CSCD306-DCIT208: SOFTWARE ENGINEERING

Mark Atta Mensah | Department of Computer Science | 2020-2021

Session 2

Agile Software Development | Requirement Engineering

Reference: Sommerville, Software Engineering, 10 ed., Chapters 3 & 4

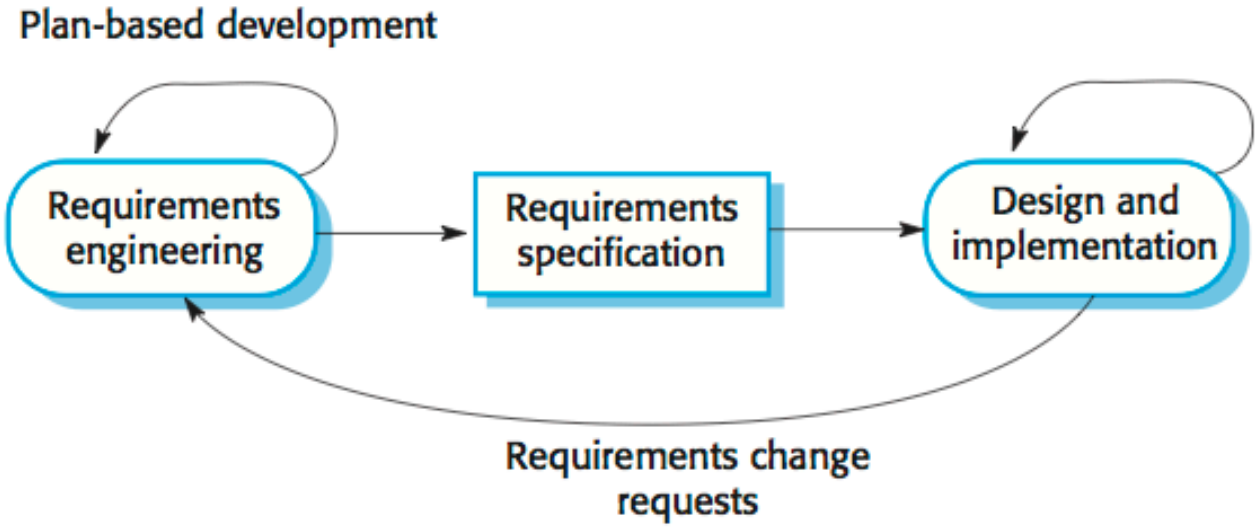
The big Picture

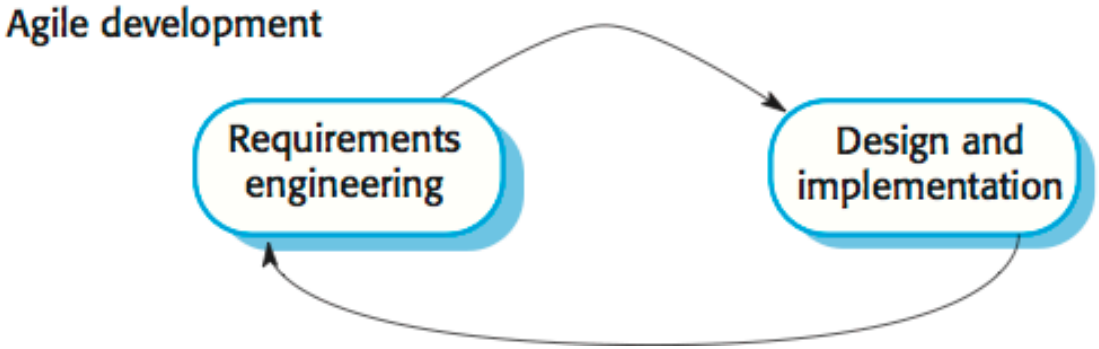
Rapid development and delivery is now often the most important requirement for software systems. Businesses operate in a fast-changing requirement and it is practically impossible to produce a set of stable software requirements. Software has to evolve quickly to reflect changing business needs.

Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems:

- Program specification, design, and implementation are interleaved
- The system is developed as a series of frequent versions or increments
- **Stakeholders** involved in version specification and evaluation
- Extensive **tool support** (e.g. automