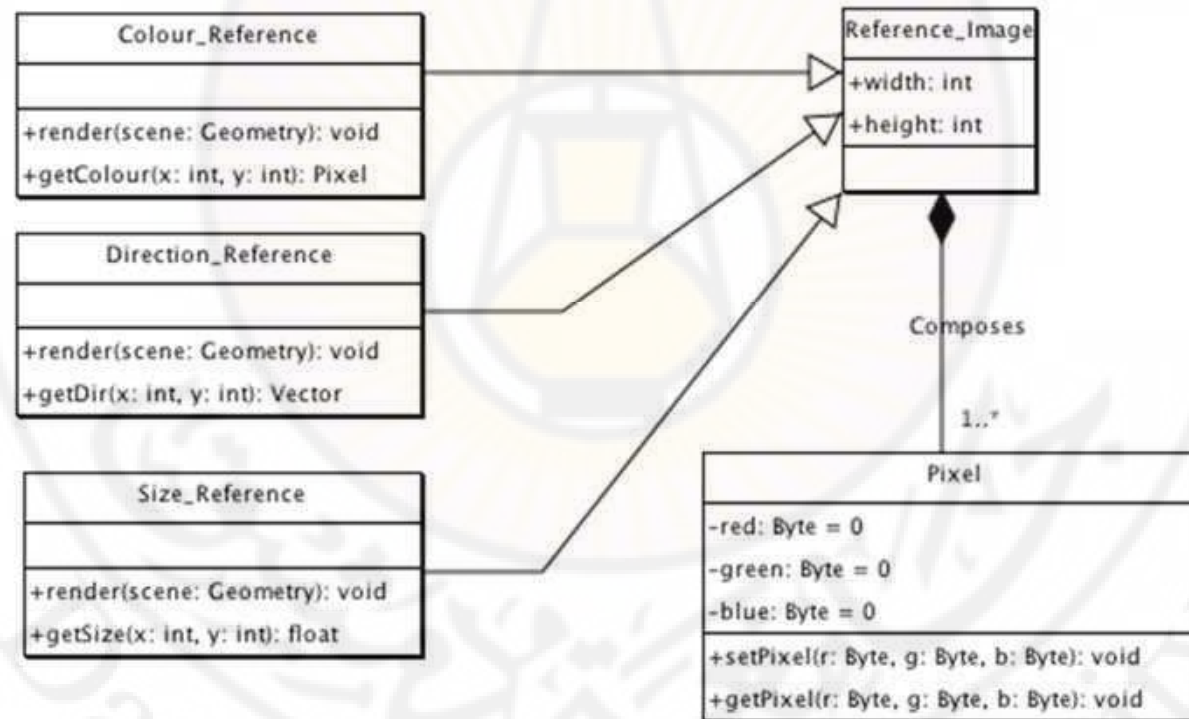
- Syntax: *visibility name : type = default-value*
- *visibility* is + (public) # (protected) or – (private)
- *name* is a string
- *type* language-dependent specification
- *default-value* the initial value of the attribute
- Example: # numberPlate : string = “CA 101 010”

Name
Attributes
Methods

Operation Syntax

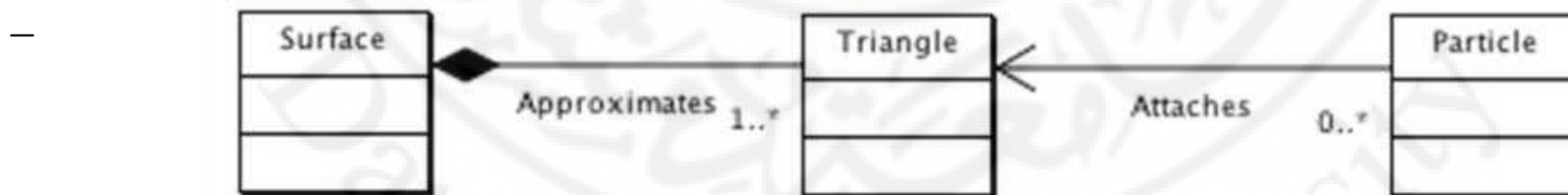
- Operations (Methods):
 - Syntax: *visibility name (parameter-list) : return-type-expression*
 - *visibility* and *name* are the same as for attribute syntax
 - *parameter-list* contains optional arguments with the same syntax as attributes (separated by commas)
 - *return-type-expression* is an optional language-dependent specification
 - Example: + latestAmountOf (PhenomenonType value) : Quantity

Example: Specification Class Diagram

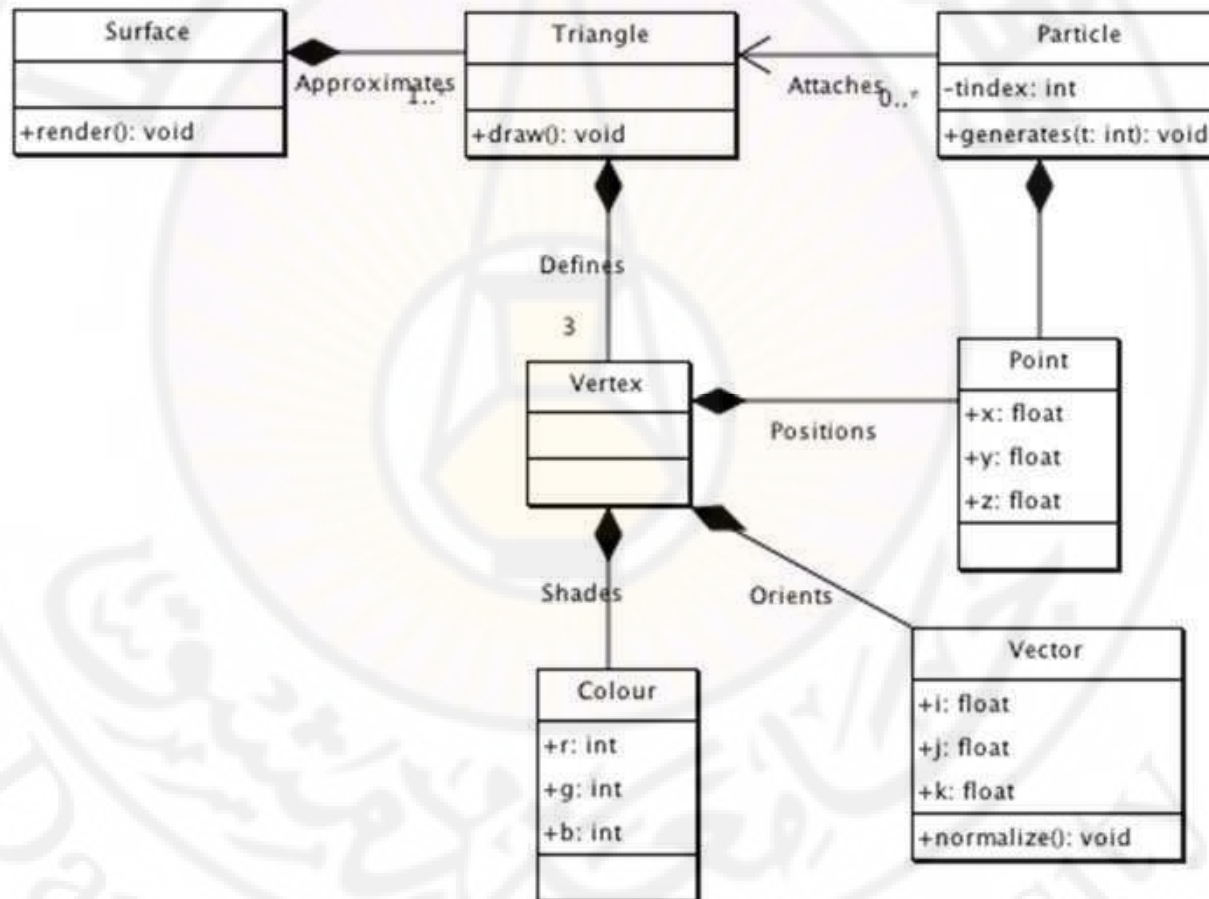


Exercise: Specification Class Diagram

- Refine a conceptual class diagram to produce a specification class diagram given the following information:
 - A triangle consists of three vertices each with a point position, vector direction and r-g-b colour. The vector can be normalized (converted to unit length).
 - A particle has a point position and a triangle index. It must be able to generate a random position given a triangle index.
 - A triangle can be drawn and a surface rendered (by making multiple calls to the draw method)



Solution: Specification Class Diagram



Design and implementation

- Software design and implementation activities are invariably interleaved.
 - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
 - Implementation is the process of realizing the design as a program.

Build or buy

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

Good Signs of OO Thinking

✧ Short methods

- Simple method logic

✧ Few instance variables

✧ Clear object responsibilities

- State the purpose of the class in one sentence
- No super-intelligent objects
- No manager objects

Some Principles

✧ The Dependency Inversion Principle

- Depend on abstractions, not concrete implementations
 - Write to an interface, not a class

✧ The Interface Segregation Principle

- Many small interfaces are better than one “fat” one

✧ The Acyclic Dependencies Principle

- Dependencies between package must not form cycles.
 - Break cycles by forming new packages

The Open-Closed Principle

- ❖ Software entities should be *open for extension* but *closed for modifications*.
- ❖ Design classes and packages so their functionality can be extended without modifying the source code

Packages, Modules and other

✧ The Common Closure Principle

- Classes that change together, belong together
 - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.

✧ The Common Reuse Principle

- Classes that aren't reused together don't belong together
 - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

What is Refactoring?

- The process of *changing a software system* in such a way that it *does not alter the external behavior* of the code, yet *improves its internal structure*.
 - Fowler, et al., Refactoring, 1999.

Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Why Refactor?

✧ The reality:

- Extremely difficult to get the design “right” the first time
- Hard to fully understand the problem domain
- Hard to understand user requirements, even if the user does!
- Hard to know how the system will evolve in five years
- Original design is often inadequate
- System becomes brittle over time, and more difficult to change

✧ Refactoring helps you to

- Manipulate code in a safe environment (behavior preserving)
- Recreate a situation where evolution is possible
- Understand existing code

Rename Method — manual steps

✧ Do it yourself approach:

- Check that no method with the new name already exists in any subclass or superclass.
- Browse all the implementers (method definitions)
- Browse all the senders (method invocations)
- Edit and rename all implementers
- Edit and rename all senders
- Remove all implementers
- Test

✧ Automated refactoring is better !

The Law of Demeter

✧ ***“Do not talk to strangers”***

■ You should only send messages to:

- an argument passed to you
- an object you create
- self, super
- your class

✧ Don't send messages to objects returned from other message sends

Code Smells

“If it stinks, change it”

- Duplicated Code
 - Missing inheritance or delegation
- Long Method
 - Inadequate decomposition
- Large Class / God Class
 - Too many responsibilities
- Long Parameter List
 - Object is missing
- Type Tests
 - Missing polymorphism
- Shotgun Surgery
 - Small changes affect too many objects

Design patterns

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

Pattern elements

✧Name

- A meaningful pattern identifier.

✧Problem description.

✧Solution description.

- Not a concrete design but a template for a design solution that can be instantiated in different ways.

✧Consequences

- The results and trade-offs of applying the pattern.

Design Patterns

✧ Design Patterns document *recurrent solutions to design problems*

- They have *names*
 - Composite, Visitor, Observer...
- They are not components!
- Design Patterns entail *tradeoffs*
- Will be implemented in different ways in different contexts

Why Design Patterns?

- Smart
 - Elegant solutions that a novice would not think of
- Generic
 - Independent of specific system type, language
- Well-proven
 - Successfully tested in several systems
- Simple
 - Combine them for more complex solutions

Alert!!! Patterns are invading!



✧ Design Patterns are not “good” just because they are patterns

- It is just as important to understand when *not* to use a Design Pattern
- Every Design Pattern has tradeoffs
- Most Design Patterns will make your design *more complicated*
 - More classes, more indirections, more messages
- Don't use Design Patterns unless you really need them!

About Pattern Implementation

✧ Do not confuse *structure* and *intent*!

- Design Patterns document a *possible* implementation
 - Not a definitive one
- Design Patterns are about *intent* and *tradeoffs*



The Observer pattern

✧Name

- Observer.

✧Description

- Separates the display of object state from the object itself.

✧Problem description

- Used when multiple displays of state are needed.

✧Solution description

- See slide with UML description.

✧Consequences

- Optimisations to enhance display performance are impractical.

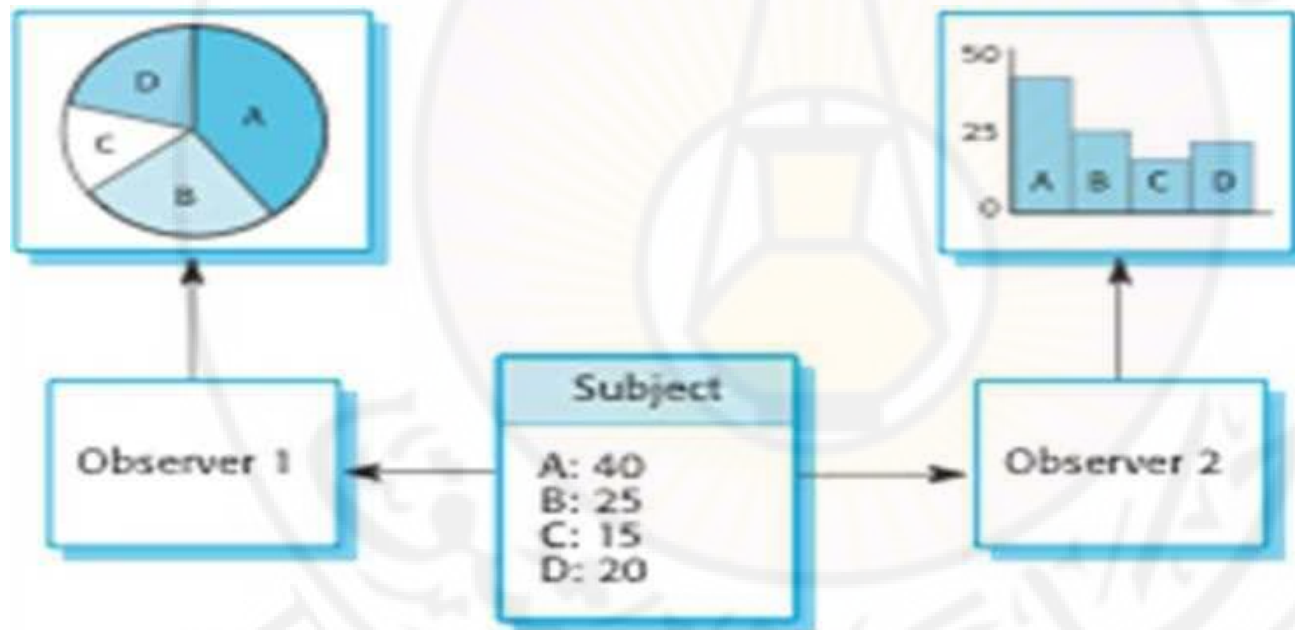
The Observer pattern (1)

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

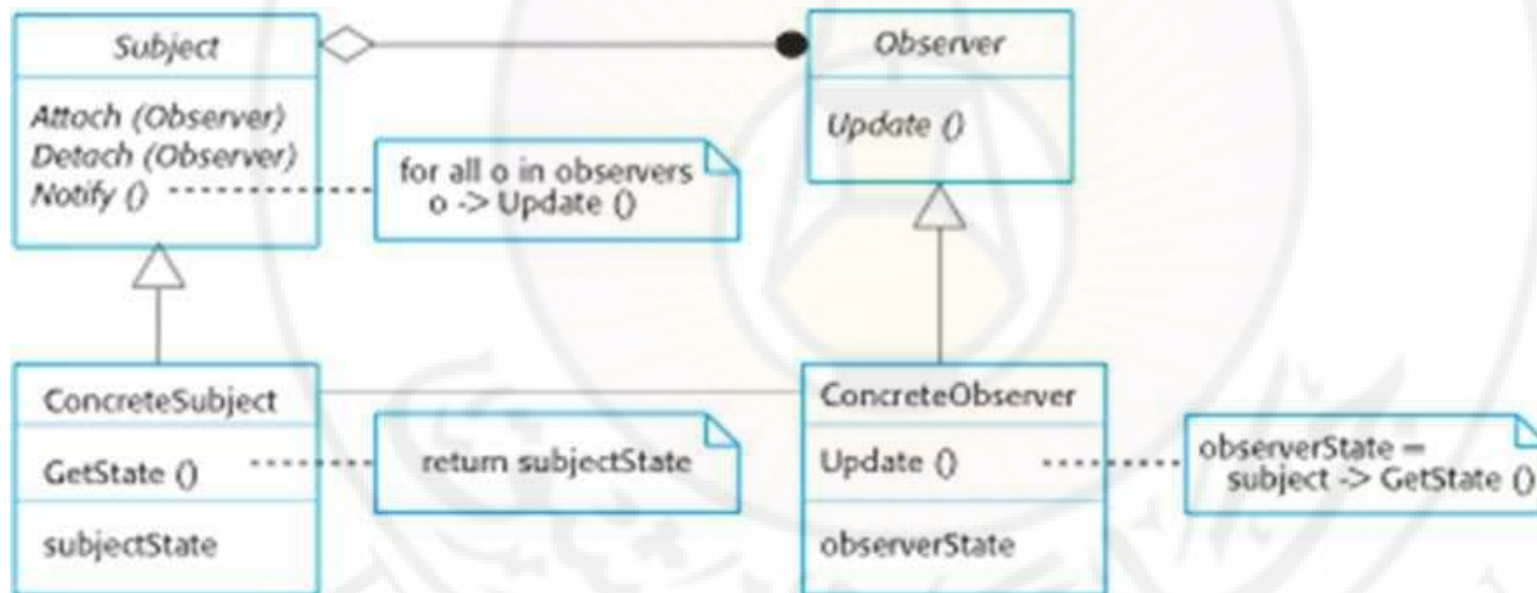
The Observer pattern (2)

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

Multiple displays using the Observer pattern



A UML model of the Observer pattern



Implementation issues

✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:

- **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
- **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
- **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse

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

Reuse levels

✧ The abstraction level

- At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.

✧ The object level

- At this level, you directly reuse objects from a library rather than writing the code yourself.

✧ The component level

- Components are collections of objects and object classes that you reuse in application systems.

✧ The system level

- At this level, you reuse entire application systems.

Reuse costs

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

Host-target development

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

Integrated development environments (IDEs)

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

Component/system deployment factors

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

Open source development

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

Open source systems

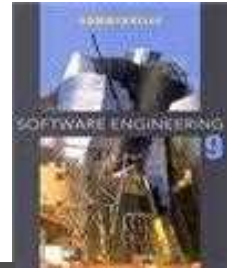
- ✧ The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- ✧ Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues

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

Open source business

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



Software Processes

Lecture 2

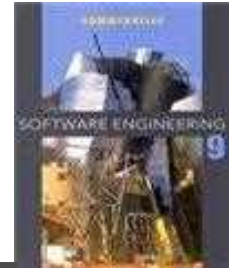
Dr. Mohammad Ahmad

Ph.D. in Software Engineering

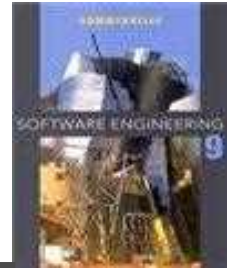
MCA Master of Computer Application

MBA-IT Master of Business Administration in Information Technology

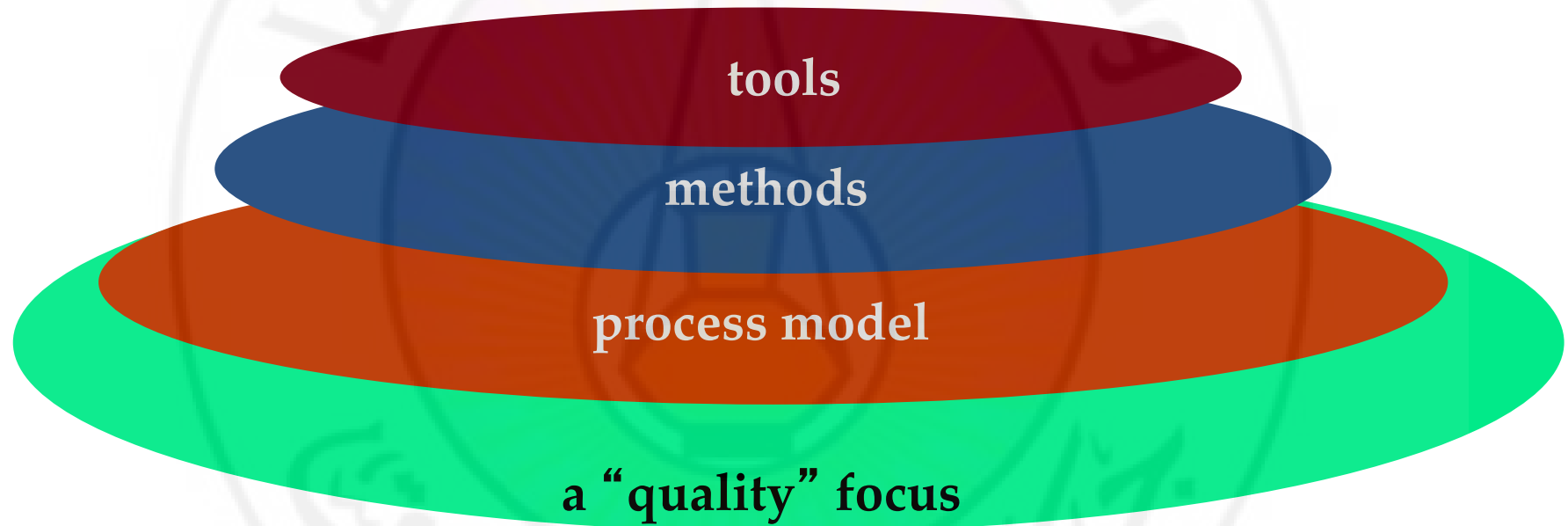
Topics covered

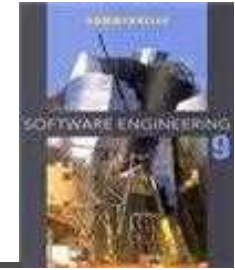


- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ The Rational Unified Process
 - An example of a modern software process.



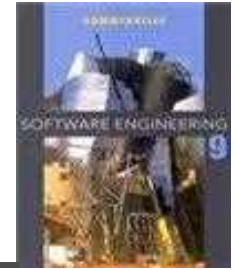
A Layered Technology





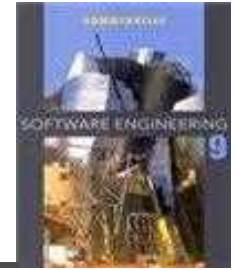
The software process

- ✧ A structured set of activities required to develop a software system.
- ✧ Purpose of process is to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.
- ✧ Many different software processes but all involve:
 - Specification – defining what the system should do;
 - Design and implementation – defining the organization of the system and implementing the system;
 - Validation – checking that it does what the customer wants;
 - Evolution – changing the system in response to changing customer needs.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.



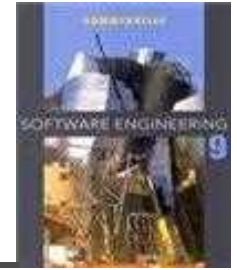
Software process descriptions

- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
 - Products, which are the outcomes of a process activity;
 - Roles, which reflect the responsibilities of the people involved in the process;
 - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.



Plan-driven and agile processes

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.



Software process models

✧ The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

✧ Incremental development

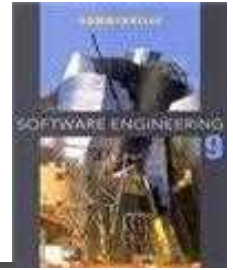
- Specification, development and validation are interleaved. May be plan-driven or agile.

✧ Reuse-oriented software engineering

- The system is assembled from existing components. May be plan-driven or agile.

✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

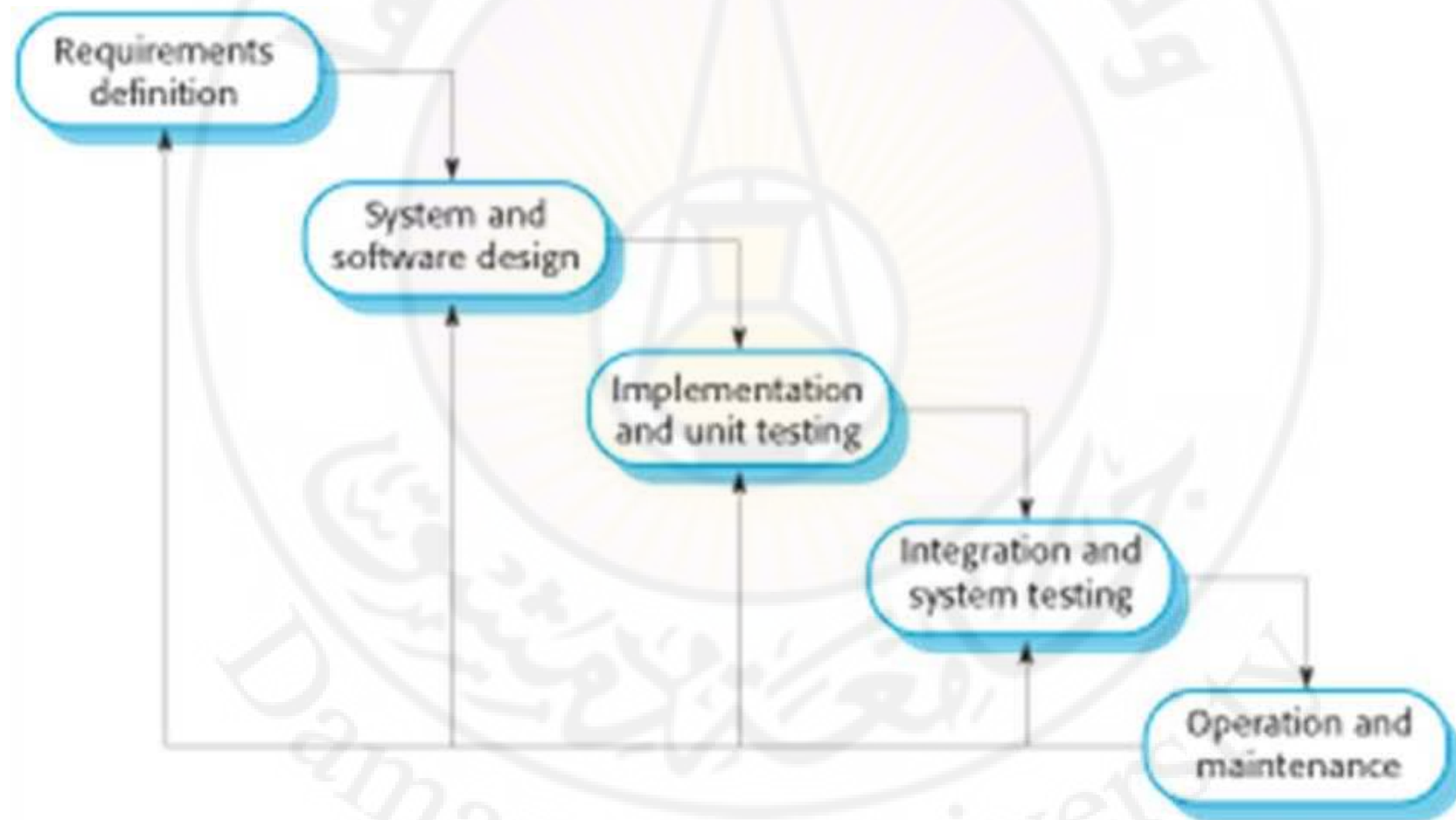
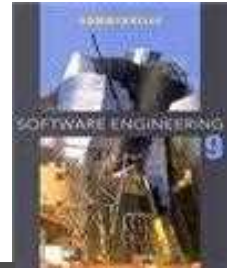
Adapting a Process Model



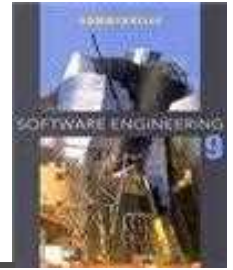
Process adopted for one project might be significantly different than a process adopted from another project. (to the problem, the project, the team, organizational culture). Among the differences are:

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

The waterfall model

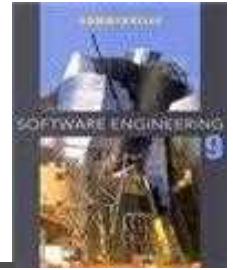


Waterfall model phases



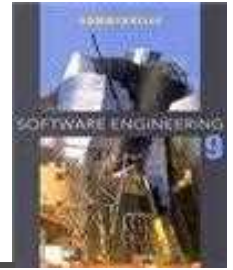
- ✧ There are separate identified phases in the waterfall model:
 - Requirements analysis and definition
 - System and software design
 - Implementation and unit testing
 - Integration and system testing
 - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

Waterfall model problems



- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
 - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
 - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
 - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Waterfall Model pros and cons

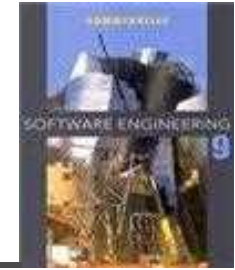


✧ Pros:

- This model is easy to understand and reinforces the notion of “define before design” and “design before code”

✧ Cons: Problems of waterfall model

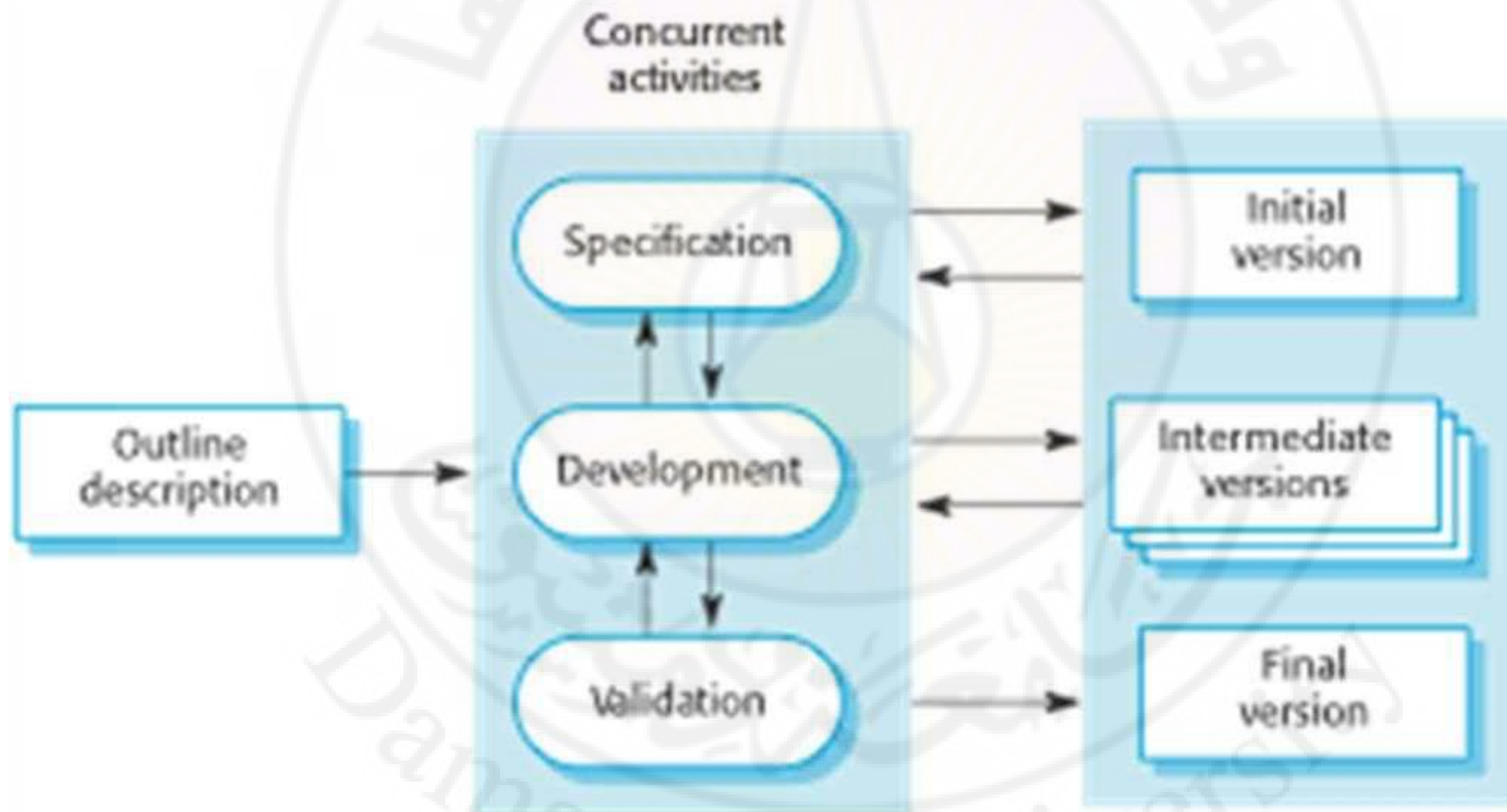
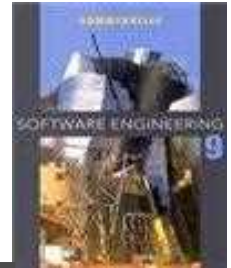
- It is difficult to define all requirements at the beginning of a project
- This model is not suitable for accommodating any change
- A working version of the system is not seen until late in the project's life
- It does not scale up well to large projects.
- Real projects are rarely sequential.

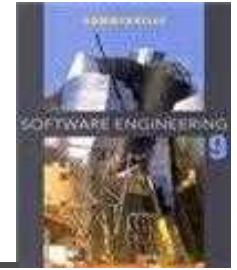


Incremental Process Models

- ✧ They are effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product.
- ✧ After every cycle a useable product is given to the customer.
- ✧ Popular particularly when we have to quickly deliver a limited functionality system.

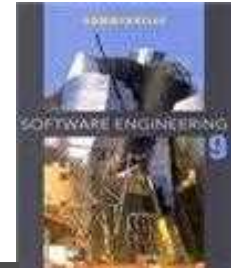
Incremental development





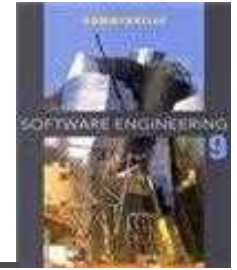
Incremental development benefits

- ✧ The cost of accommodating changing customer requirements is reduced.
 - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
 - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
 - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.



Incremental development problems

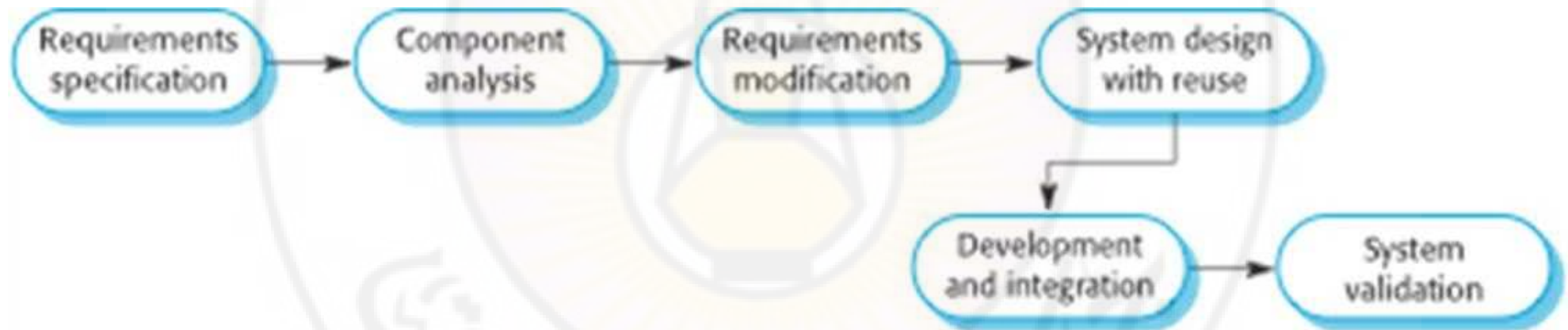
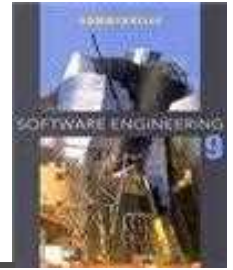
- ✧ The process is not visible.
 - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
 - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.



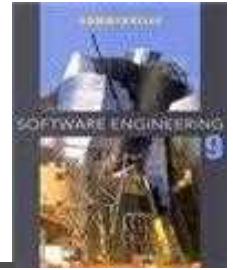
Reuse-oriented software engineering

- ✧ Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- ✧ Process stages
 - Component analysis;
 - Requirements modification;
 - System design with reuse;
 - Development and integration.
- ✧ Reuse is now the standard approach for building many types of business system

Reuse-oriented software engineering

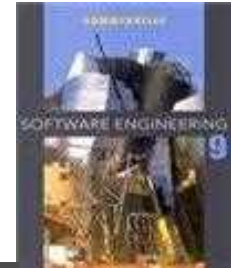


Types of software component



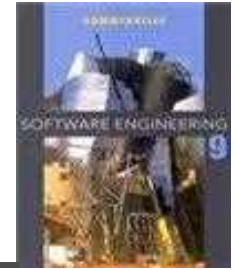
- ✧ Web services that are developed according to service standards and which are available for remote invocation.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Stand-alone software systems (COTS) that are configured for use in a particular environment.





Process activities

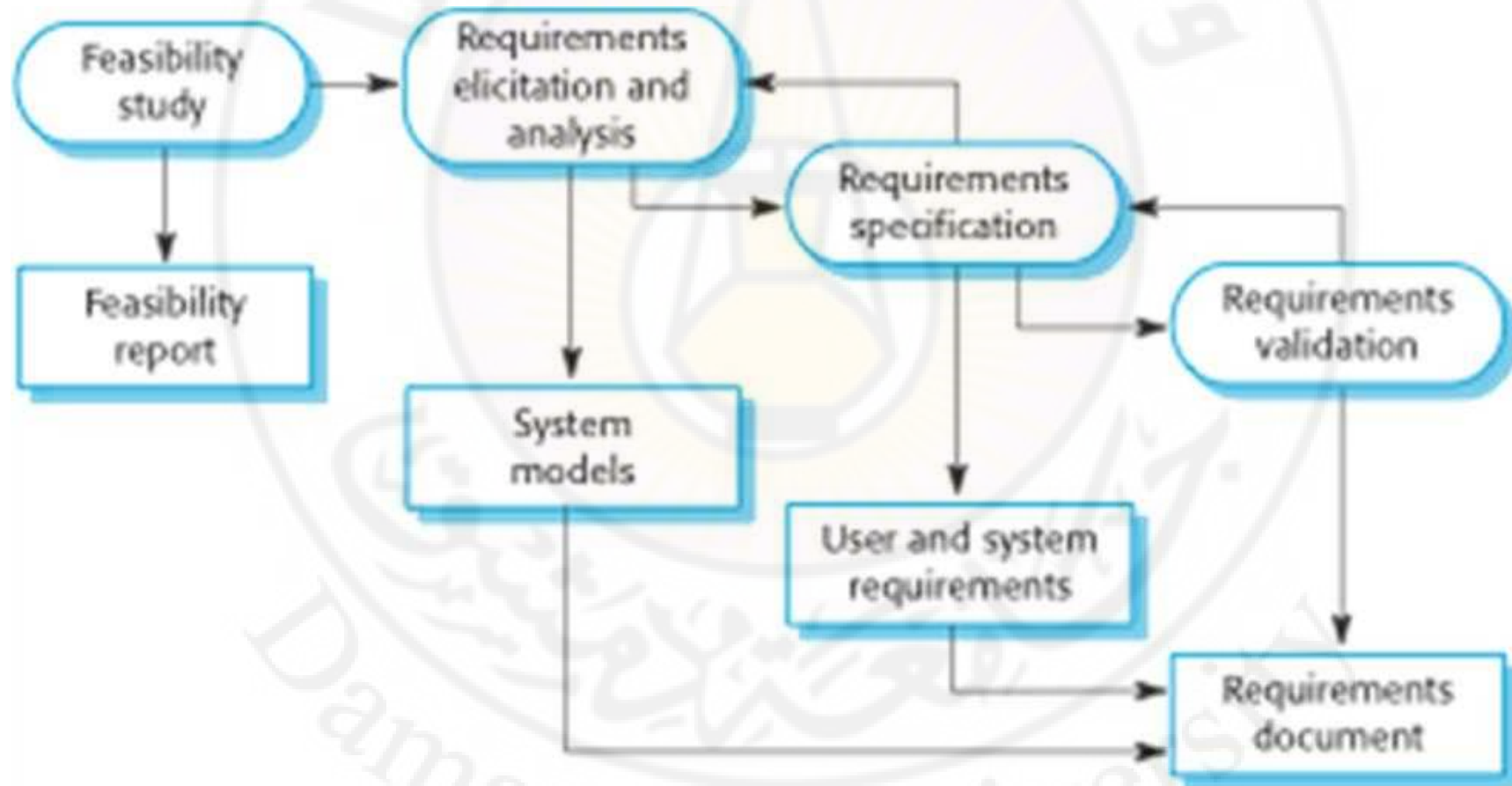
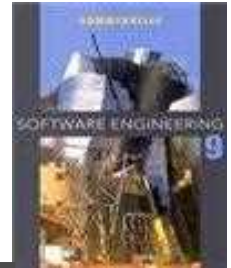
- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.

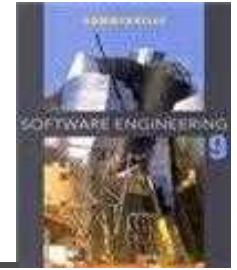


Software specification

- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
 - Feasibility study
 - Is it technically and financially feasible to build the system?
 - Requirements elicitation and analysis
 - What do the system stakeholders require or expect from the system?
 - Requirements specification
 - Defining the requirements in detail
 - Requirements validation
 - Checking the validity of the requirements

The requirements engineering process

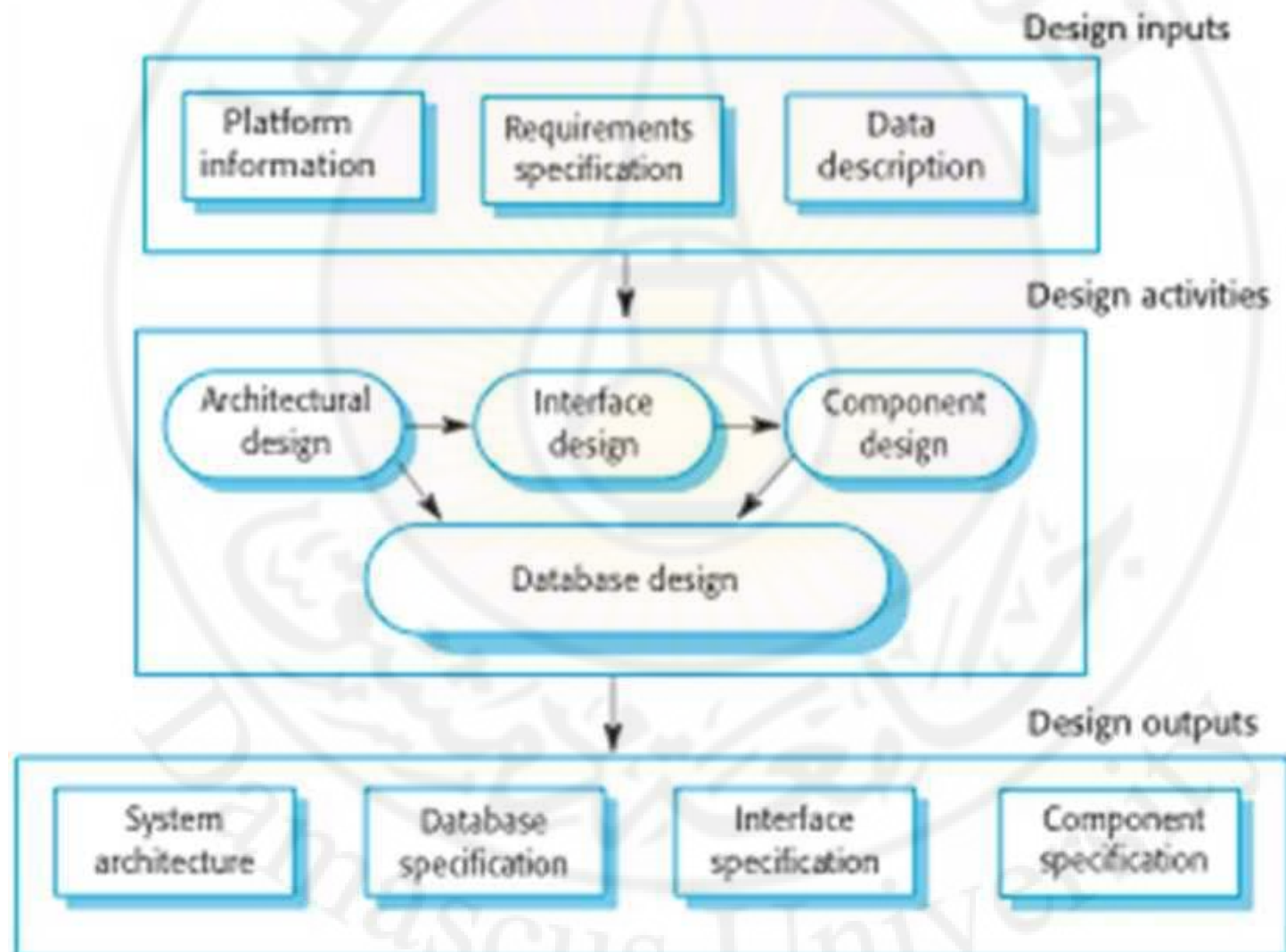
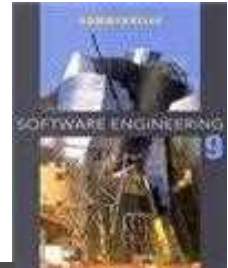


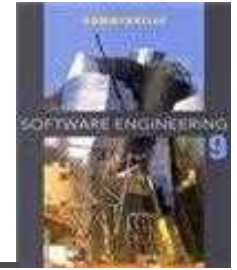


Software design and implementation

- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
 - Design a software structure that realises the specification;
- ✧ Implementation
 - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

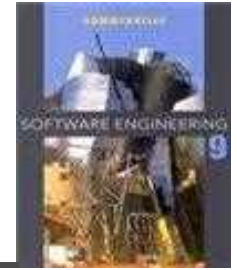
A general model of the design process





Design activities

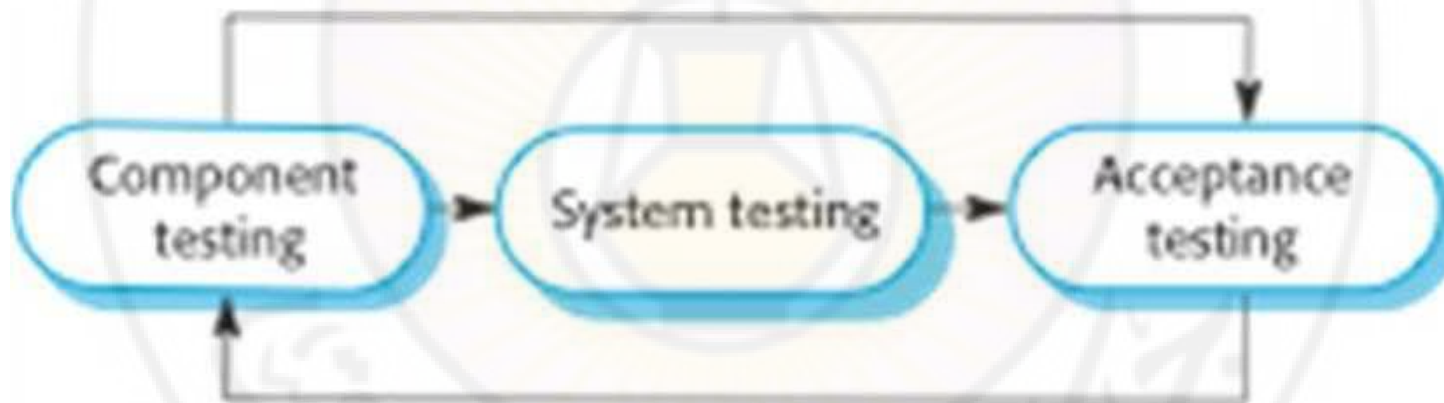
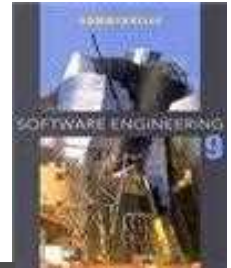
- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component design*, where you take each system component and design how it will operate.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.



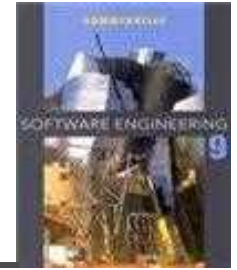
Software validation

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

Stages of testing



Testing stages



✧ Development or component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

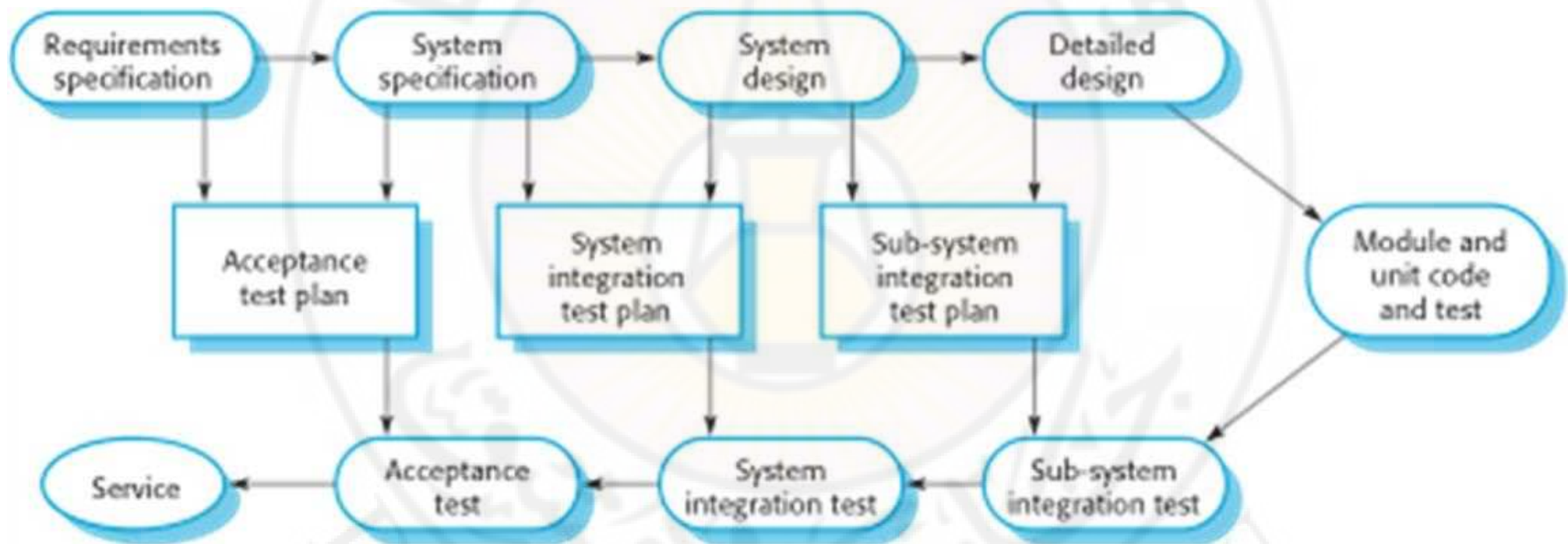
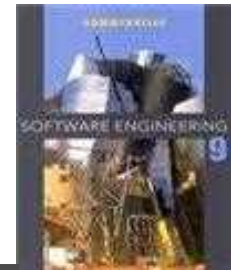
✧ System testing

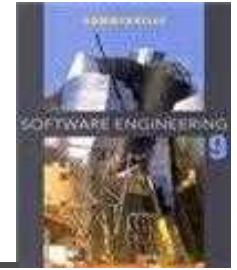
- Testing of the system as a whole. Testing of emergent properties is particularly important.

✧ Acceptance testing

- Testing with customer data to check that the system meets the customer's needs.

Testing phases in a plan-driven software process

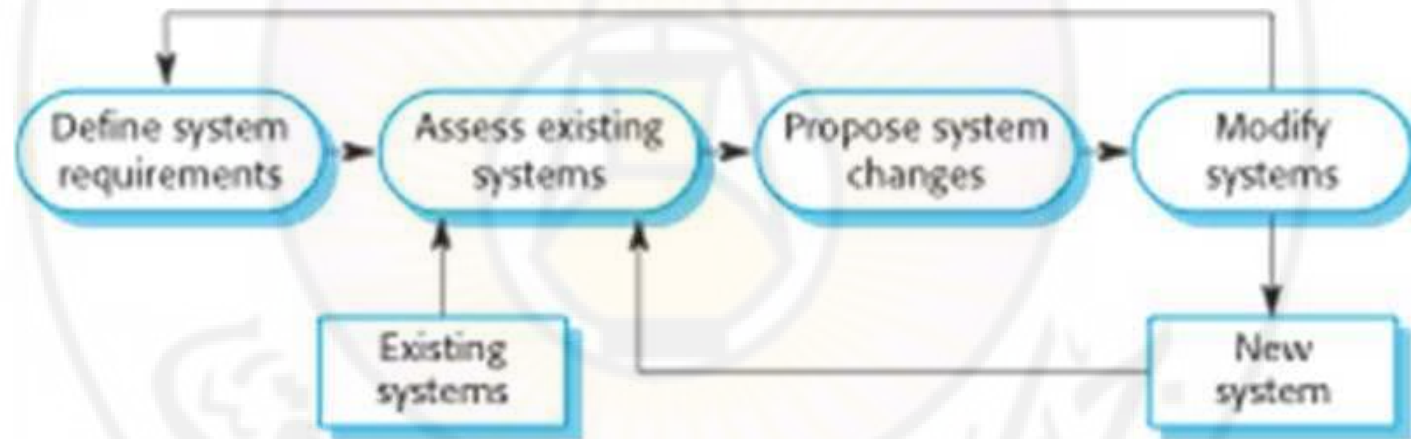
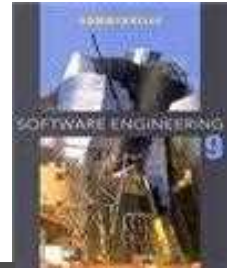


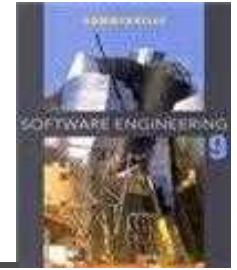


Software evolution

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

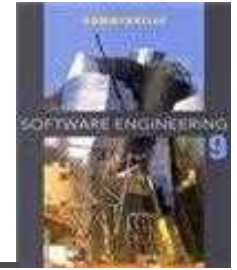
System evolution





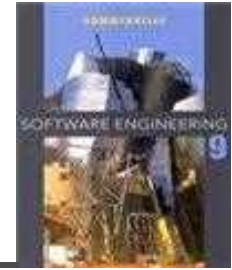
Coping with change

- ✧ Change is inevitable in all large software projects.
 - Business changes lead to new and changed system requirements
 - New technologies open up new possibilities for improving implementations
 - Changing platforms require application changes
- ✧ Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality



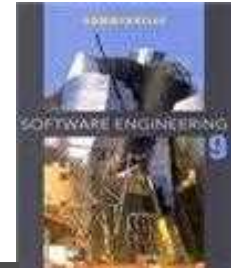
Reducing the costs of rework

- ✧ Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required.
 - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
 - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.



Software prototyping

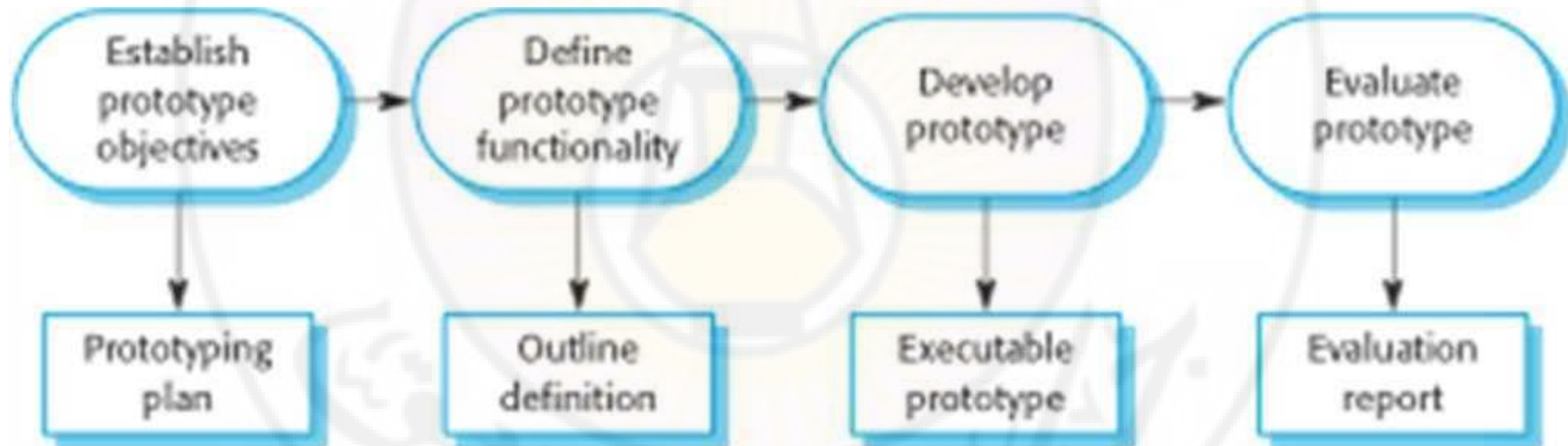
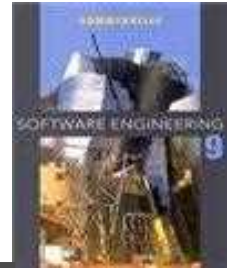
- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
 - The requirements engineering process to help with requirements elicitation and validation;
 - In design processes to explore options and develop a UI design;
 - In the testing process to run back-to-back tests.
 - This model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

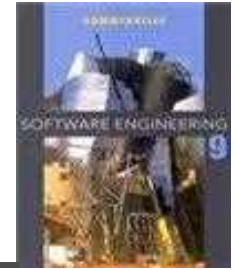


Benefits of prototyping

- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

The process of prototype development

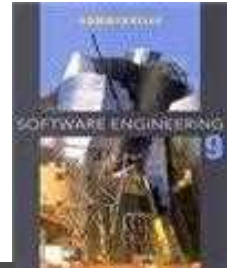




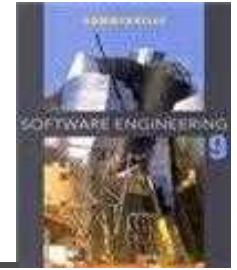
Prototype development

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
 - Prototype should focus on areas of the product that are not well-understood;
 - Error checking and recovery may not be included in the prototype;
 - Focus on functional rather than non-functional requirements such as reliability and security

Throw-away prototypes

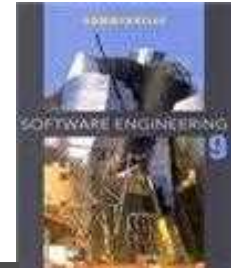


- ✧ Prototypes should be discarded after development as they are not a good basis for a production system:
 - It may be impossible to tune the system to meet non-functional requirements;
 - Prototypes are normally undocumented;
 - The prototype structure is usually degraded through rapid change;
 - The prototype probably will not meet normal organisational quality standards.



Incremental delivery

- ✧ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- ✧ User requirements are prioritised and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



Incremental development and delivery

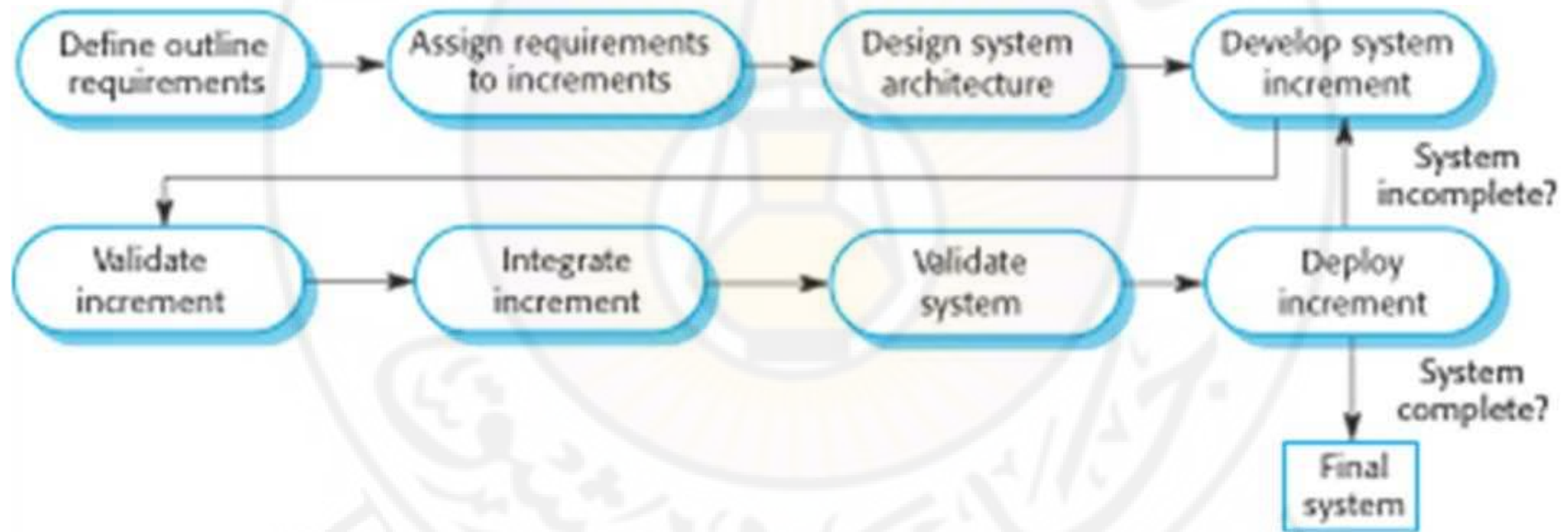
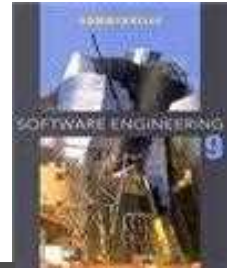
✧ Incremental development

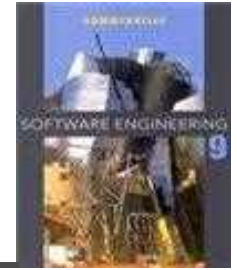
- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.

✧ Incremental delivery

- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

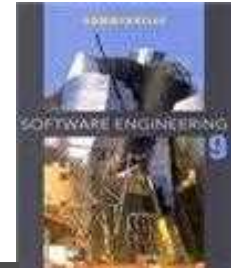
Incremental delivery





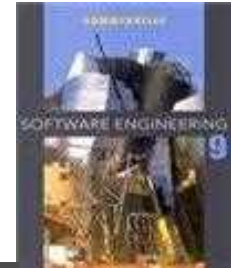
Incremental delivery advantages

- ✧ Customer value can be delivered with each increment so system functionality is available earlier.
- ✧ Early increments act as a prototype to help elicit requirements for later increments.
- ✧ Lower risk of overall project failure.
- ✧ The highest priority system services tend to receive the most testing.



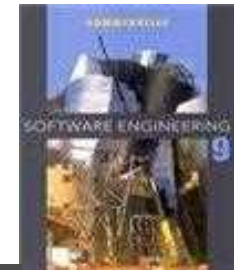
Incremental delivery problems

- ✧ Most systems require a set of basic facilities that are used by different parts of the system.
 - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the specification is developed in conjunction with the software.
 - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

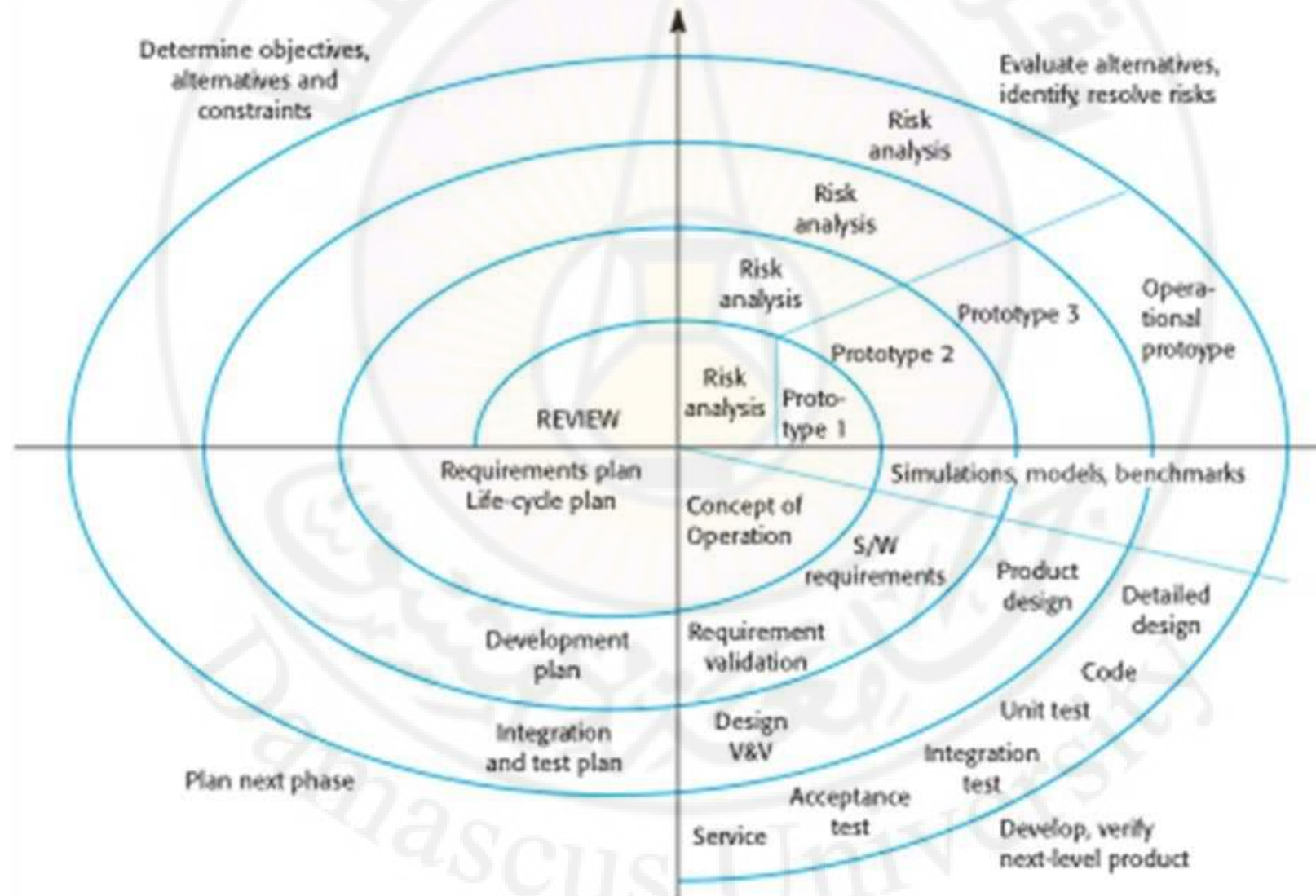


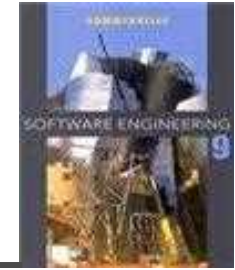
Boehm's spiral model

- ✧ Process is represented as a spiral rather than as a sequence of activities with backtracking.
- ✧ Each loop in the spiral represents a phase in the process.
- ✧ No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- ✧ Risks are explicitly assessed and resolved throughout the process.



Boehm's spiral model of the software process

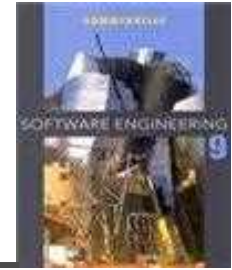




Spiral Model

The angular dimension represents the progress made in completing each cycle. Each loop of the spiral from X-axis clockwise through 360°, represents one phase. One phase is split roughly into four sectors of major activities.

- ✧ Planning: Determination of objectives, alternatives & constraints.
- ✧ Risk Analysis: Analyze alternatives and attempts to identify and resolve the risks involved.
- ✧ Development: Product development and testing product
- ✧ Assessment: Customer evaluation



Spiral model sectors

✧ Objective setting

- Specific objectives for the phase are identified.

✧ Risk assessment and reduction

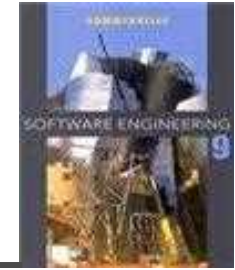
- Risks are assessed and activities put in place to reduce the key risks.

✧ Development and validation

- A development model for the system is chosen which can be any of the generic models.

✧ Planning

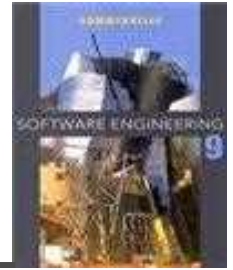
- The project is reviewed and the next phase of the spiral is planned.



Spiral model usage

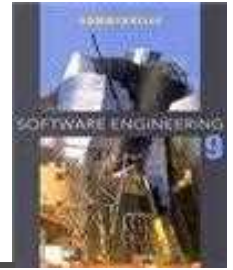
- ✧ Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- ✧ In practice, however, the model is rarely used as published for practical software development.
- ✧ An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers)

The Unified Process

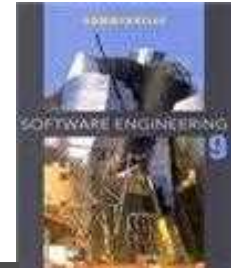


- ✧ Software engineering process with the goal of producing good quality maintainable software within specified time and budget.
- ✧ Developed through a series of fixed length mini projects called iterations.
- ✧ Maintained and enhanced by Rational Software Corporation and thus referred to as Rational Unified Process (RUP).

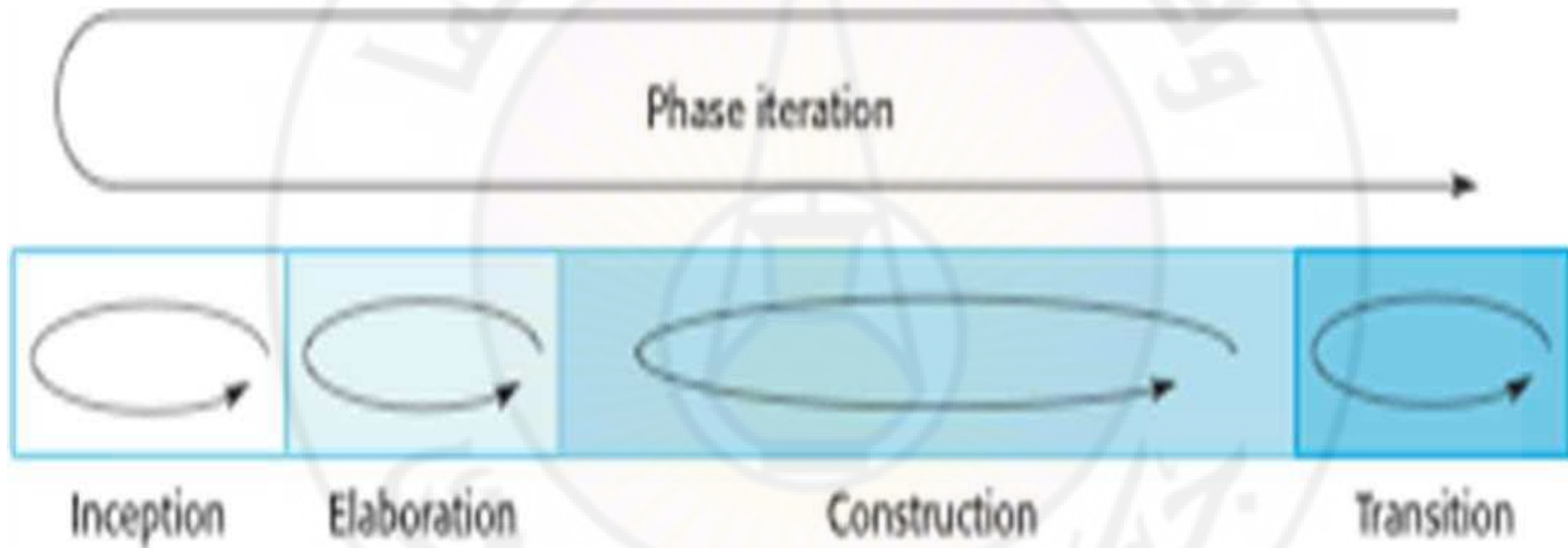
The Rational Unified Process

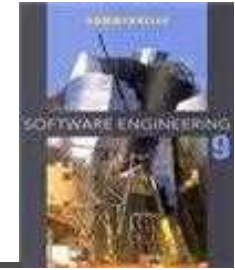


- ✧ A modern generic process derived from the work on the UML and associated process.
- ✧ Brings together aspects of the 3 generic process models discussed previously.
- ✧ Normally described from 3 perspectives
 - A dynamic perspective that shows phases over time;
 - A static perspective that shows process activities;
 - A practice perspective that suggests good practice.



Phases in the Rational Unified Process





RUP phases

✧ Inception

- Establish the business case for the system, defines scope of the project.
- Identify all external entities (people and system) that will interact with system and define these interactions.
- Use the previous information to assess the contribution that the system makes to the business. If this contribution is minor then the project may be cancelled

✧ Elaboration

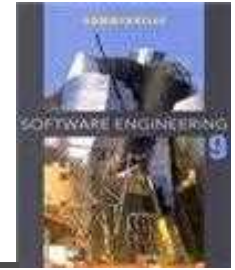
- Develop an understanding of the problem domain and the system architecture, How do we plan & design the project?
- What resources are required?
- What type of architecture may be suitable?

✧ Construction

- System design, programming and testing.

✧ Transition

- Deploy the system in its operating environment, it involves many activities like delivering, training, supporting, and maintaining the product.



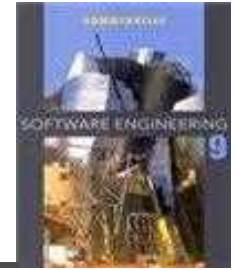
RUP iteration

✧ In-phase iteration

- Each phase is iterative with results developed incrementally.

✧ Cross-phase iteration

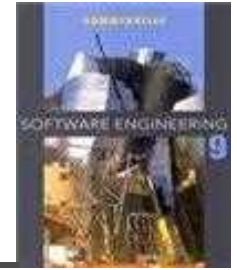
- As shown by the loop in the RUP model, the whole set of phases may be enacted incrementally.



Inception Phase

The inception phase has the following objectives:

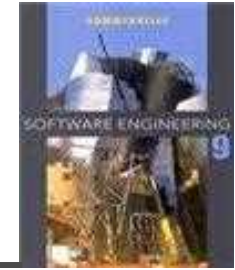
- ✧ Gathering and analyzing the requirements.
- ✧ Planning and preparing a business case and evaluating alternatives for risk management, scheduling resources etc.
- ✧ Estimating the overall cost and schedule for the project.
- ✧ Studying the feasibility and calculating profitability of the project.



Elaboration Phase

The elaboration phase has the following objectives:

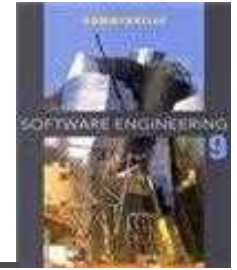
- ✧ Establishing architectural foundations.
- ✧ Design of use case model.
- ✧ Elaborating the process, infrastructure & development environment.
- ✧ Selecting component.
- ✧ Demonstrating that architecture support the vision at reasonable cost & within specified time.



Construction Phase

The construction phase has the following objectives:

- ✧ Implementing the project.
- ✧ Minimizing development cost.
- ✧ Management and optimizing resources.
- ✧ Testing the product.
- ✧ Assessing the product releases against acceptance criteria.

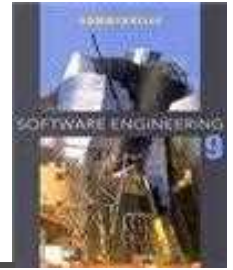


Transition Phase

The transition phase has the following objectives:

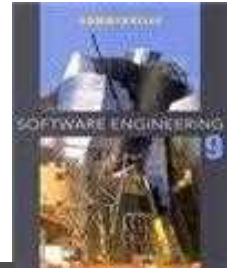
- ✧ Starting of beta testing
- ✧ Analysis of user's views.
- ✧ Training of users.
- ✧ Tuning activities including bug fixing and enhancements for performance & usability
- ✧ Assessing the customer satisfaction.

Static workflows in the Rational Unified Process

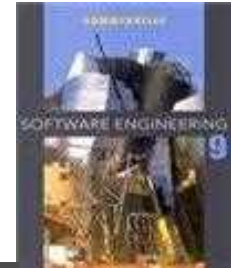


Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.

Static workflows in the Rational Unified Process



Workflow	Description
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system.
Project management	This supporting workflow manages the system development.
Environment	This workflow is concerned with making appropriate software tools available to the software development team.



RUP good practice

✧ Develop software iteratively

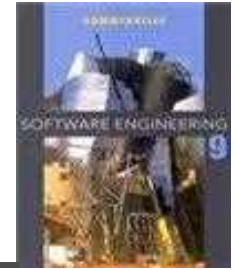
- Plan increments based on customer priorities and deliver highest priority increments first.

✧ Manage requirements

- Explicitly document customer requirements and keep track of changes to these requirements.

✧ Use component-based architectures

- Organize the system architecture as a set of reusable components.



RUP good practice

✧ Visually model software

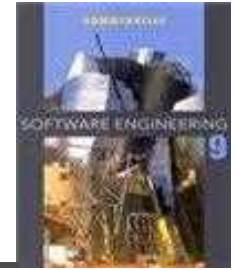
- Use graphical UML models to present static and dynamic views of the software.

✧ Verify software quality

- Ensure that the software meet's organizational quality standards.

✧ Control changes to software

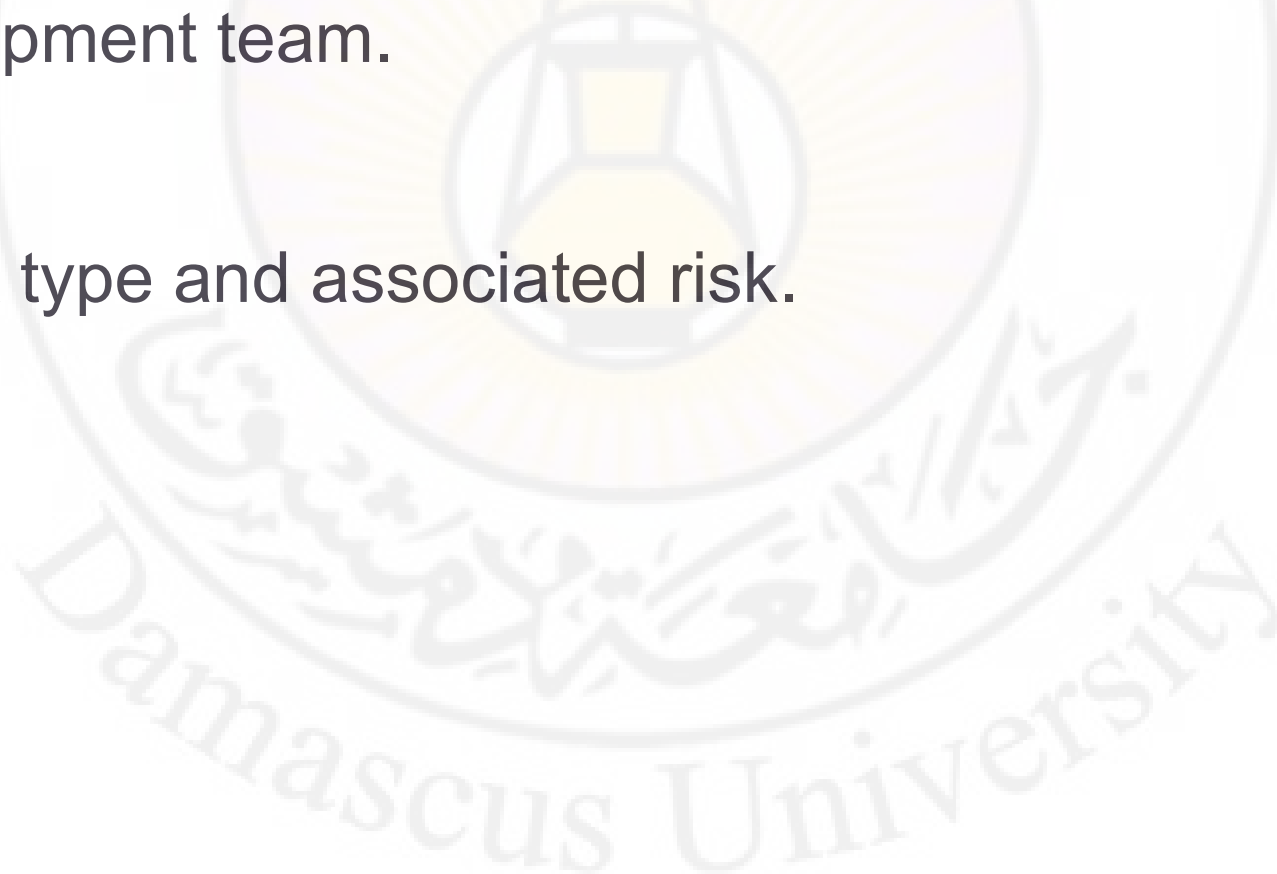
- Manage software changes using a change management system and configuration management tools.

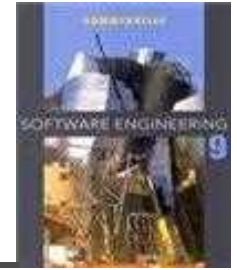


Selection of a Life Cycle Model

Selection of a model is based on:

- a) Requirements.
- b) Development team.
- c) Users.
- d) Project type and associated risk.

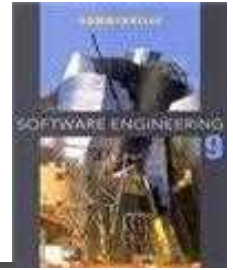




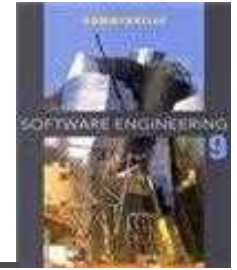
Key points

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes. Examples of these general models include the 'waterfall' model, incremental development, and reuse-oriented development.

Key points

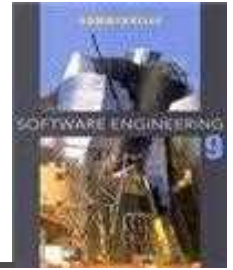


- ✧ Requirements engineering is the process of developing a software specification.
- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.



Key points

- ✧ Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design.
- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The Rational Unified Process is a modern generic process model that is organized into phases (inception, elaboration, construction and transition) but separates activities (requirements, analysis and design, etc.) from these phases.



Chapter 5 – System Modeling

Lecture 5

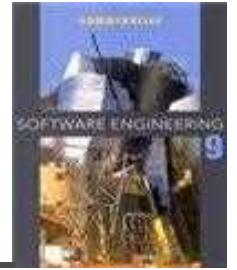
Dr. Mohammad Ahmad

Ph.D. in Software Engineering

MCA Master of Computer Application

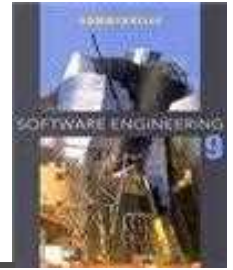
MBA-IT Master of Business Administration in Information Technology

Topics covered

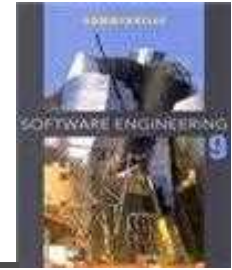


- ✧ Context models
- ✧ Interaction models
- ✧ Structural models
- ✧ Behavioral models
- ✧ Model-driven engineering

System modeling

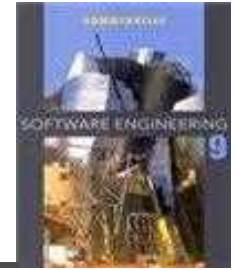


- ✧ System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- ✧ System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- ✧ System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.



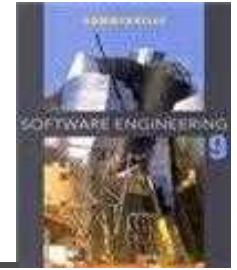
Existing and planned system models

- ✧ Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- ✧ Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- ✧ In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.



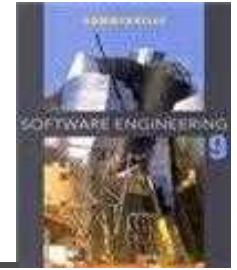
System perspectives

- ✧ An external perspective, where you model the context or environment of the system.
- ✧ An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- ✧ A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- ✧ A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.



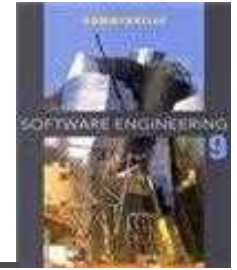
UML diagram types

- ✧ Activity diagrams, which show the activities involved in a process or in data processing .
- ✧ Use case diagrams, which show the interactions between a system and its environment.
- ✧ Sequence diagrams, which show interactions between actors and the system and between system components.
- ✧ Class diagrams, which show the object classes in the system and the associations between these classes.
- ✧ State diagrams, which show how the system reacts to internal and external events.



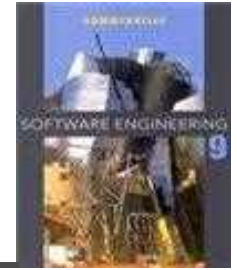
Use of graphical models

- ✧ As a means of facilitating discussion about an existing or proposed system
 - Incomplete and incorrect models are OK as their role is to support discussion.
- ✧ As a way of documenting an existing system
 - Models should be an accurate representation of the system but need not be complete.
- ✧ As a detailed system description that can be used to generate a system implementation
 - Models have to be both correct and complete.



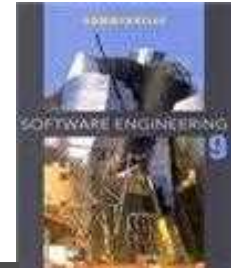
Context models

- ✧ Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- ✧ Social and organisational concerns may affect the decision on where to position system boundaries.
- ✧ Architectural models show the system and its relationship with other systems.



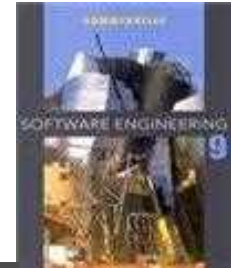
System boundaries

- ✧ System boundaries are established to define what is inside and what is outside the system.
 - They show other systems that are used or depend on the system being developed.
- ✧ The position of the system boundary has a profound effect on the system requirements.
- ✧ Defining a system boundary is a political judgment
 - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.



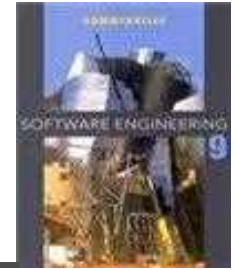
Process perspective

- ✧ Context models simply show the other systems in the environment, not how the system being developed is used in that environment.
- ✧ Process models reveal how the system being developed is used in broader business processes.
- ✧ UML activity diagrams may be used to define business process models.



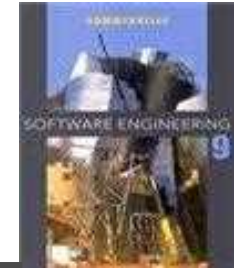
Interaction models

- ✧ Modeling user interaction is important as it helps to identify user requirements.
- ✧ Modeling system-to-system interaction highlights the communication problems that may arise.
- ✧ Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- ✧ Use case diagrams and sequence diagrams may be used for interaction modeling.



Use case modeling

- ✧ Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- ✧ Each use case represents a discrete task that involves external interaction with a system.
- ✧ Actors in a use case may be people or other systems.
- ✧ Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.

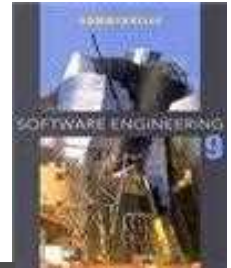


Transfer-data use case

✧ A use case in the MHC-PMS



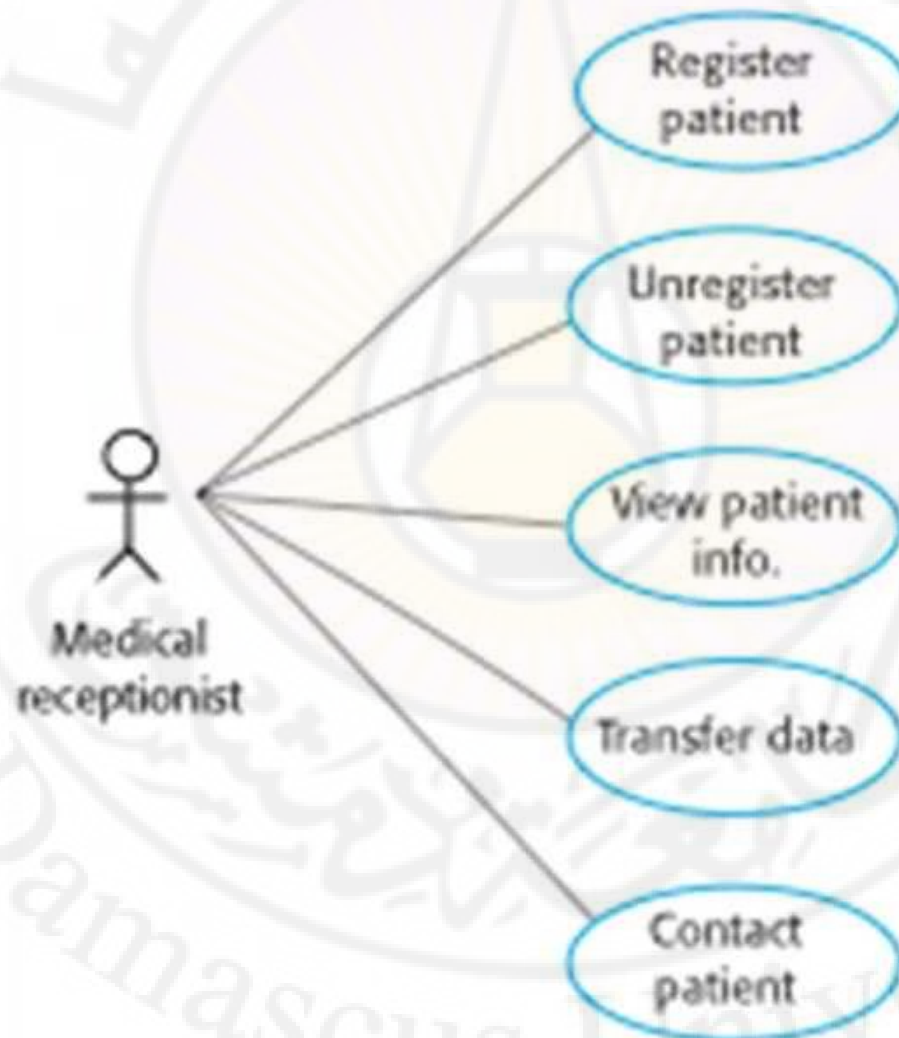
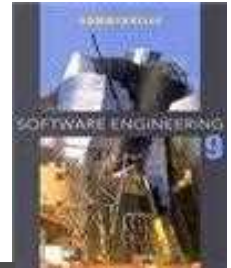
Tabular description of the 'Transfer data' use-case

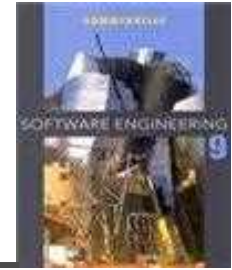


MHC-PMS: Transfer data

Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Use cases in the MHC-PMS involving the role 'Medical Receptionist'

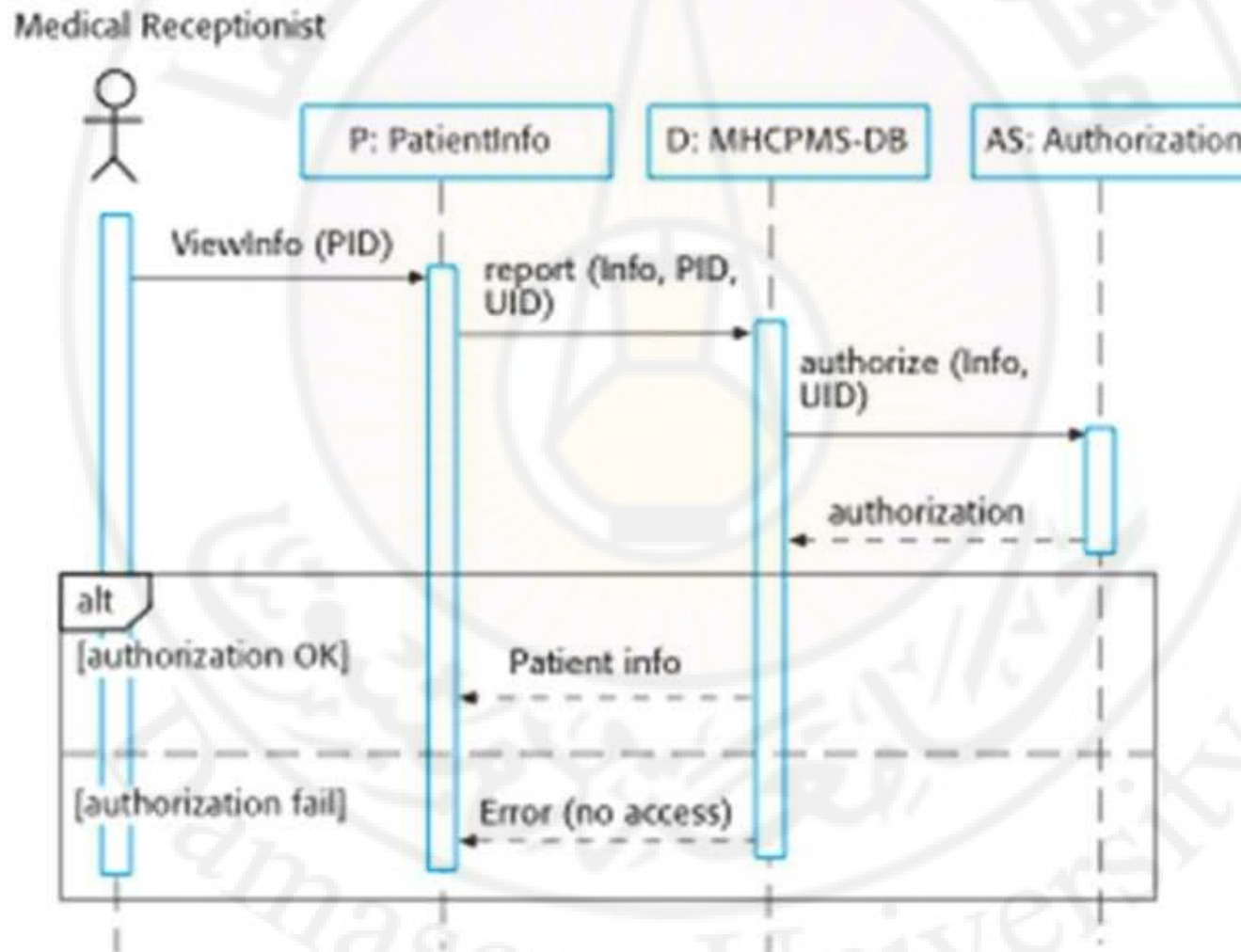


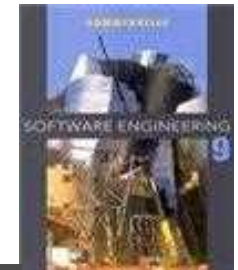


Sequence diagrams

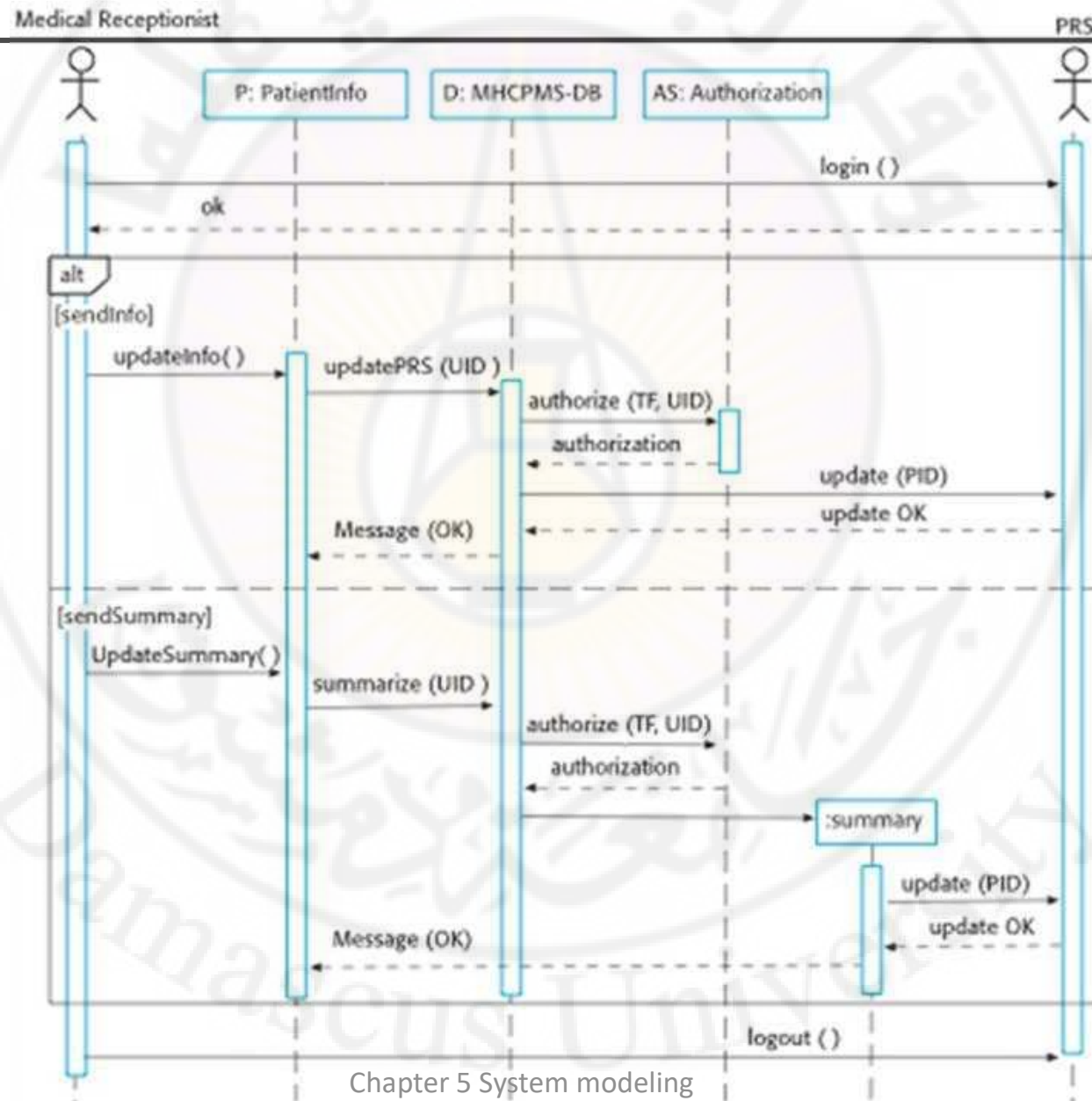
- ✧ Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
- ✧ A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
- ✧ The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- ✧ Interactions between objects are indicated by annotated arrows.

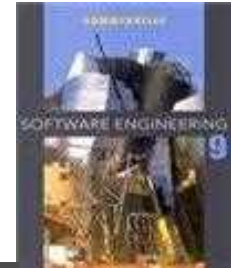
Sequence diagram for View patient information





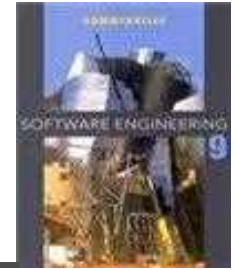
Sequence diagram for Transfer Data





Structural models

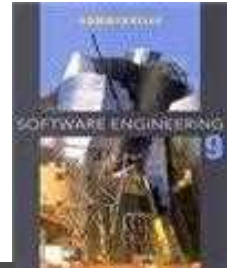
- ✧ Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- ✧ Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- ✧ You create structural models of a system when you are discussing and designing the system architecture.



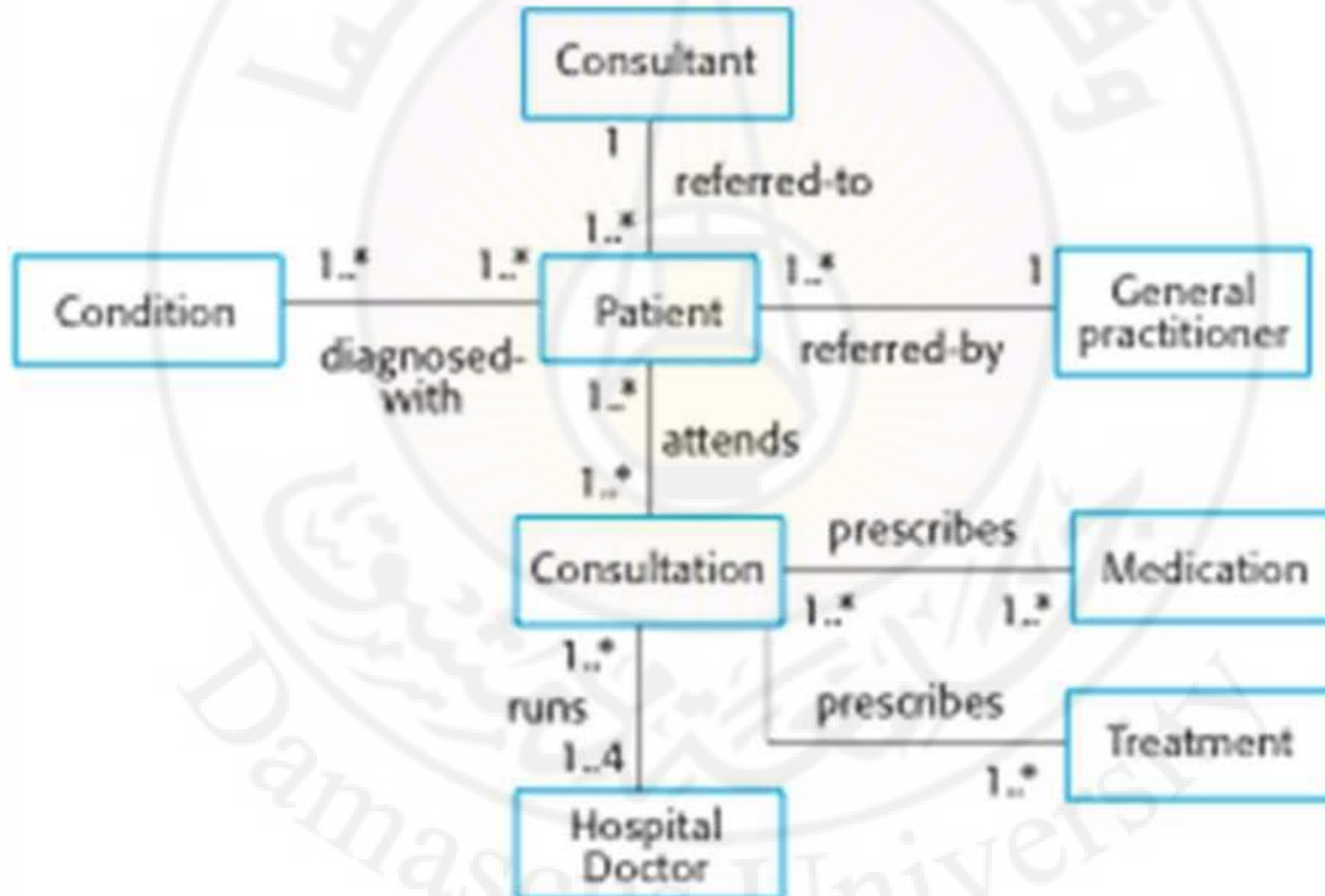
Class diagrams

- ✧ Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- ✧ An object class can be thought of as a general definition of one kind of system object.
- ✧ An association is a link between classes that indicates that there is some relationship between these classes.
- ✧ When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

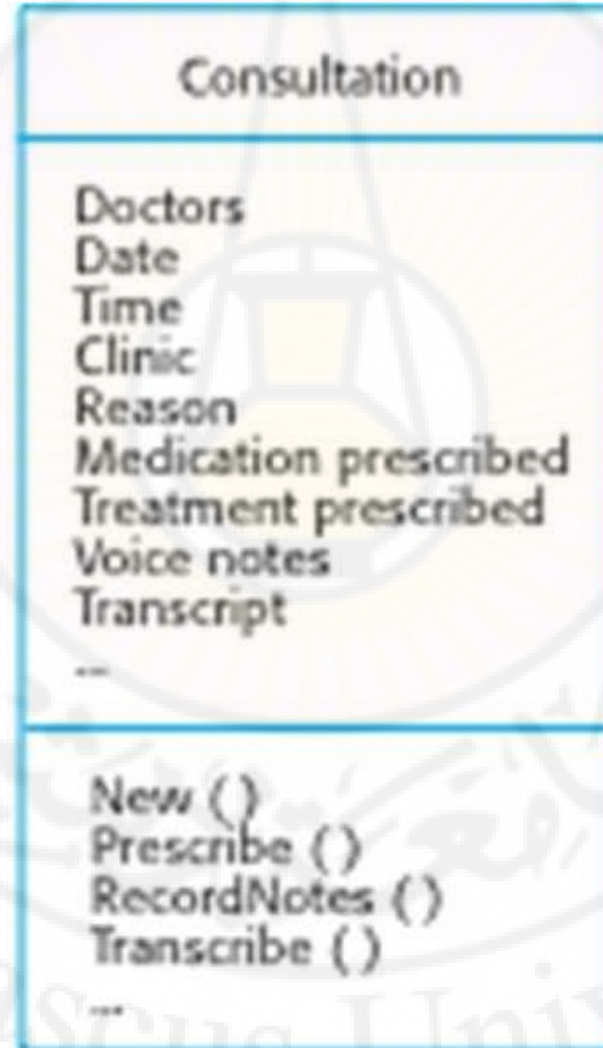
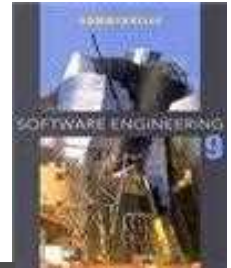
UML classes and association



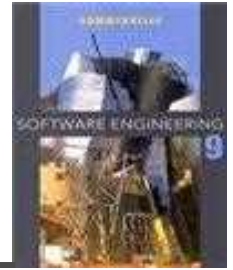
Classes and associations in the MHC-PMS



The Consultation class

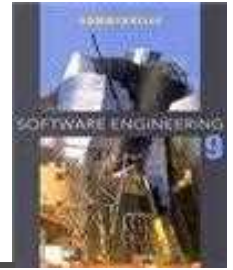


Generalization



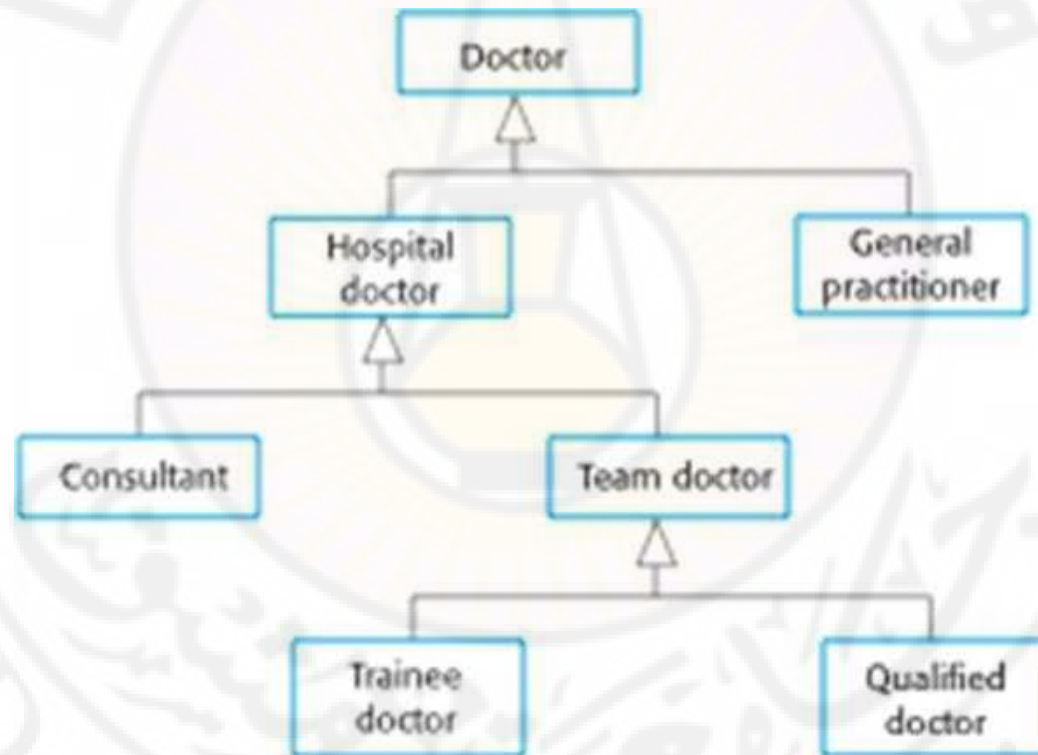
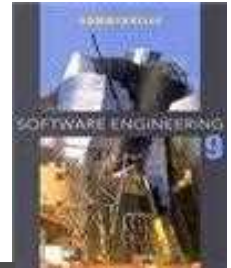
- ✧ Generalization is an everyday technique that we use to manage complexity.
- ✧ Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
- ✧ This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

Generalization

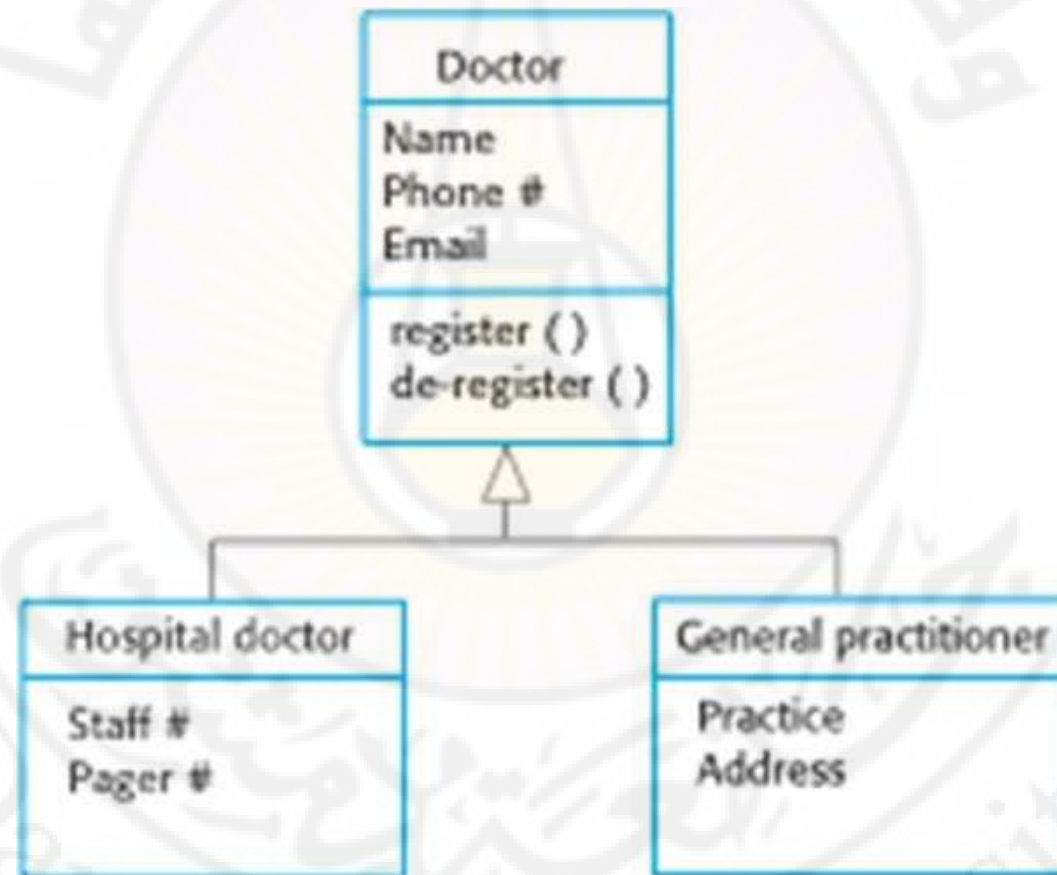
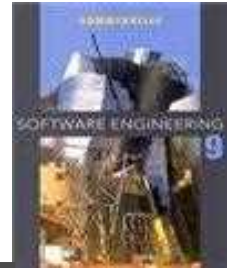


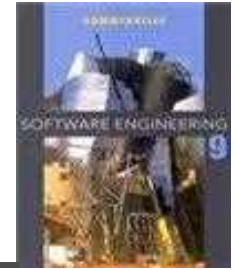
- ✧ In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
- ✧ In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
- ✧ In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
- ✧ The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A generalization hierarchy



A generalization hierarchy with added detail



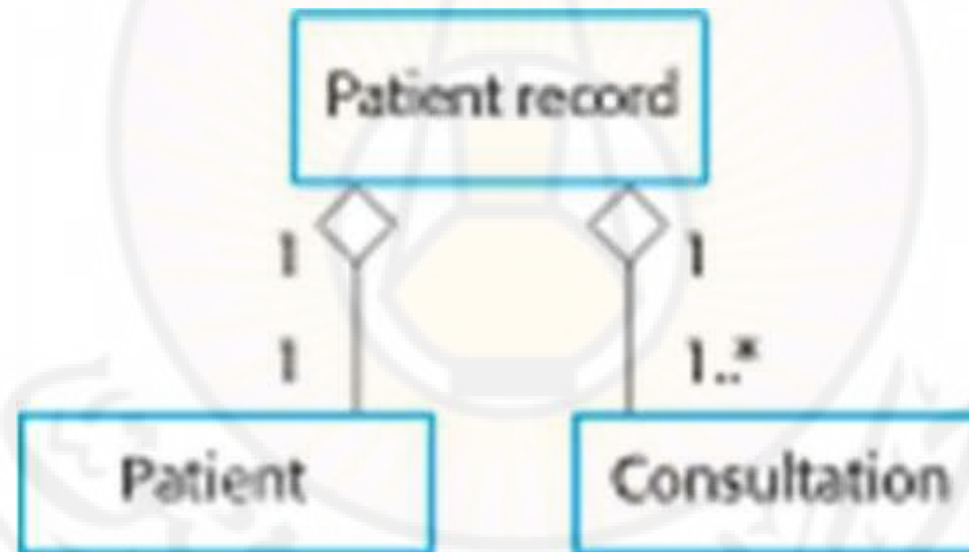
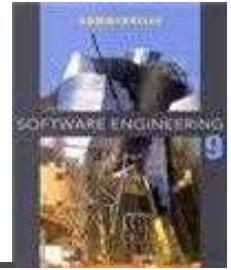


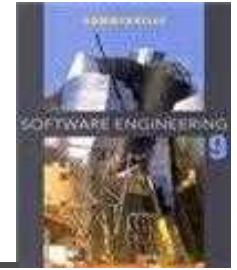
Object class aggregation models

- ✧ An aggregation model shows how classes that are collections are composed of other classes.
- ✧ Aggregation models are similar to the part-of relationship in semantic data models.



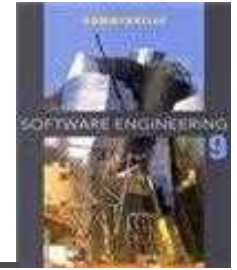
The aggregation association





Behavioral models

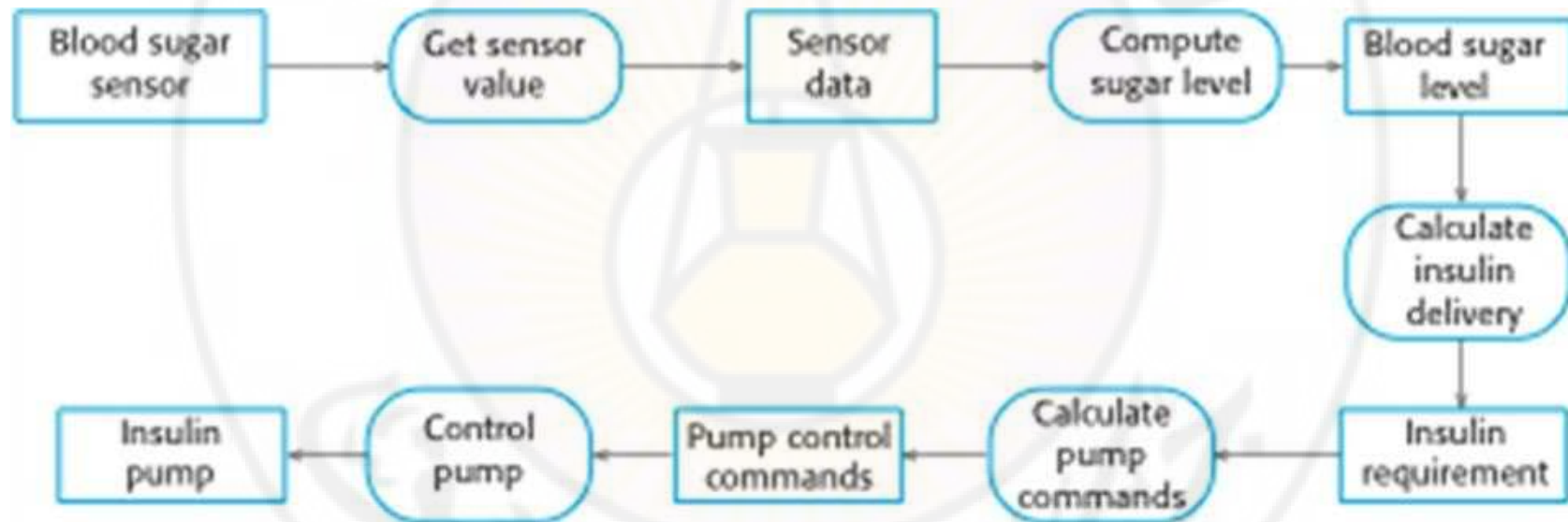
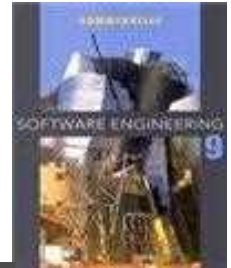
- ✧ Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
- ✧ You can think of these stimuli as being of two types:
 - **Data** Some data arrives that has to be processed by the system.
 - **Events** Some event happens that triggers system processing. Events may have associated data, although this is not always the case.



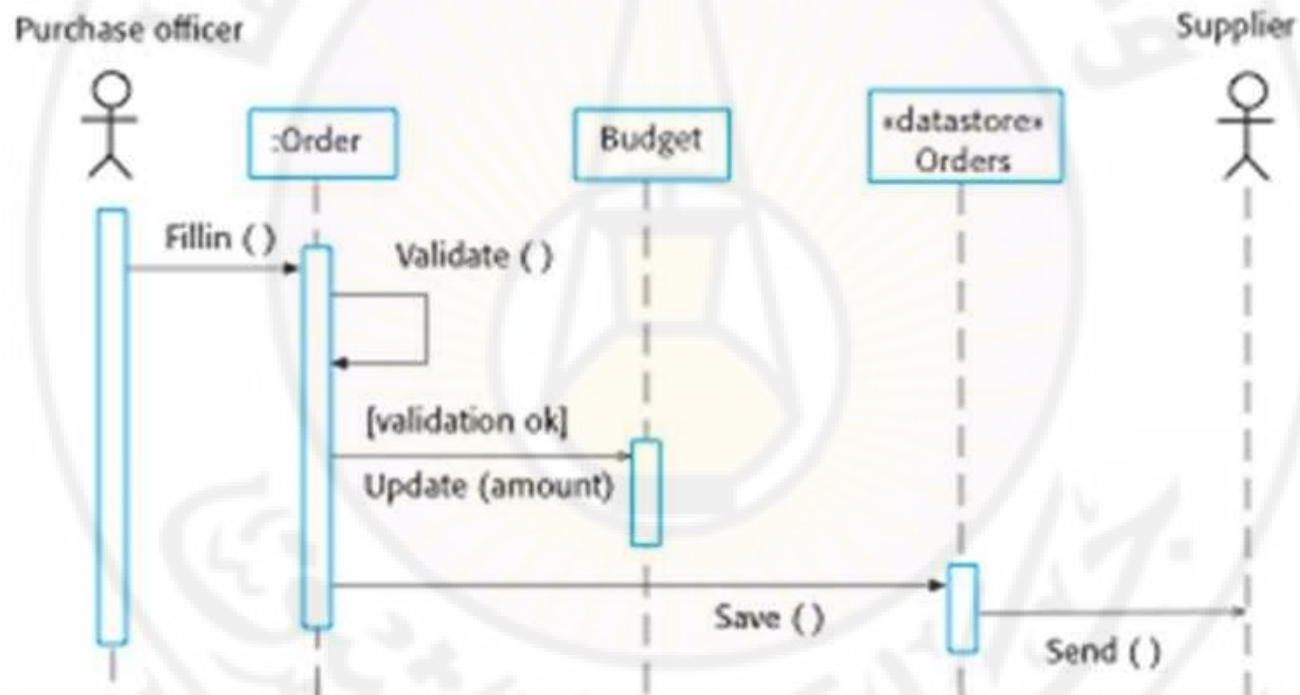
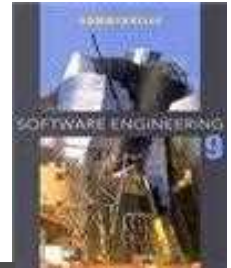
Data-driven modeling

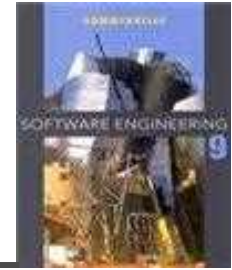
- ✧ Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- ✧ Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- ✧ They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

An activity model of the insulin pump's operation



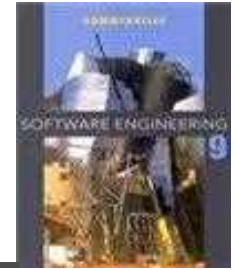
Order processing





Event-driven modeling

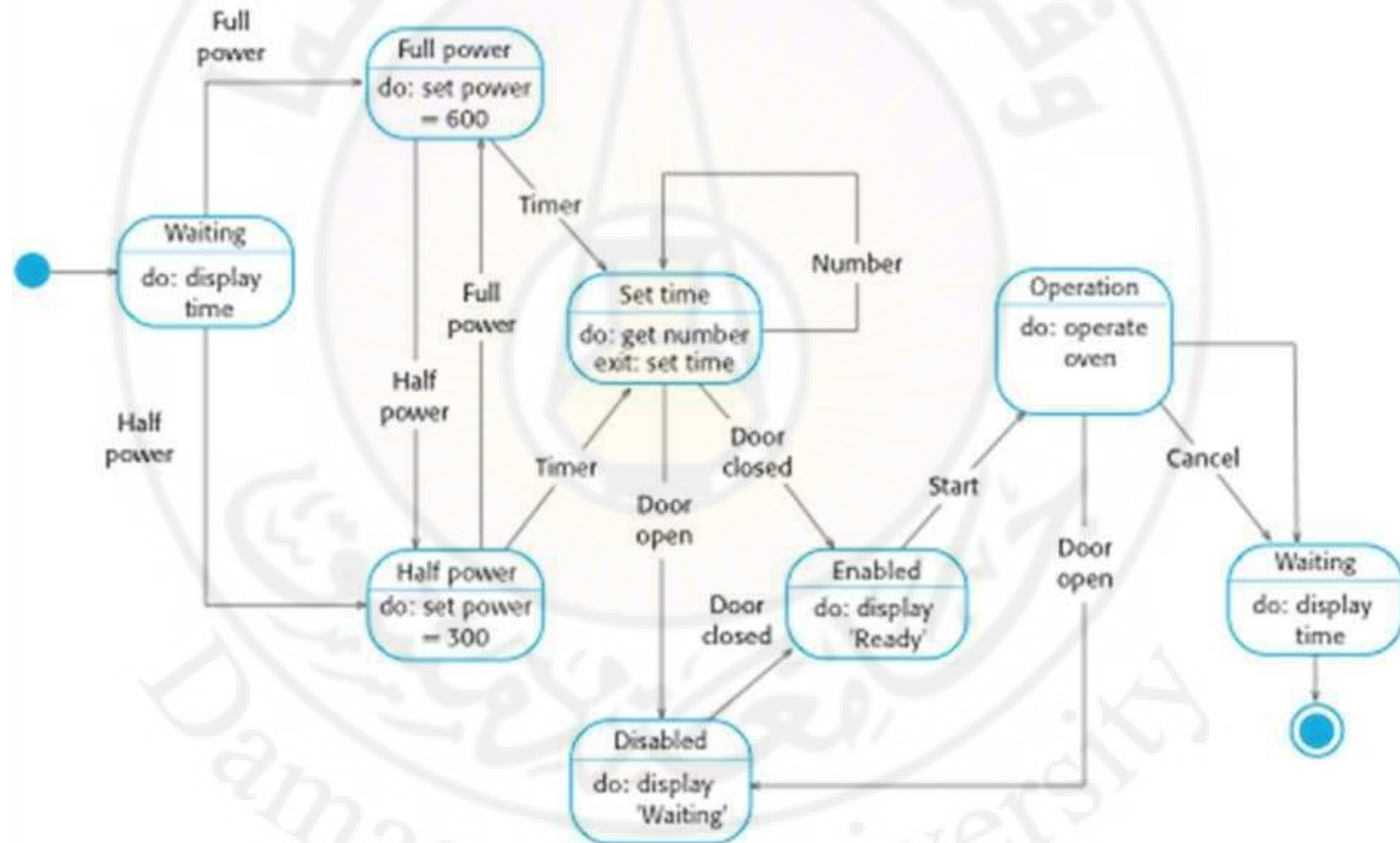
- ✧ Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone.
- ✧ Event-driven modeling shows how a system responds to external and internal events.
- ✧ It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

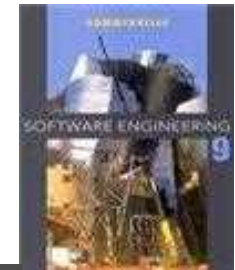


State machine models

- ✧ These model the behaviour of the system in response to external and internal events.
- ✧ They show the system's responses to stimuli so are often used for modelling real-time systems.
- ✧ State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- ✧ Statecharts are an integral part of the UML and are used to represent state machine models.

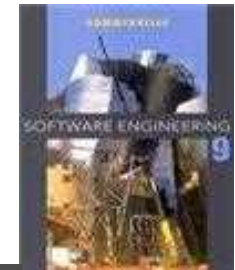
State diagram of a microwave oven





States and stimuli for the microwave oven (a)

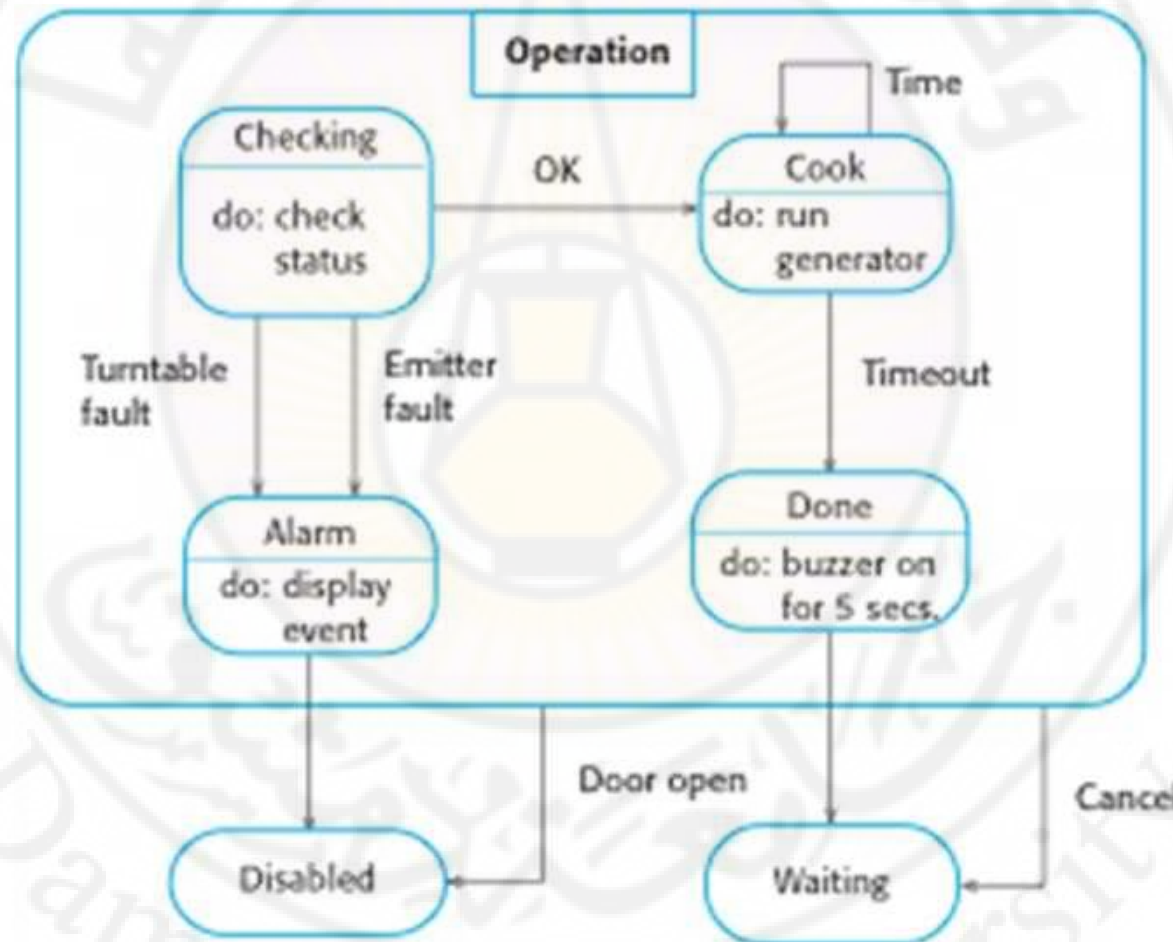
State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

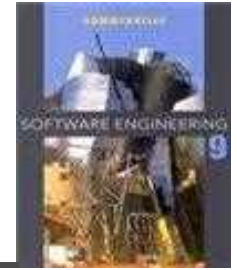


States and stimuli for the microwave oven (b)

Stimulus	Description
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

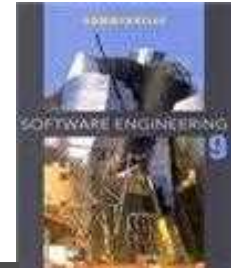
Microwave oven operation





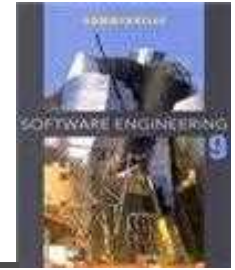
Model-driven engineering

- ✧ Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
- ✧ The programs that execute on a hardware/software platform are then generated automatically from the models.
- ✧ Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.



Usage of model-driven engineering

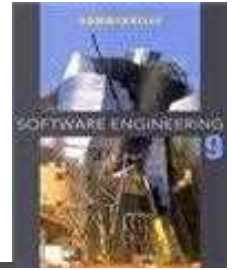
- ✧ Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
- ✧ Pros
 - Allows systems to be considered at higher levels of abstraction
 - Generating code automatically means that it is cheaper to adapt systems to new platforms.
- ✧ Cons
 - Models for abstraction and not necessarily right for implementation.
 - Savings from generating code may be outweighed by the costs of developing translators for new platforms.



Model driven architecture

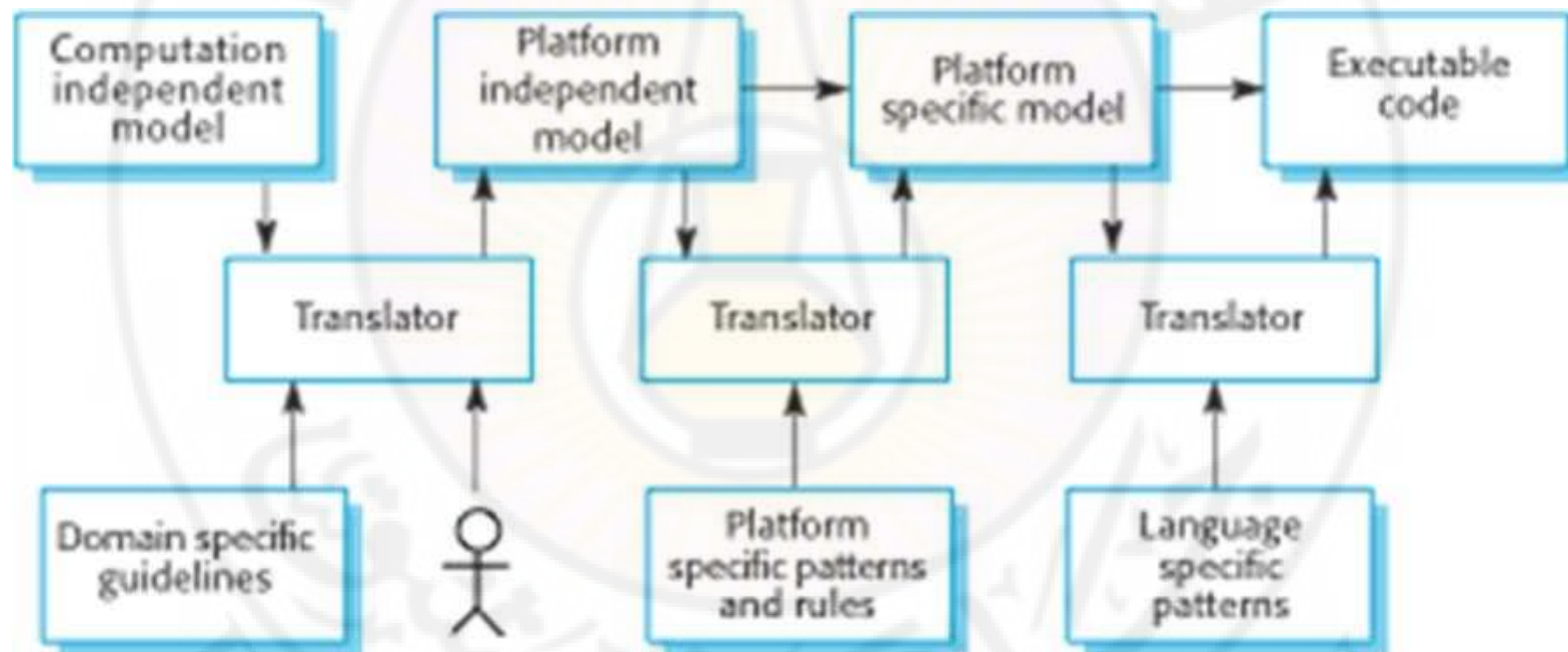
- ✧ Model-driven architecture (MDA) was the precursor of more general model-driven engineering
- ✧ MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- ✧ Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

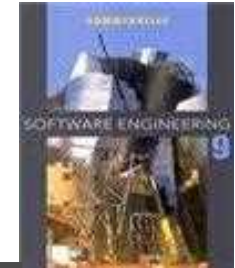
Types of model



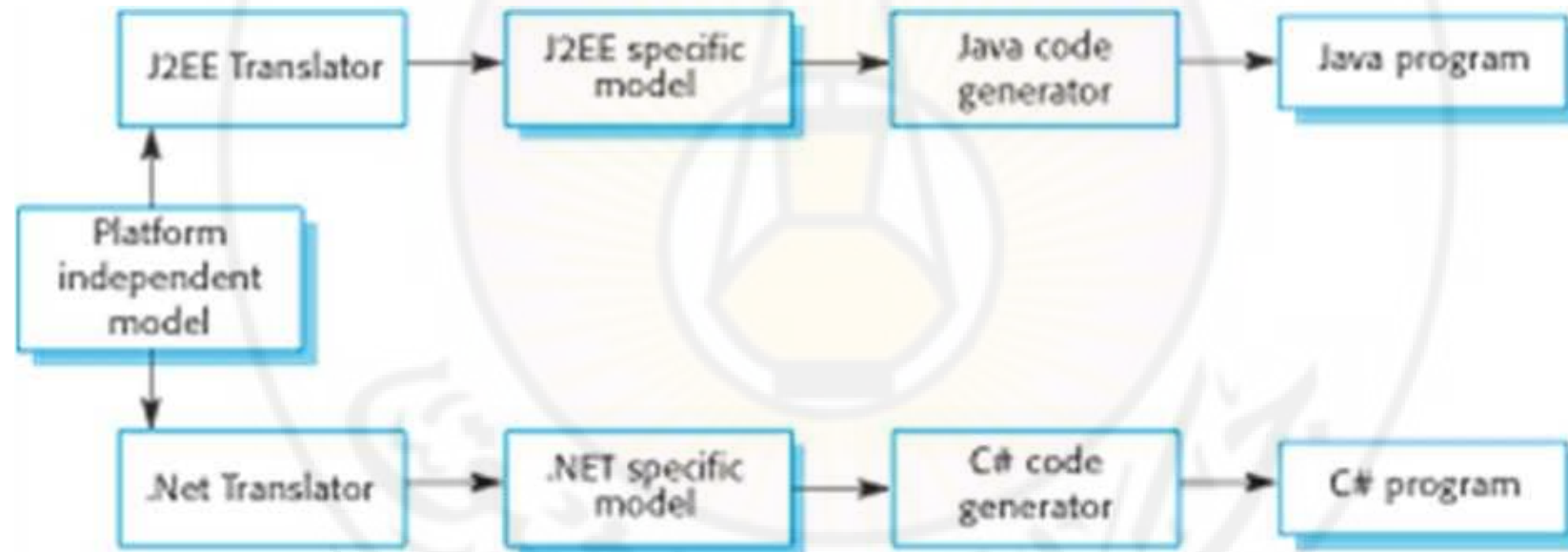
- ✧ A computation independent model (CIM)
 - These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
- ✧ A platform independent model (PIM)
 - These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
- ✧ *Platform specific models (PSM)*
 - These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA transformations

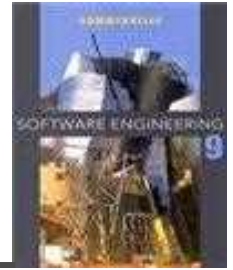




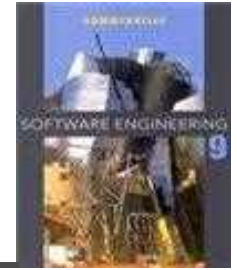
Multiple platform-specific models



Agile methods and MDA

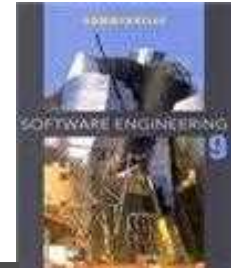


- ✧ The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
- ✧ The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.
- ✧ If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.



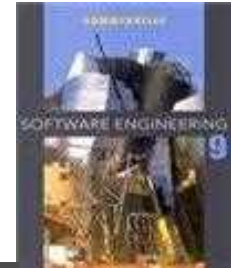
Executable UML

- ✧ The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
- ✧ This is possible using a subset of UML 2, called Executable UML or xUML.



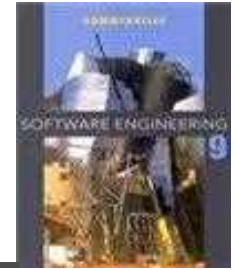
Features of executable UML

- ✧ To create an executable subset of UML, the number of model types has therefore been dramatically reduced to these 3 key types:
 - Domain models that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.
 - Class models in which classes are defined, along with their attributes and operations.
 - State models in which a state diagram is associated with each class and is used to describe the life cycle of the class.
- ✧ The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.



Key points

- ✧ A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- ✧ Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- ✧ Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- ✧ Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.



Key points

- ✧ Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
- ✧ Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- ✧ State diagrams are used to model a system's behavior in response to internal or external events.
- ✧ Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.



Chapter 4 – Requirements Engineering

Lecture 4

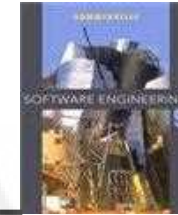
Dr. Mohammad Ahmad

Ph.D. in Software Engineering

MCA Master of Computer Application

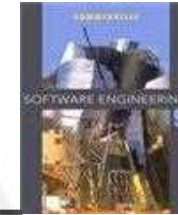
MBA-IT Master of Business Administration in Information Technology

Topics covered



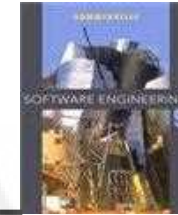
- ✧ Functional and non-functional requirements
- ✧ The software requirements document
- ✧ Requirements specification
- ✧ Requirements engineering processes
- ✧ Requirements elicitation and analysis
- ✧ Requirements validation
- ✧ Requirements management

Requirements engineering



- ✧ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- ✧ The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

What is a requirement?



- ✧ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ✧ This is inevitable as requirements may serve a dual function
 - May be the basis for a bid for a contract - therefore must be open to interpretation;
 - May be the basis for the contract itself - therefore must be defined in detail;
 - Both these statements may be called requirements.

Types of requirement



✧ User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

✧ System requirements

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Functional and non-functional requirements



✧ Functional requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

✧ Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

✧ Domain requirements

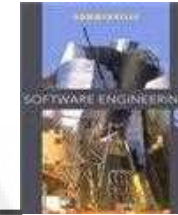
- Constraints on the system from the domain of operation

Functional requirements



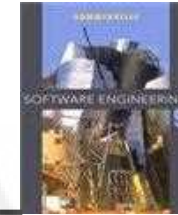
- ✧ Describe functionality or system services.
- ✧ Depend on the type of software, expected users and the type of system where the software is used.
- ✧ Functional user requirements may be high-level statements of what the system should do.
- ✧ Functional system requirements should describe the system services in detail.

Requirements completeness and consistency



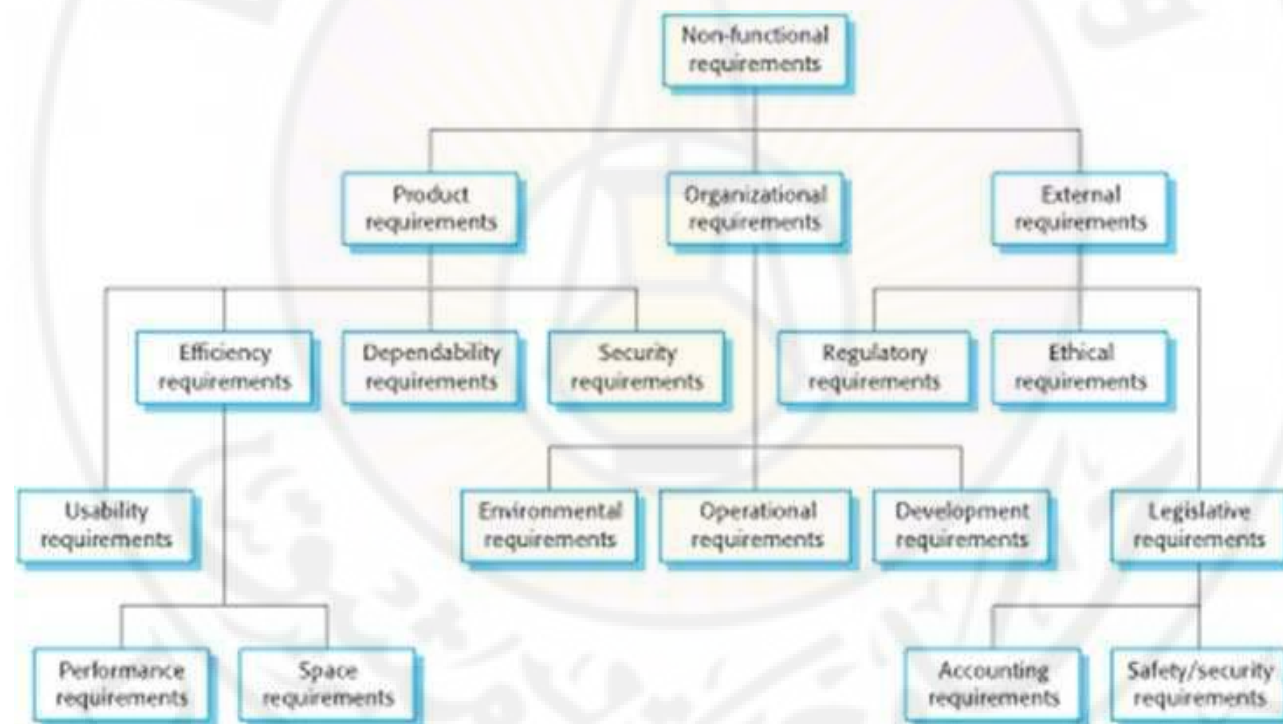
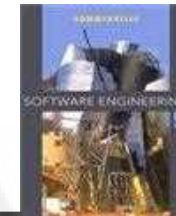
- ✧ In principle, requirements should be both complete and consistent.
- ✧ Complete
 - They should include descriptions of all facilities required.
- ✧ Consistent
 - There should be no conflicts or contradictions in the descriptions of the system facilities.
- ✧ In practice, it is impossible to produce a complete and consistent requirements document.

Non-functional requirements

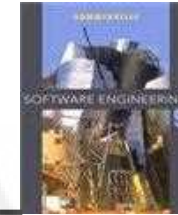


- ✧ These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- ✧ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ✧ Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Types of nonfunctional requirement



Non-functional requirements implementation



- ✧ Non-functional requirements may affect the overall architecture of a system rather than the individual components.
 - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- ✧ A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
 - It may also generate requirements that restrict existing requirements.

Non-functional classifications



✧ Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

✧ Organisational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

✧ External requirements

- Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Goals and requirements



- ✧ Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- ✧ Goal
 - A general intention of the user such as ease of use.
- ✧ Verifiable non-functional requirement
 - A statement using some measure that can be objectively tested.
- ✧ Goals are helpful to developers as they convey the intentions of the system users.

Metrics for specifying nonfunctional requirements



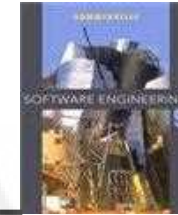
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Domain requirements



- ✧ The system's operational domain imposes requirements on the system.
 - For example, a train control system has to take into account the braking characteristics in different weather conditions.
- ✧ Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.
- ✧ If domain requirements are not satisfied, the system may be unworkable.

Domain requirements problems



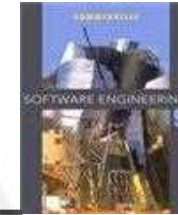
✧ Understandability

- Requirements are expressed in the language of the application domain;
- This is often not understood by software engineers developing the system.

✧ Implicitness

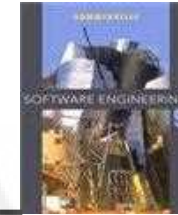
- Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

The software requirements document



- ✧ The software requirements document is the official statement of what is required of the system developers.
- ✧ Should include both a definition of user requirements and a specification of the system requirements.
- ✧ It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Requirements document variability



- ✧ Information in requirements document depends on type of system and the approach to development used.
- ✧ Systems developed incrementally will, typically, have less detail in the requirements document.
- ✧ Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

The structure of a requirements document



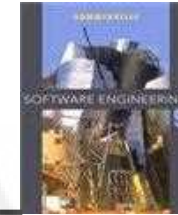
Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

The structure of a requirements document



Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements specification



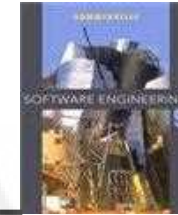
- ✧ The process of writing down the user and system requirements in a requirements document.
- ✧ User requirements have to be understandable by end-users and customers who do not have a technical background.
- ✧ System requirements are more detailed requirements and may include more technical information.
- ✧ The requirements may be part of a contract for the system development
 - It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification



Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and design



- ✧ In principle, requirements should state what the system should do and the design should describe how it does this.
- ✧ In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - The system may inter-operate with other systems that generate design requirements;
 - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
 - This may be the consequence of a regulatory requirement.

Natural language specification



- ✧ Requirements are written as natural language sentences supplemented by diagrams and tables.
- ✧ Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements



- ✧ Invent a standard format and use it for all requirements.
- ✧ Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- ✧ Use text highlighting to identify key parts of the requirement.
- ✧ Avoid the use of computer jargon.
- ✧ Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language



✧ Lack of clarity

- Precision is difficult without making the document difficult to read.

✧ Requirements confusion

- Functional and non-functional requirements tend to be mixed-up.

✧ Requirements amalgamation

- Several different requirements may be expressed together.

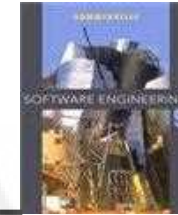
Example requirements for the insulin pump software system



3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. *(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)*

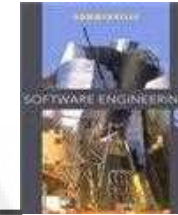
3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. *(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)*

Structured specifications



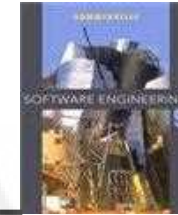
- ✧ An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- ✧ This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications



- ✧ Definition of the function or entity.
- ✧ Description of inputs and where they come from.
- ✧ Description of outputs and where they go to.
- ✧ Information about the information needed for the computation and other entities used.
- ✧ Description of the action to be taken.
- ✧ Pre and post conditions (if appropriate).
- ✧ The side effects (if any) of the function.

Tabular specification



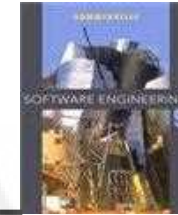
- ✧ Used to supplement natural language.
- ✧ Particularly useful when you have to define a number of possible alternative courses of action.
- ✧ For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Tabular specification of computation for an insulin pump



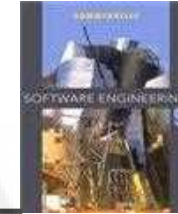
Condition	Action
Sugar level falling ($r2 < r1$)	CompDose = 0
Sugar level stable ($r2 = r1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing ($(r2 - r1) < (r1 - r0)$)	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ($(r2 - r1) \geq (r1 - r0)$)	CompDose = round $((r2 - r1)/4)$ If rounded result = 0 then CompDose = MinimumDose

Requirements engineering processes



- ✧ The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- ✧ However, there are a number of generic activities common to all processes
 - Requirements elicitation;
 - Requirements analysis;
 - Requirements validation;
 - Requirements management.
- ✧ In practice, RE is an iterative activity in which these processes are interleaved.

Requirements elicitation and analysis



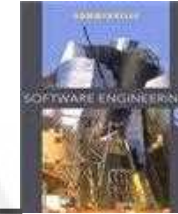
- ✧ Sometimes called requirements elicitation or requirements discovery.
- ✧ Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- ✧ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Problems of requirements analysis



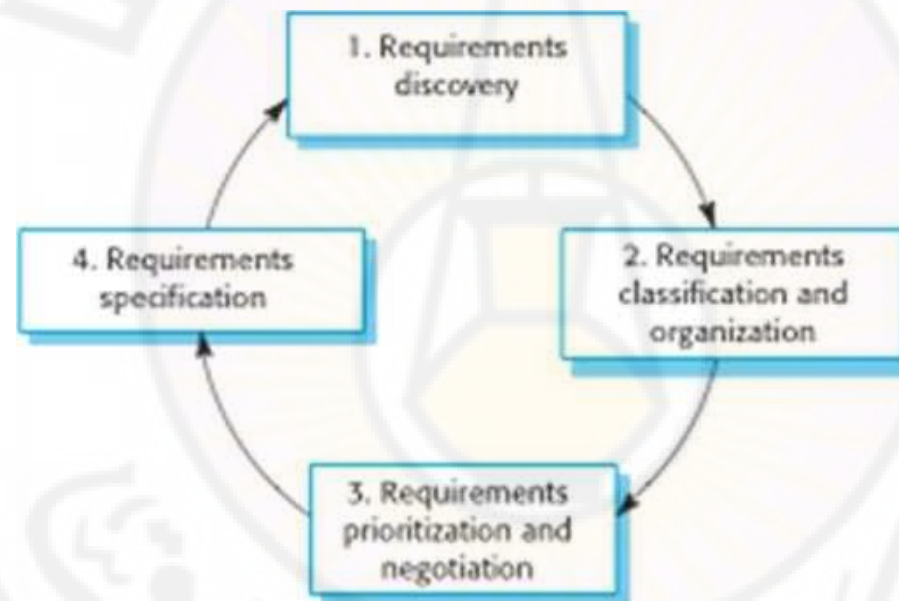
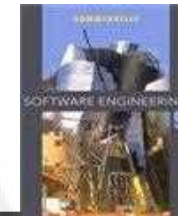
- ✧ Stakeholders don't know what they really want.
- ✧ Stakeholders express requirements in their own terms.
- ✧ Different stakeholders may have conflicting requirements.
- ✧ Organisational and political factors may influence the system requirements.
- ✧ The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

Requirements elicitation and analysis



- ✧ Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
- ✧ Stages include:
 - Requirements discovery,
 - Requirements classification and organization,
 - Requirements prioritization and negotiation,
 - Requirements specification.

The requirements elicitation and analysis process



Process activities



✧ Requirements discovery

- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

✧ Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

✧ Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

✧ Requirements specification

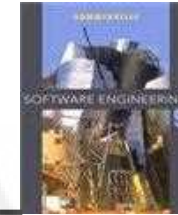
- Requirements are documented and input into the next round of the spiral.

Problems of requirements elicitation



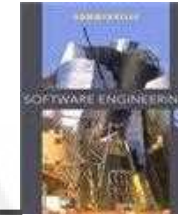
- ✧ Stakeholders don't know what they really want.
- ✧ Stakeholders express requirements in their own terms.
- ✧ Different stakeholders may have conflicting requirements.
- ✧ Organisational and political factors may influence the system requirements.
- ✧ The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

Requirements discovery



- ✧ The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.
- ✧ Interaction is with system stakeholders from managers to external regulators.
- ✧ Systems normally have a range of stakeholders.

Interviewing



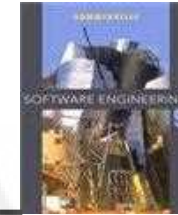
- ✧ Formal or informal interviews with stakeholders are part of most RE processes.
- ✧ Types of interview
 - Closed interviews based on pre-determined list of questions
 - Open interviews where various issues are explored with stakeholders.
- ✧ Effective interviewing
 - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
 - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

Interviews in practice



- ✧ Normally a mix of closed and open-ended interviewing.
- ✧ Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- ✧ Interviews are not good for understanding domain requirements
 - Requirements engineers cannot understand specific domain terminology;
 - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Scenarios



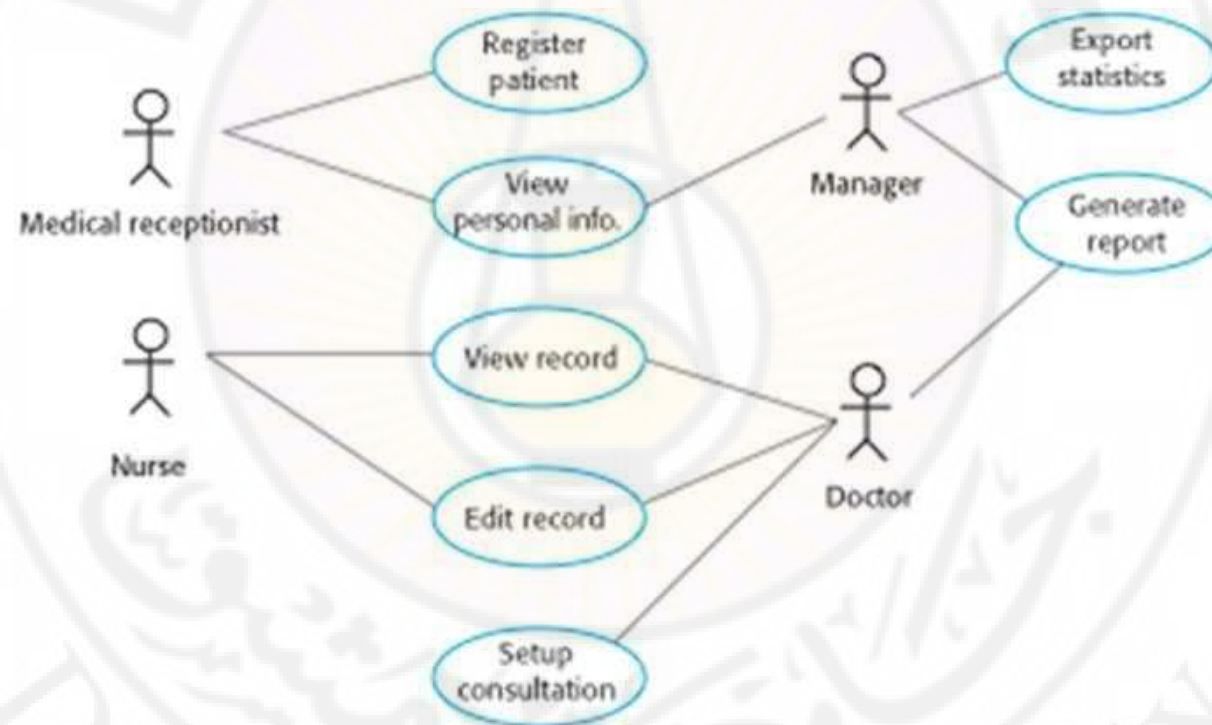
- ✧ Scenarios are real-life examples of how a system can be used.
- ✧ They should include
 - A description of the starting situation;
 - A description of the normal flow of events;
 - A description of what can go wrong;
 - Information about other concurrent activities;
 - A description of the state when the scenario finishes.

Use cases

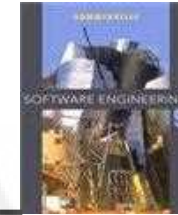


- ✧ Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- ✧ A set of use cases should describe all possible interactions with the system.
- ✧ High-level graphical model supplemented by more detailed tabular description (see Chapter 5).
- ✧ Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

Use cases for the MHC-PMS

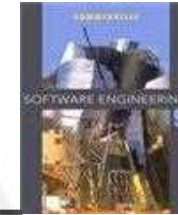


Requirements validation



- ✧ Concerned with demonstrating that the requirements define the system that the customer really wants.
- ✧ Requirements error costs are high so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking



- ✧ **Validity**. Does the system provide the functions which best support the customer's needs?
- ✧ **Consistency**. Are there any requirements conflicts?
- ✧ **Completeness**. Are all functions required by the customer included?
- ✧ **Realism**. Can the requirements be implemented given available budget and technology
- ✧ **Verifiability**. Can the requirements be checked?

Requirements validation techniques



- ✧ Requirements reviews
 - Systematic manual analysis of the requirements.
- ✧ Prototyping
 - Using an executable model of the system to check requirements.
- ✧ Test-case generation
 - Developing tests for requirements to check testability.

Requirements reviews



- ✧ Regular reviews should be held while the requirements definition is being formulated.
- ✧ Both client and contractor staff should be involved in reviews.
- ✧ Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks



✧ Verifiability

- Is the requirement realistically testable?

✧ Comprehensibility

- Is the requirement properly understood?

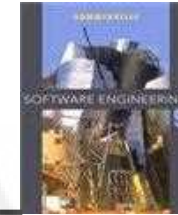
✧ Traceability

- Is the origin of the requirement clearly stated?

✧ Adaptability

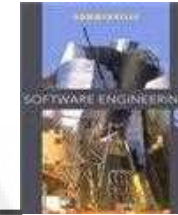
- Can the requirement be changed without a large impact on other requirements?

Requirements management



- ✧ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- ✧ New requirements emerge as a system is being developed and after it has gone into use.
- ✧ You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

Changing requirements



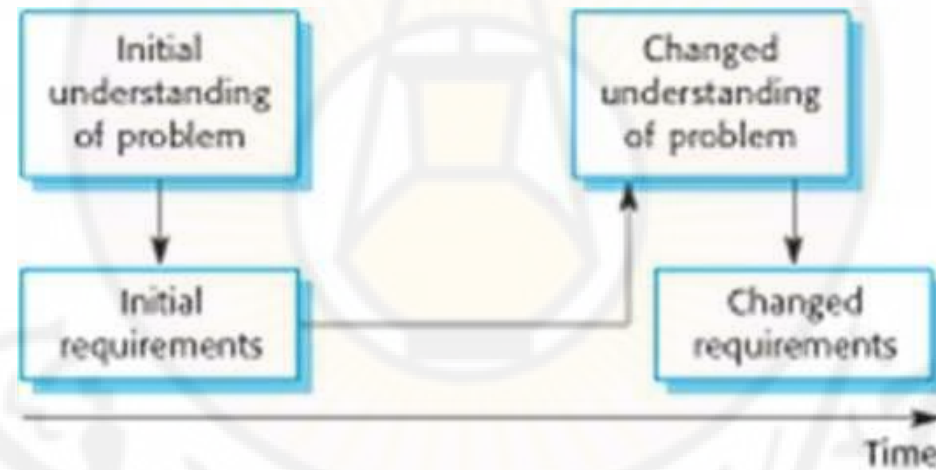
- ✧ The business and technical environment of the system always changes after installation.
 - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- ✧ The people who pay for a system and the users of that system are rarely the same people.
 - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

Changing requirements

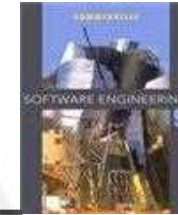


- ✧ Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
 - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements evolution

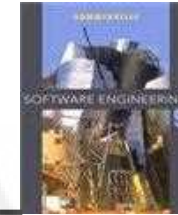


Requirements management planning



- ✧ Establishes the level of requirements management detail that is required.
- ✧ Requirements management decisions:
 - *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
 - *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
 - *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
 - *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management



✧ Deciding if a requirements change should be accepted

- *Problem analysis and change specification*

- During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

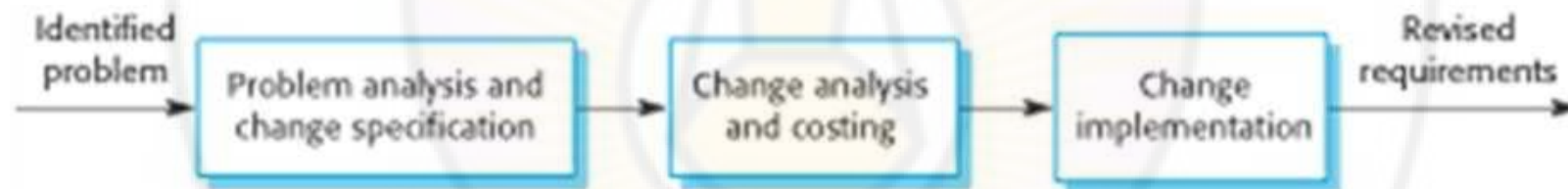
- *Change analysis and costing*

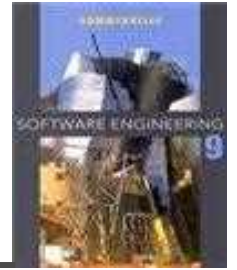
- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

- *Change implementation*

- The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Requirements change management



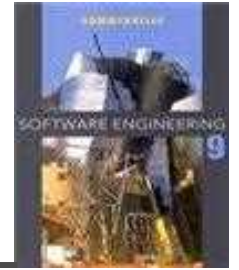


Architectural Design

Lecture ٦

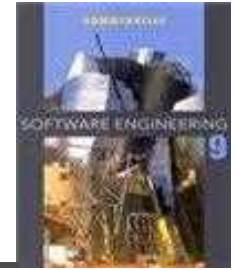
Dr. Mohammad Ahmad

Topics covered



- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

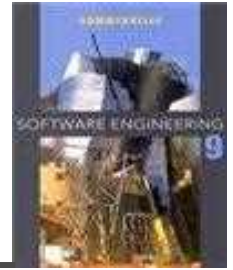




Software architecture

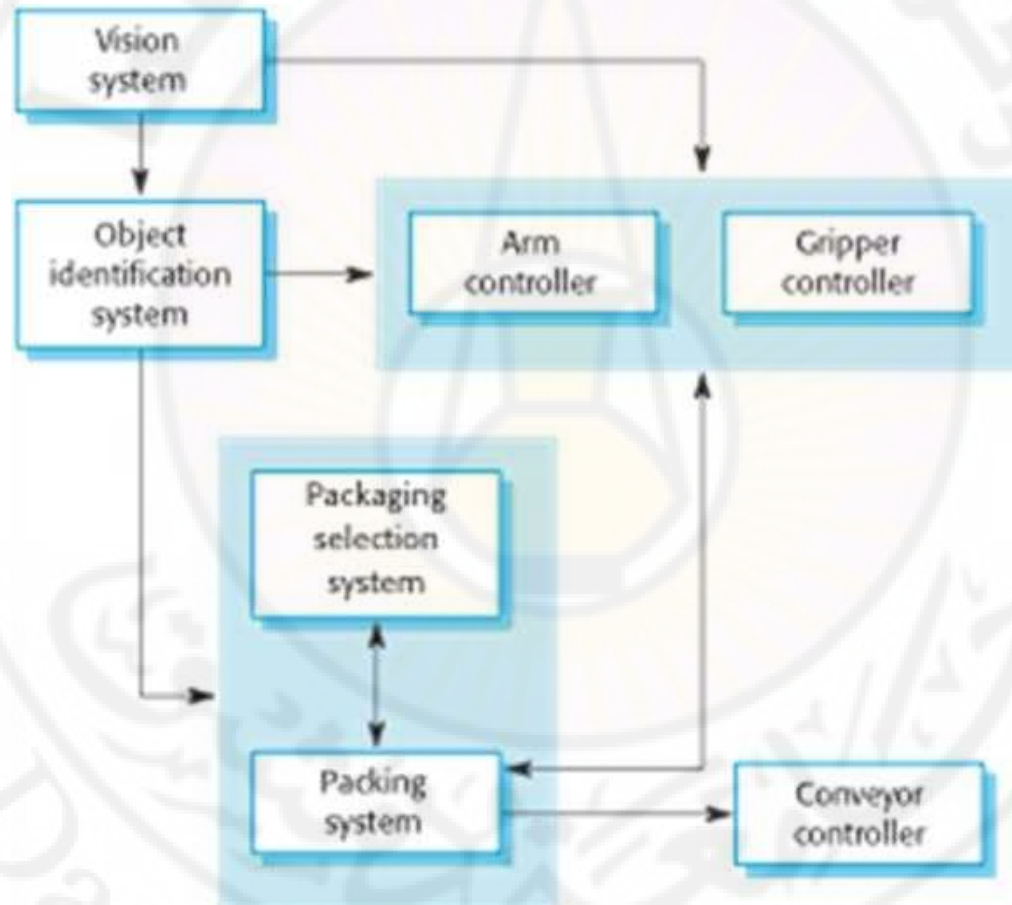
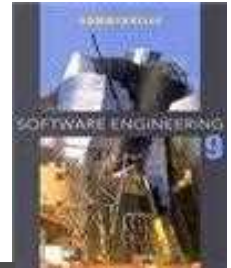
- ✧ The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is **architectural design**.
- ✧ The output of this design process is a description of the **software architecture**.

Architectural design

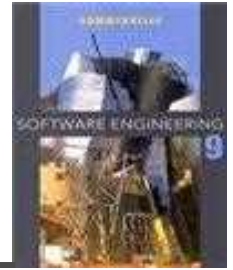


- ✧ An early stage of the system design process.
- ✧ Represents the link between specification and design processes.
- ✧ Often carried out in parallel with some specification activities.
- ✧ It involves identifying major system components and their communications.

The architecture of a packing robot control system

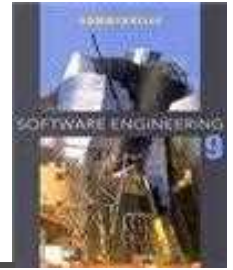


Architectural abstraction



- ✧ **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- ✧ **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture



✧ Stakeholder communication

- Architecture may be used as a focus of discussion by system stakeholders.

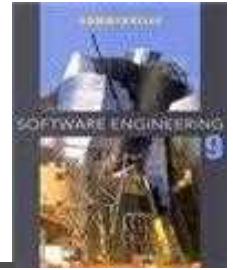
✧ System analysis

- Means that analysis of whether the system can meet its non-functional requirements is possible.

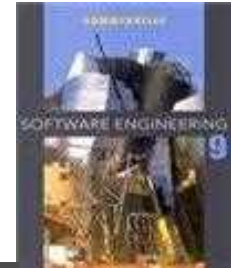
✧ Large-scale reuse

- The architecture may be reusable across a range of systems
- Product-line architectures may be developed.

Architectural representations

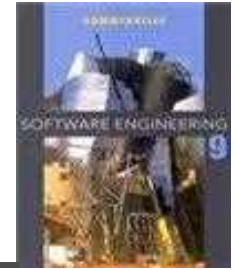


- ✧ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- ✧ But these have been criticized because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- ✧ Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.



Box and line diagrams

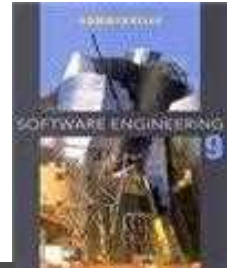
- ✧ Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- ✧ However, useful for communication with stakeholders and for project planning.



Use of architectural models

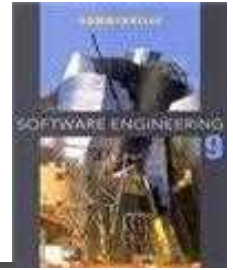
- ✧ As a way of facilitating discussion about the system design
 - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- ✧ As a way of documenting an architecture that has been designed
 - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural design decisions



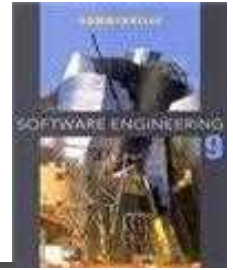
- ✧ Architectural design is a creative process so the process differs depending on the type of system being developed.
- ✧ However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architectural design decisions



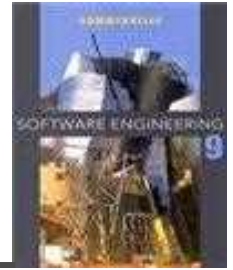
- ✧ Is there a generic application architecture that can be used?
- ✧ How will the system be distributed?
- ✧ What architectural styles are appropriate?
- ✧ What approach will be used to structure the system?
- ✧ How will the system be decomposed into modules?
- ✧ What control strategy should be used?
- ✧ How will the architectural design be evaluated?
- ✧ How should the architecture be documented?

Architecture reuse



- ✧ Systems in the same domain often have similar architectures that reflect domain concepts.
- ✧ Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- ✧ The architecture of a system may be designed around one of more architectural patterns or 'styles'.
 - These capture the essence of an architecture and can be instantiated in different ways.
 - Discussed later in this lecture.

Architecture and system characteristics



✧ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components.

✧ Security

- Use a layered architecture with critical assets in the inner layers.

✧ Safety

- Localize safety-critical features in a small number of sub-systems.

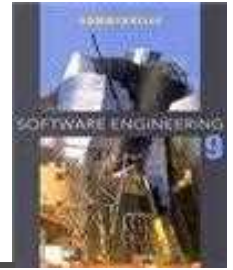
✧ Availability

- Include redundant components and mechanisms for fault tolerance.

✧ Maintainability

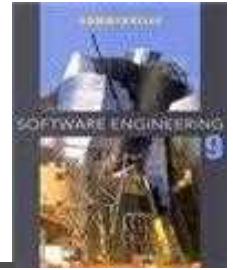
- Use fine-grain, replaceable components.

Architectural views



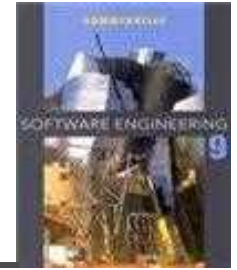
- ✧ What views or perspectives are useful when designing and documenting a system's architecture?
- ✧ What notations should be used for describing architectural models?
- ✧ Each architectural model only shows one view or perspective of the system.
 - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

4 + 1 view model of software architecture



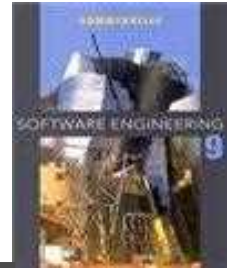
- ✧ A logical view, which shows the key abstractions in the system as objects or object classes.
- ✧ A process view, which shows how, at run-time, the system is composed of interacting processes.
- ✧ A development view, which shows how the software is decomposed for development.
- ✧ A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- ✧ Related using use cases or scenarios (+1)

Architectural patterns



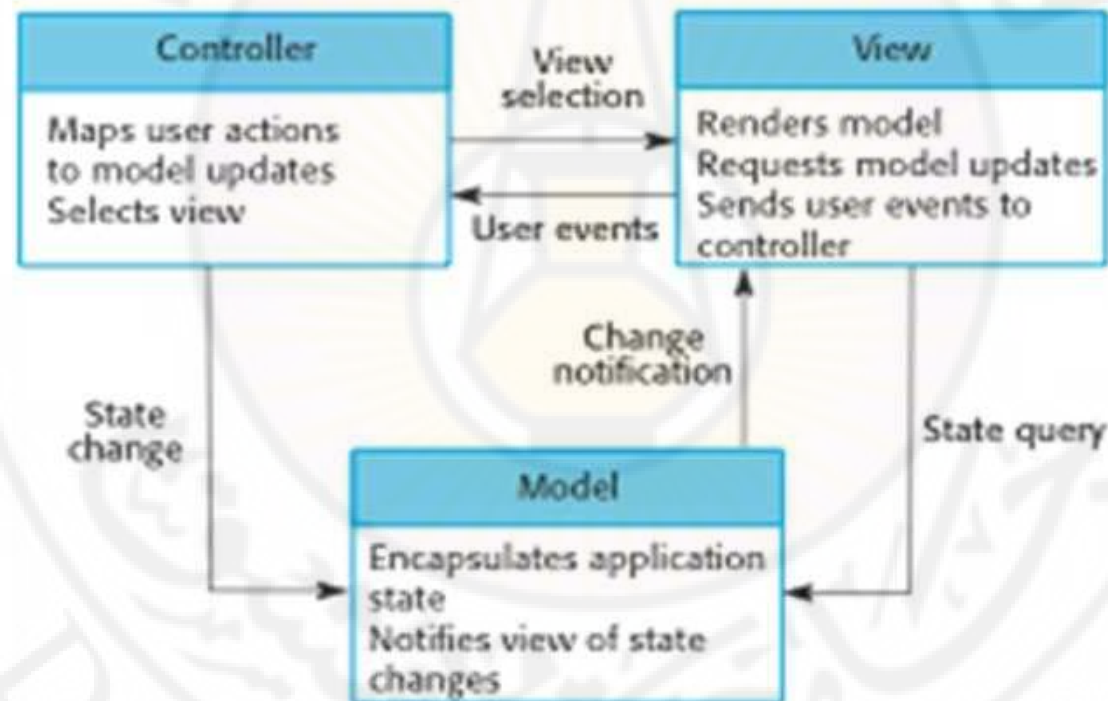
- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.
- ✧ Patterns may be represented using tabular and graphical descriptions.

The Model-View-Controller (MVC) pattern

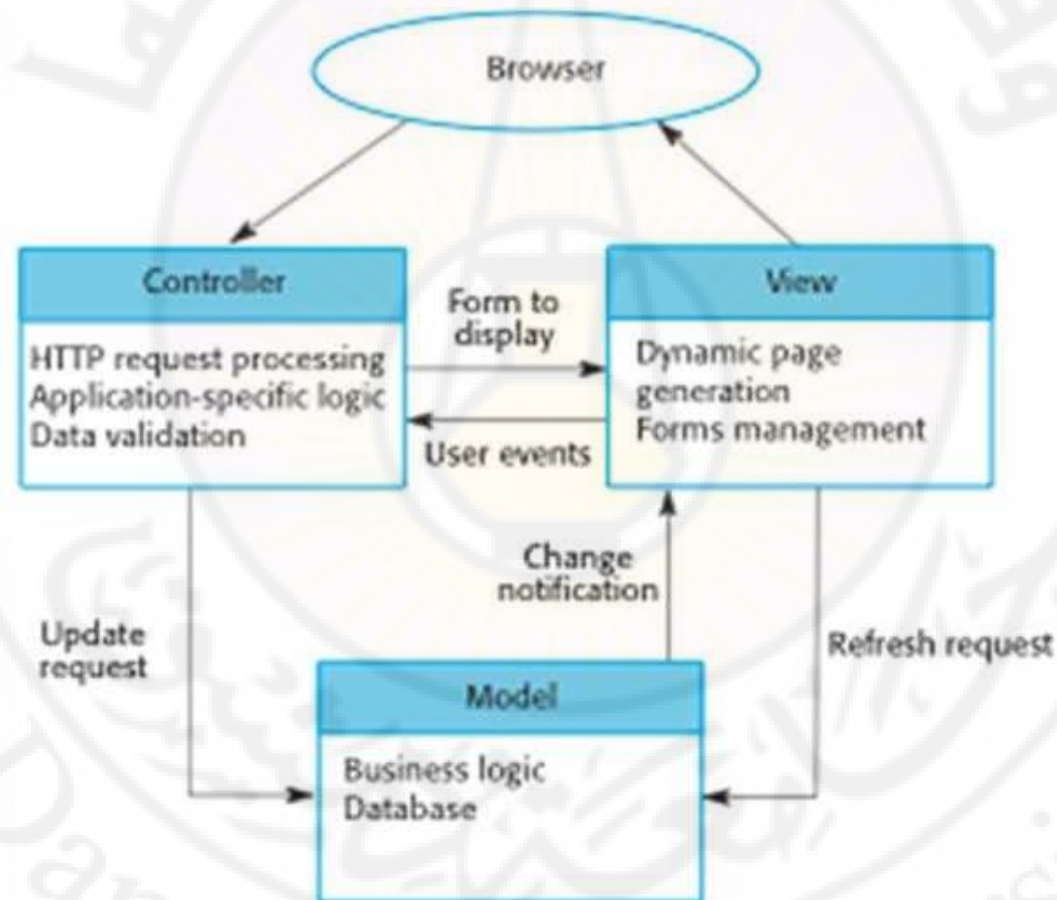
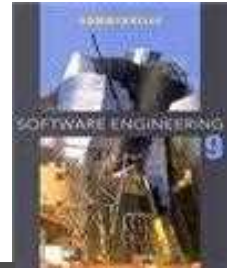


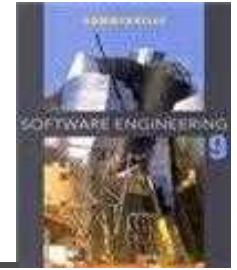
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern

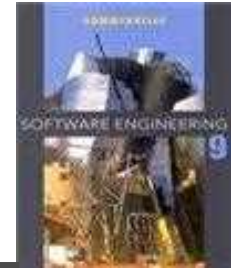




Layered architecture

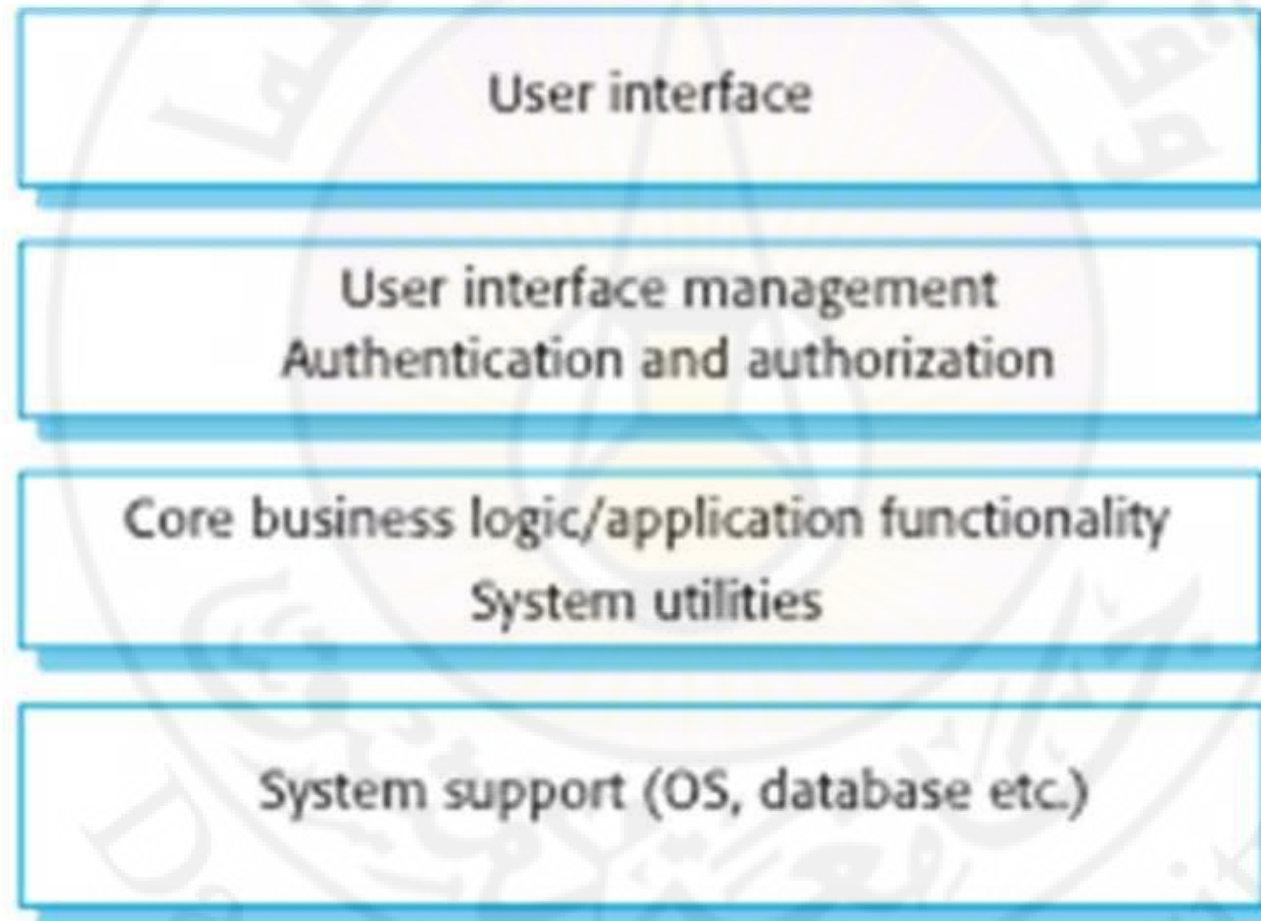
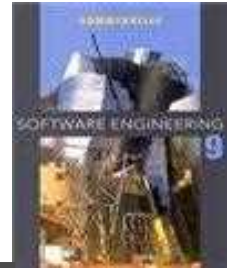
- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

The Layered architecture pattern

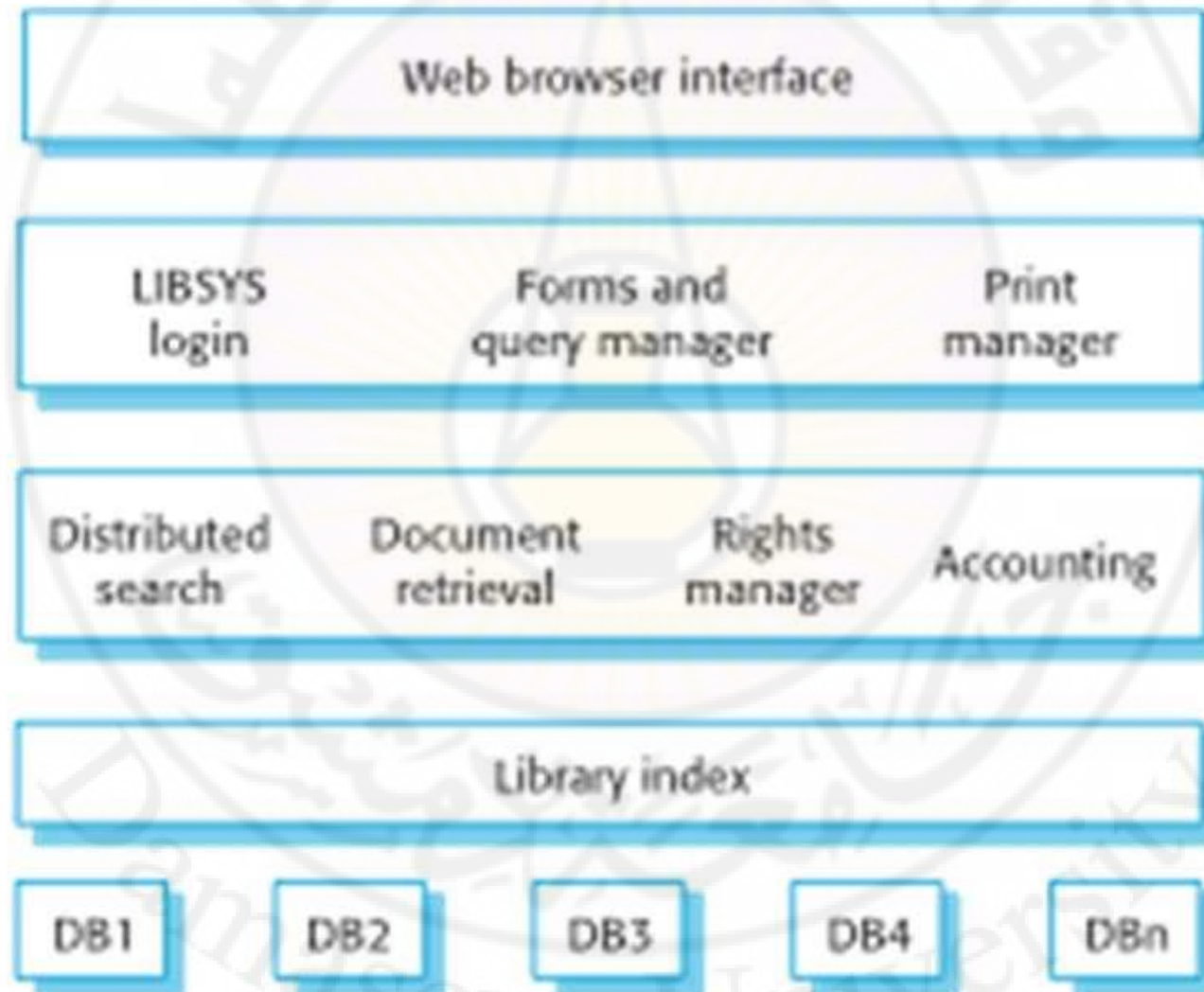
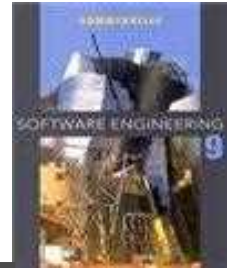


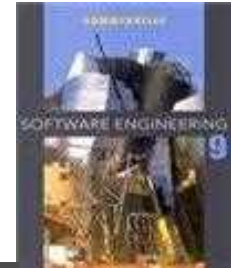
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A generic layered architecture



The architecture of the LIBSYS system

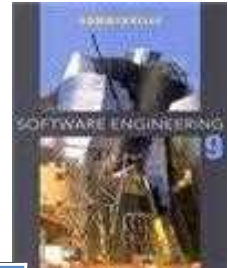




Repository architecture

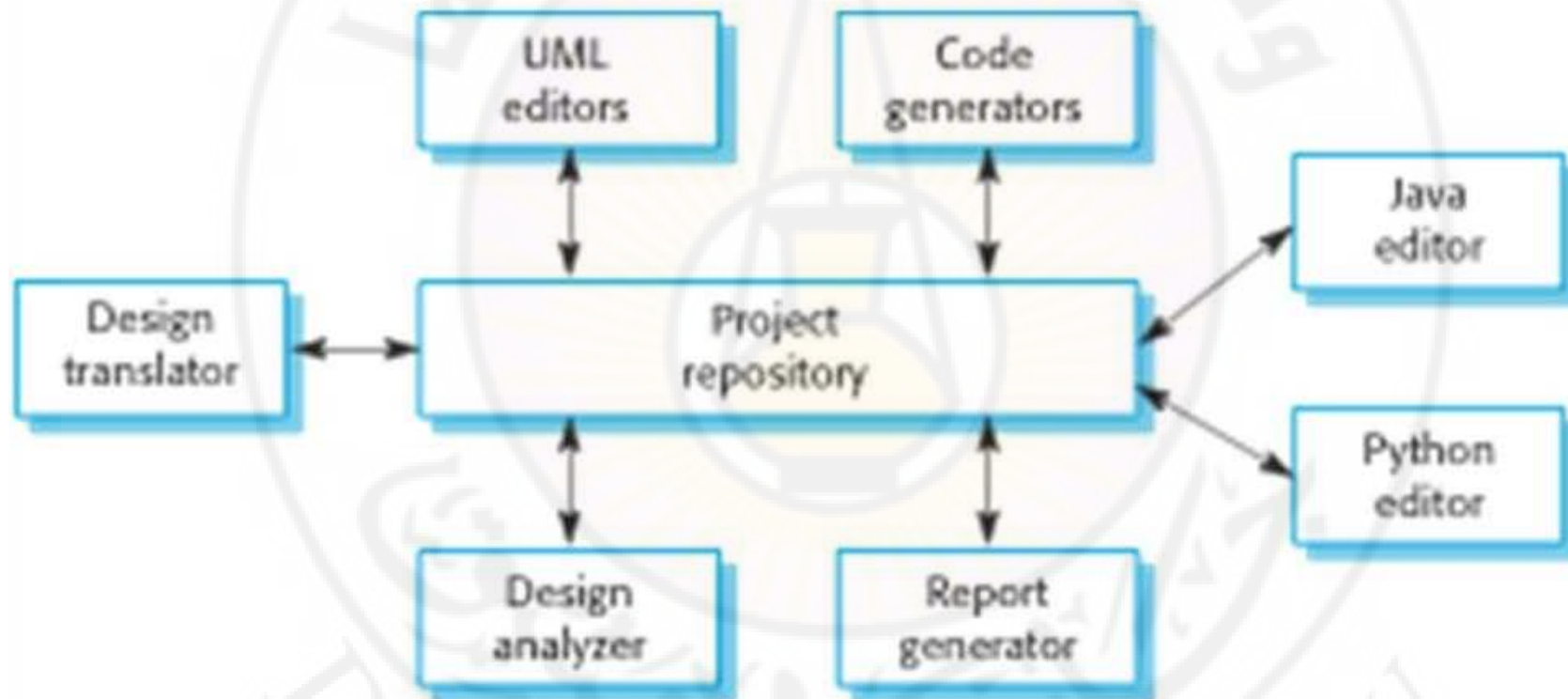
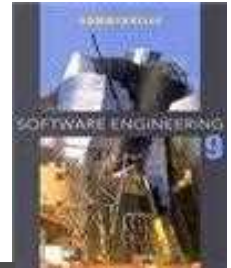
- ✧ Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

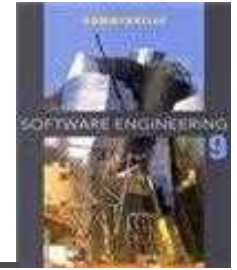
The Repository pattern



Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

A repository architecture for an IDE

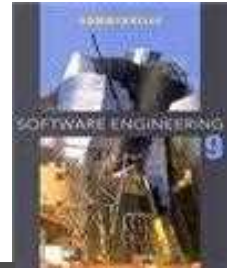




Client-server architecture

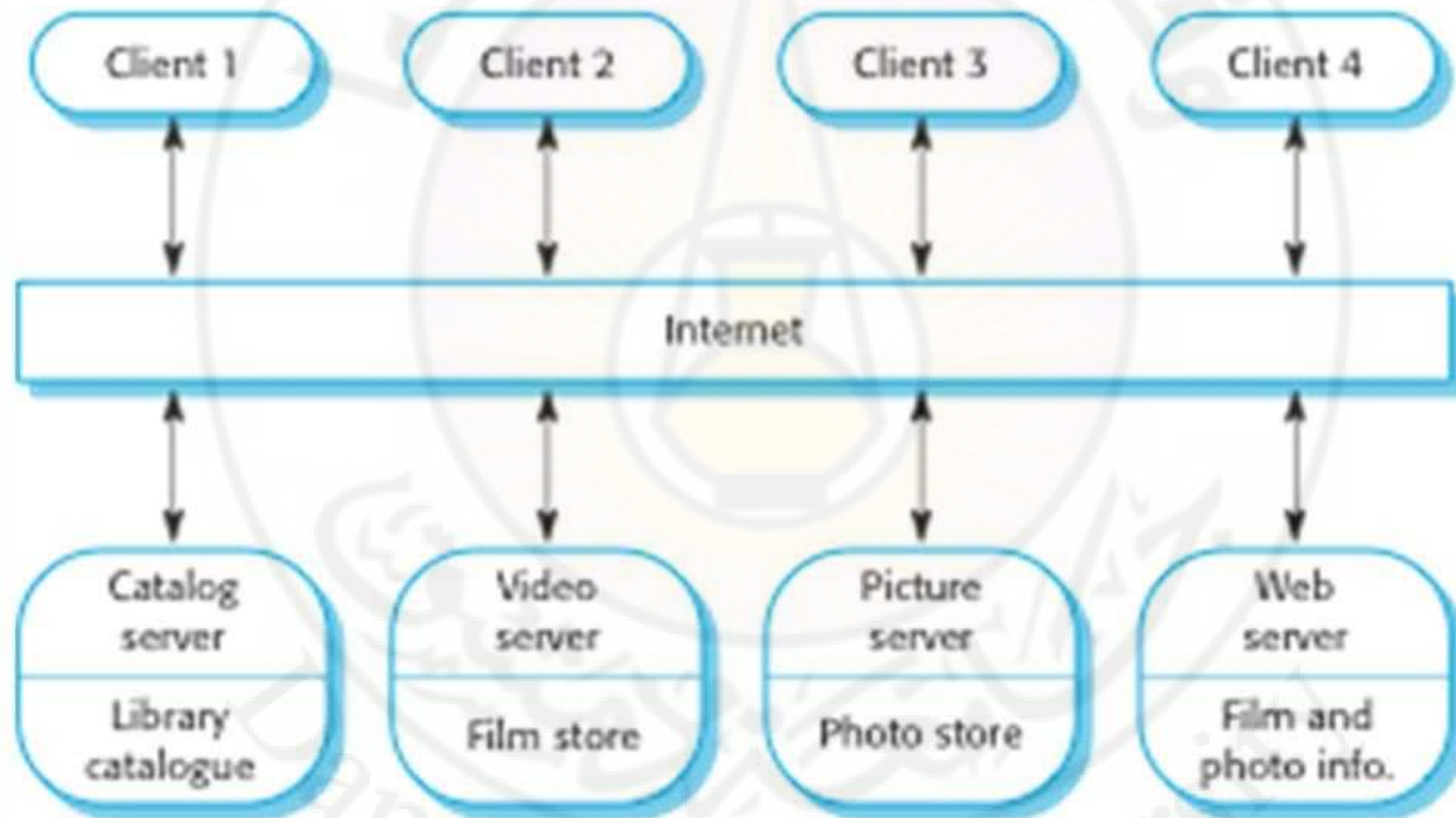
- ✧ Distributed system model which shows how data and processing is distributed across a range of components.
 - Can be implemented on a single computer.
- ✧ Set of stand-alone servers which provide specific services such as printing, data management, etc.
- ✧ Set of clients which call on these services.
- ✧ Network which allows clients to access servers.

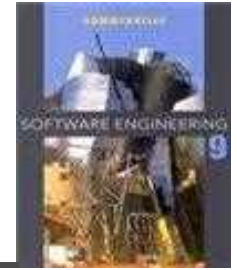
The Client–server pattern



Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A client–server architecture for a film library

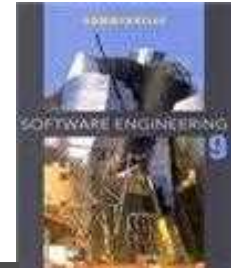




Pipe and filter architecture

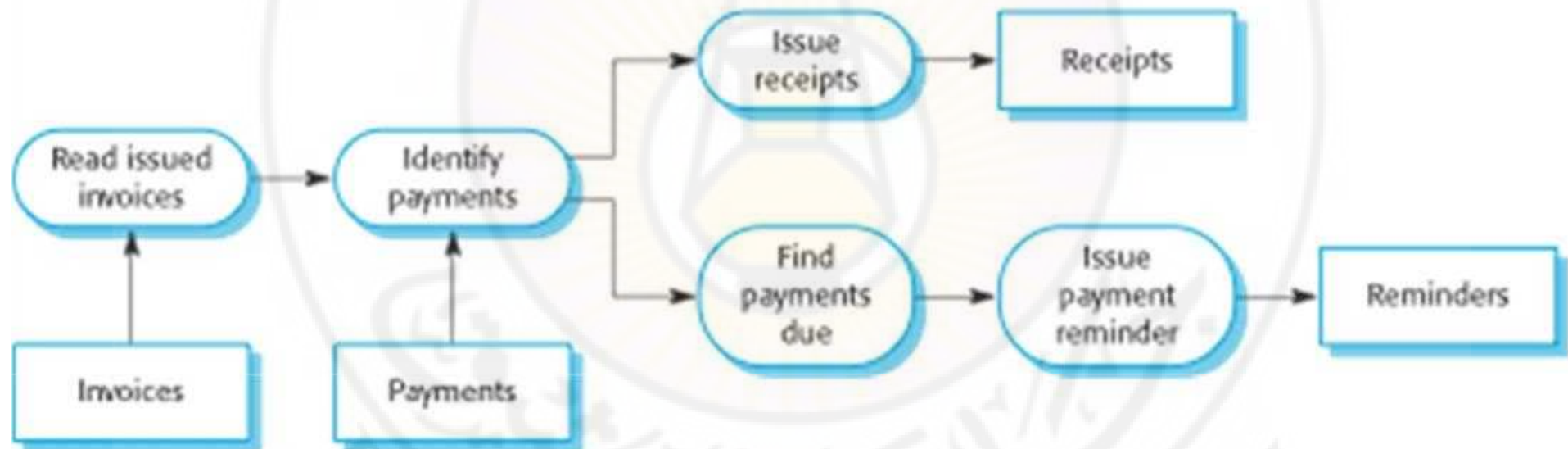
- ✧ Functional transformations process their inputs to produce outputs.
- ✧ May be referred to as a pipe and filter model (as in UNIX shell).
- ✧ Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- ✧ Not really suitable for interactive systems.

The pipe and filter pattern

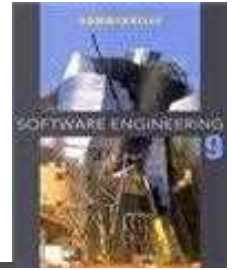


Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

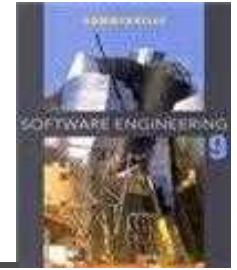
An example of the pipe and filter architecture



Application architectures

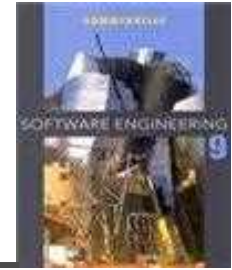


- ✧ Application systems are designed to meet an organizational need.
- ✧ As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- ✧ A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.



Use of application architectures

- ✧ As a starting point for architectural design.
- ✧ As a design checklist.
- ✧ As a way of organising the work of the development team.
- ✧ As a means of assessing components for reuse.
- ✧ As a vocabulary for talking about application types.



Examples of application types

✧ Data processing applications

- Data driven applications that process data in batches without explicit user intervention during the processing.

✧ Transaction processing applications

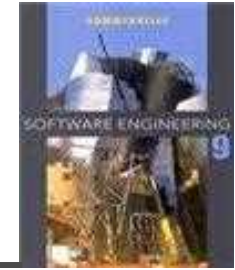
- Data-centred applications that process user requests and update information in a system database.

✧ Event processing systems

- Applications where system actions depend on interpreting events from the system's environment.

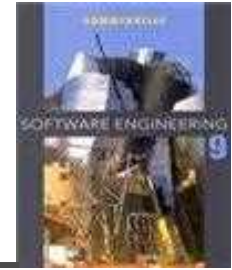
✧ Language processing systems

- Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.



Application type examples

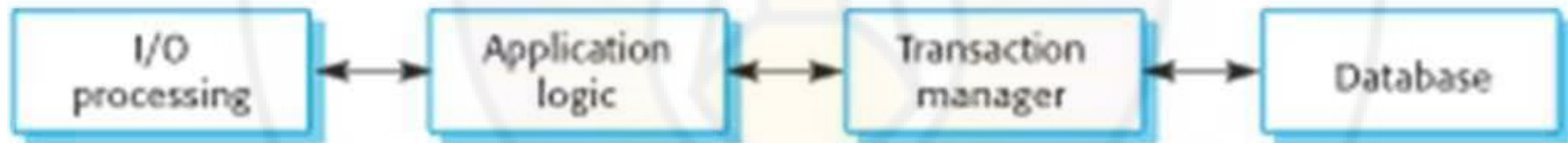
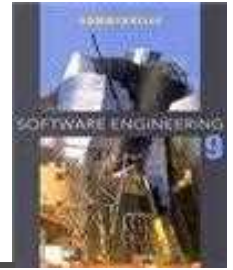
- ✧ Focus here is on transaction processing and language processing systems.
- ✧ Transaction processing systems
 - E-commerce systems;
 - Reservation systems.
- ✧ Language processing systems
 - Compilers;
 - Command interpreters.



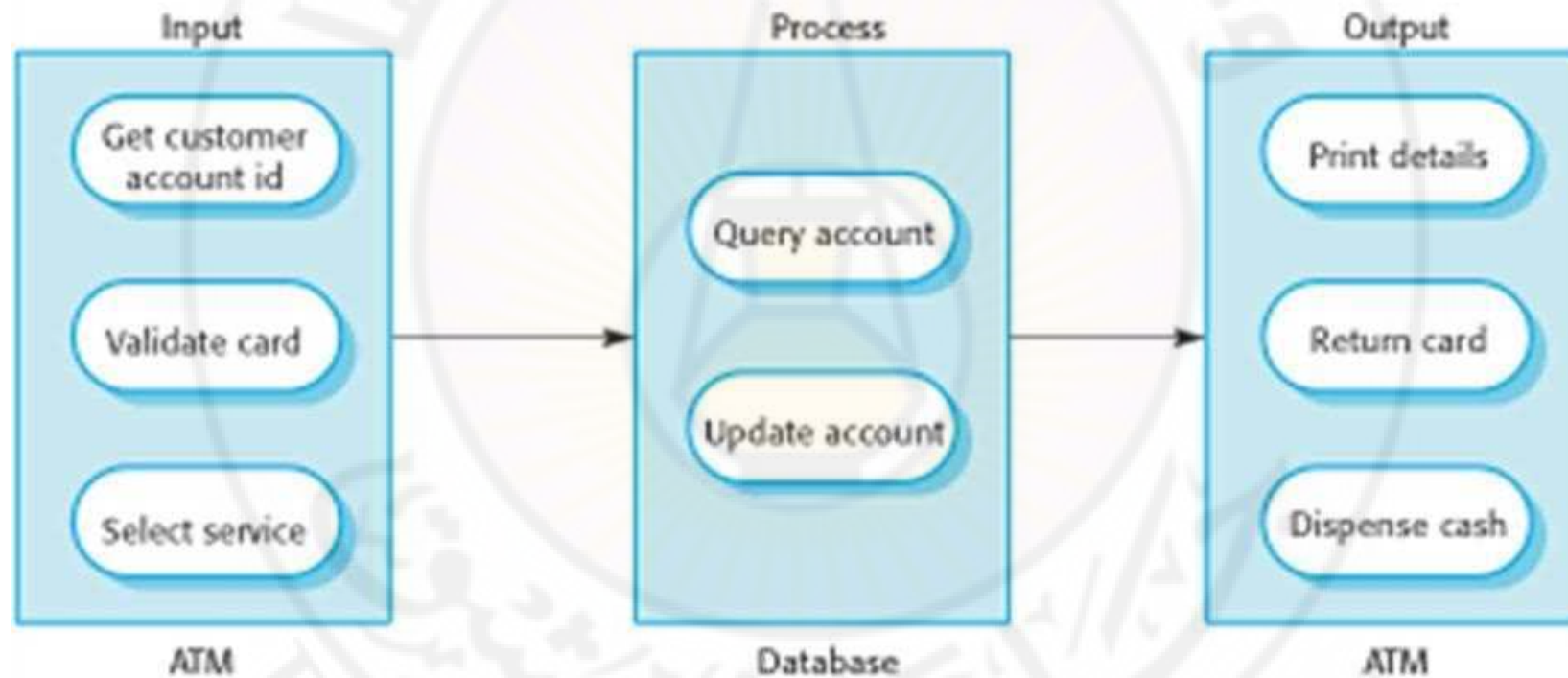
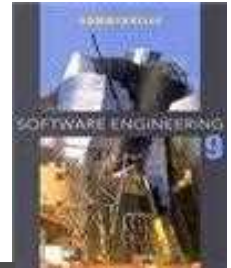
Transaction processing systems

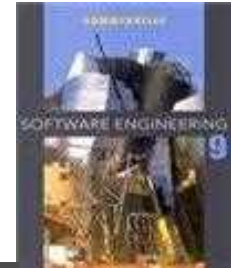
- ✧ Process user requests for information from a database or requests to update the database.
- ✧ From a user perspective a transaction is:
 - Any coherent sequence of operations that satisfies a goal;
 - For example - find the times of flights from London to Paris.
- ✧ Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications



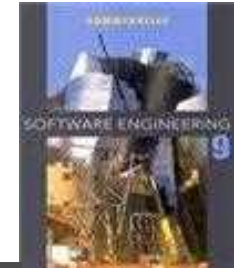
The software architecture of an ATM system



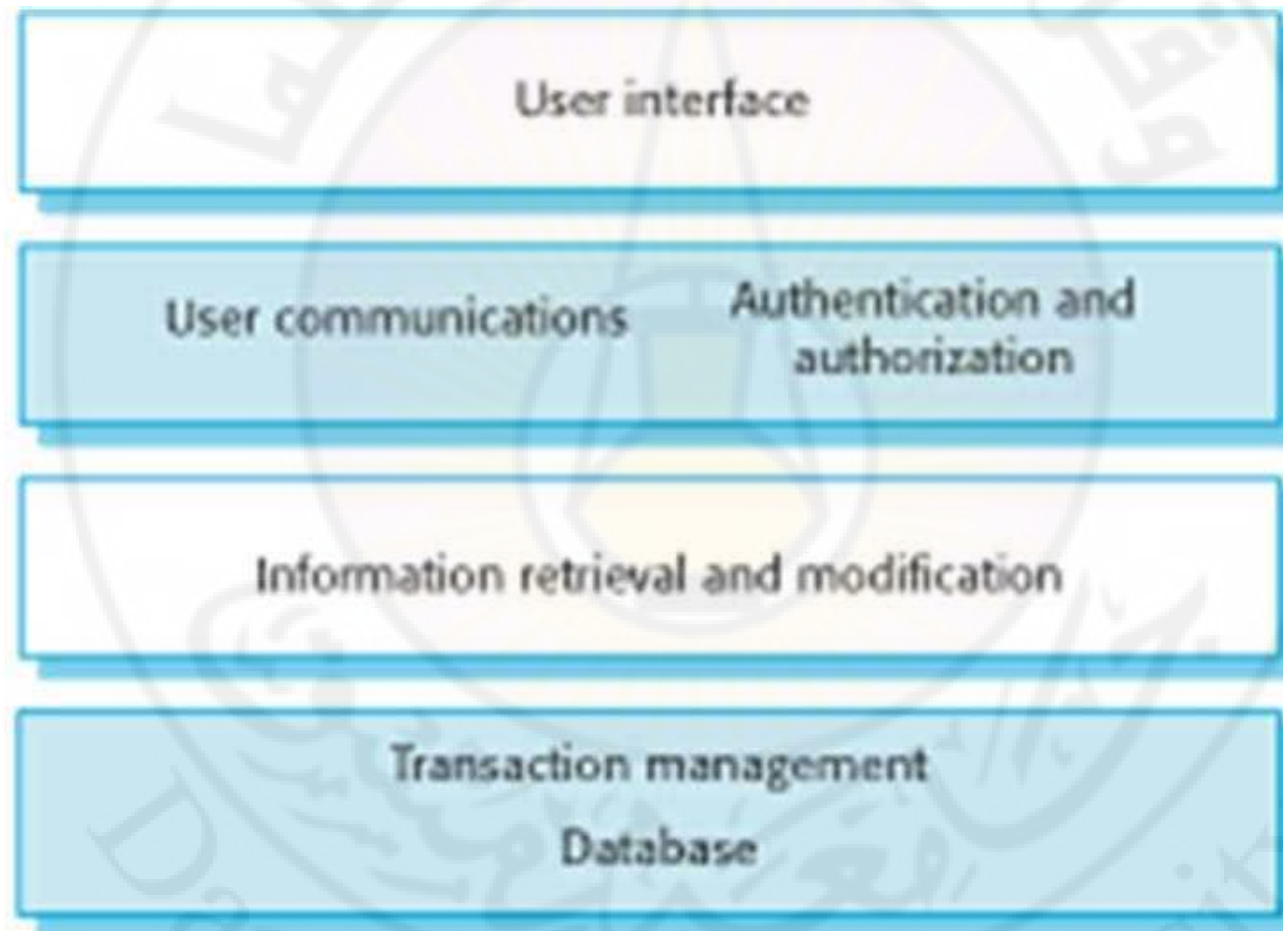


Information systems architecture

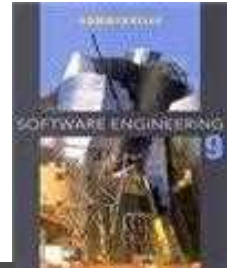
- ✧ Information systems have a generic architecture that can be organized as a layered architecture.
- ✧ These are transaction-based systems as interaction with these systems generally involves database transactions.
- ✧ Layers include:
 - The user interface
 - User communications
 - Information retrieval
 - System database



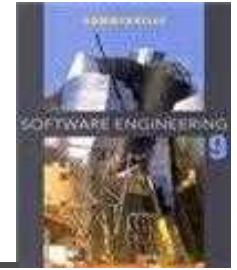
Layered information system architecture



Web-based information systems

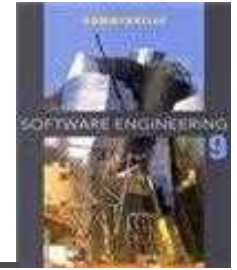


- ✧ Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
- ✧ For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer.
- ✧ In an e-commerce system, the application-specific layer includes additional functionality supporting a 'shopping cart' in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.



Server implementation

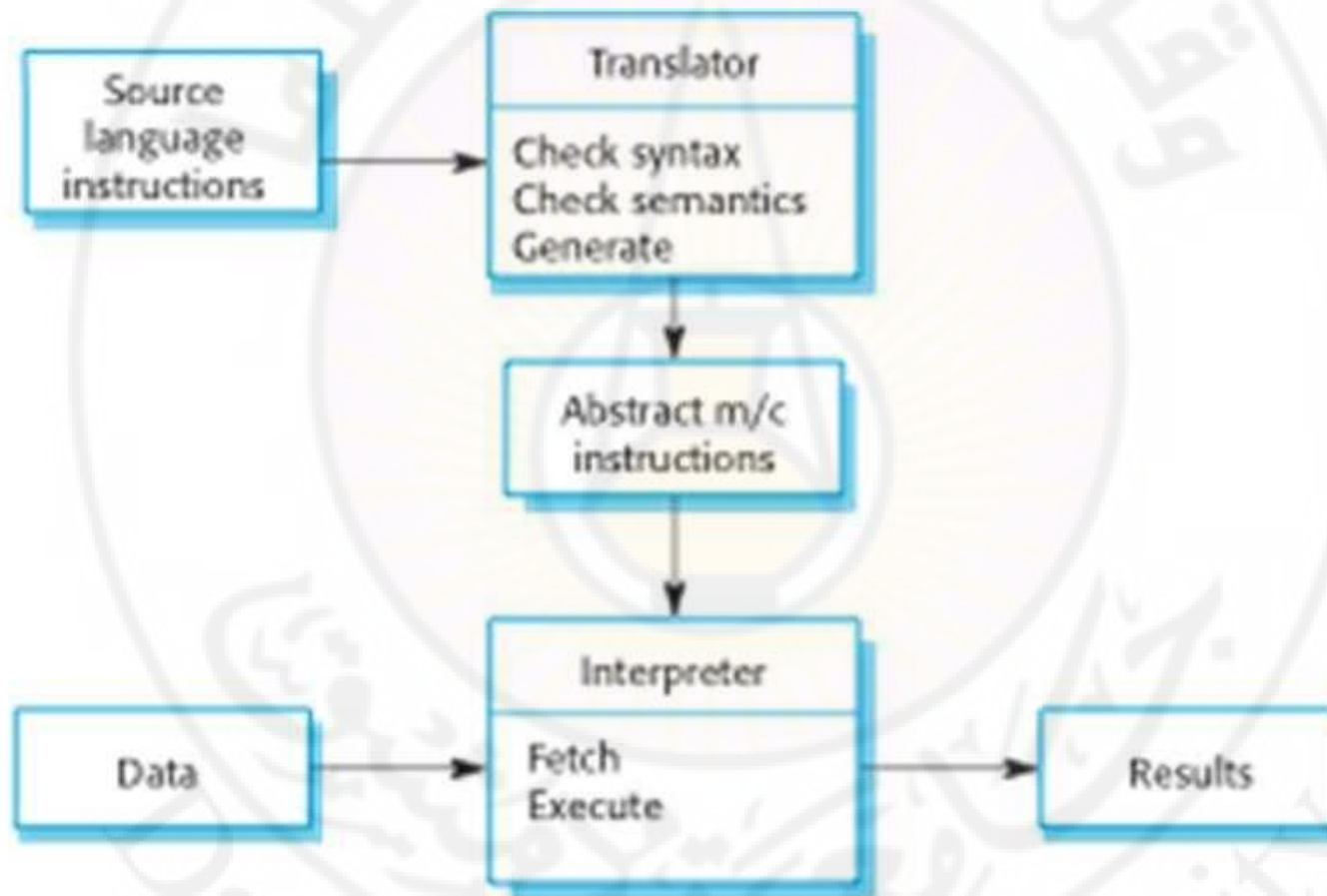
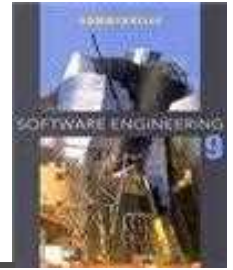
- ✧ These systems are often implemented as multi-tier client server/architectures
 - The web server is responsible for all user communications, with the user interface implemented using a web browser;
 - The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
 - The database server moves information to and from the database and handles transaction management.

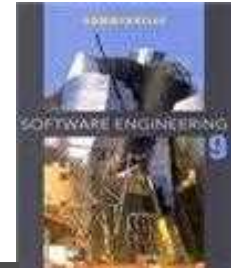


Language processing systems

- ✧ Accept a natural or artificial language as input and generate some other representation of that language.
- ✧ May include an interpreter to act on the instructions in the language that is being processed.
- ✧ Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
 - Meta-case tools process tool descriptions, method rules, etc and generate tools.

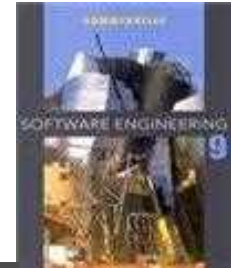
The architecture of a language processing system





Compiler components

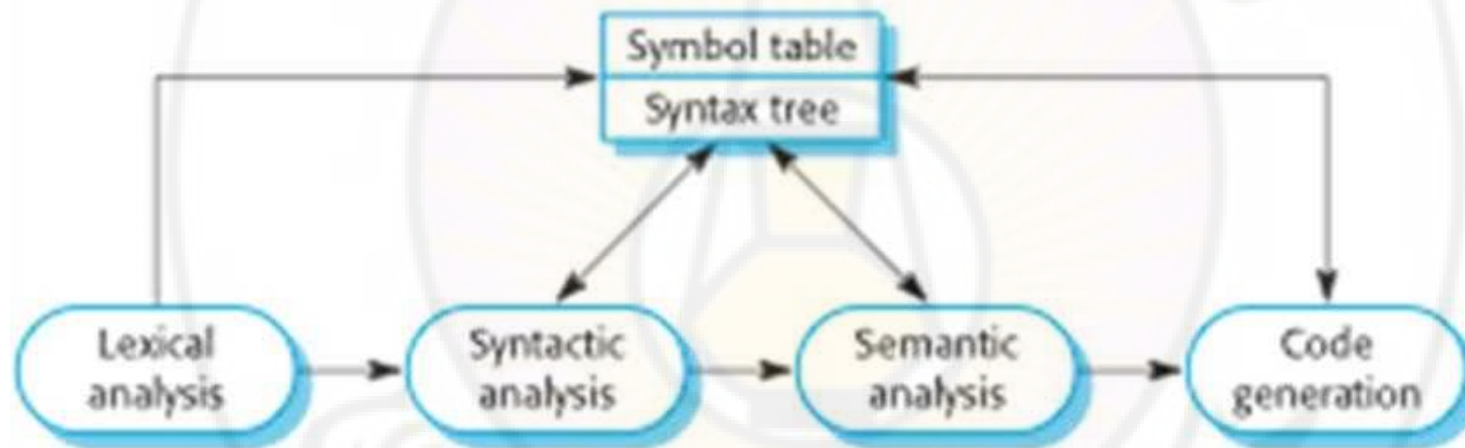
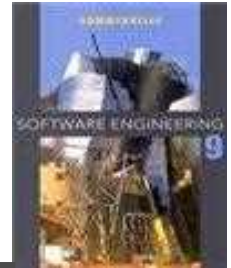
- ✧ A lexical analyzer, which takes input language tokens and converts them to an internal form.
- ✧ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- ✧ A syntax analyzer, which checks the syntax of the language being translated.
- ✧ A syntax tree, which is an internal structure representing the program being compiled.



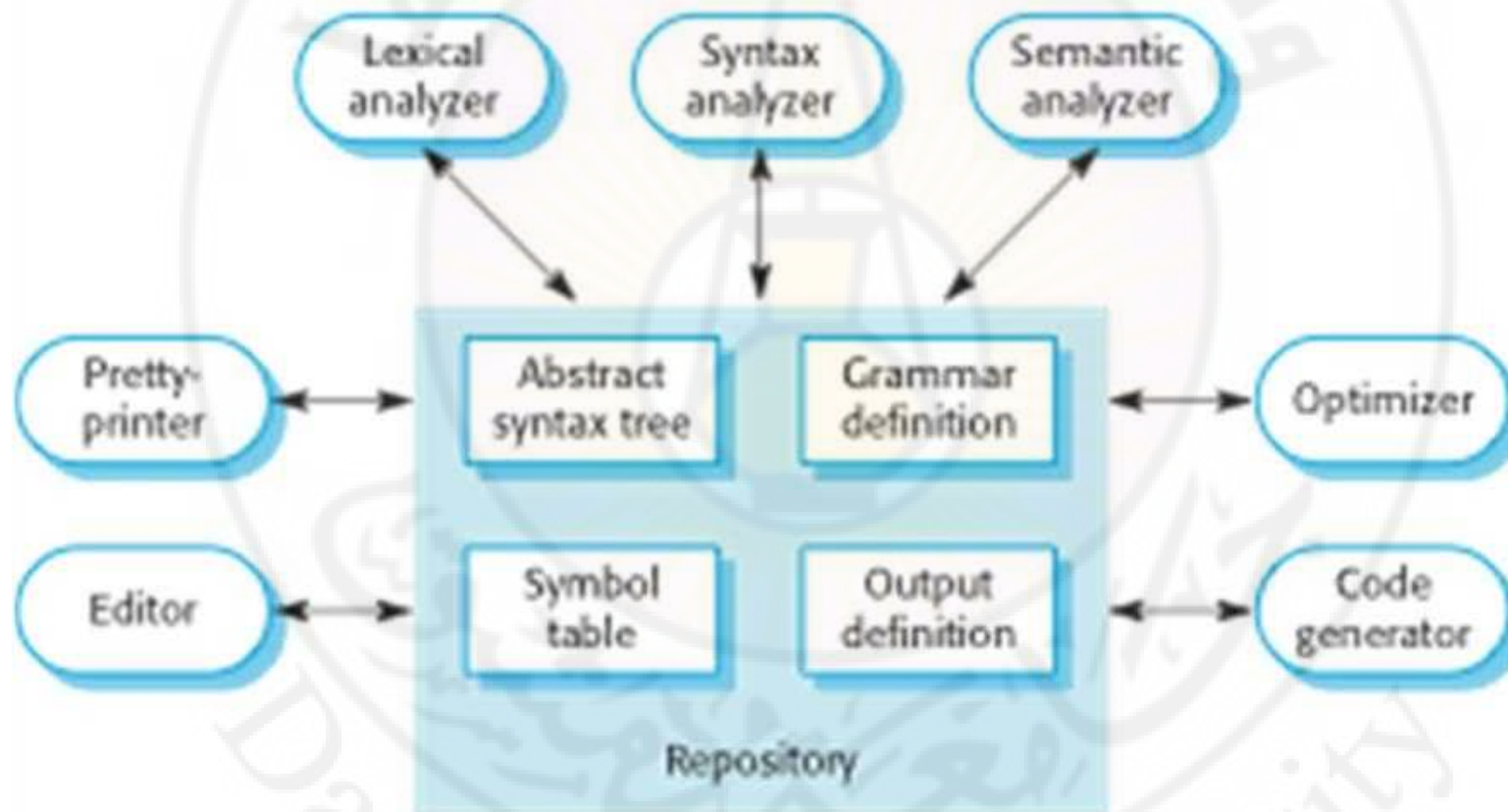
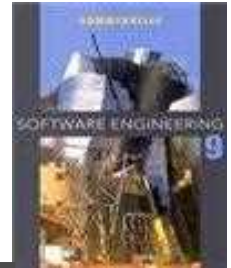
Compiler components

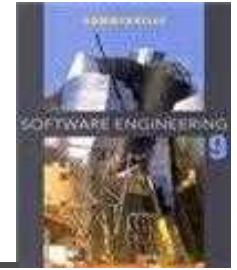
- ✧ A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- ✧ A code generator that 'walks' the syntax tree and generates abstract machine code.

A pipe and filter compiler architecture



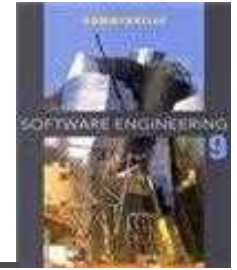
A repository architecture for a language processing system





Key points

- ✧ A software architecture is a description of how a software system is organized.
- ✧ Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- ✧ Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- ✧ Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.



Key points

- ✧ Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- ✧ Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- ✧ Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.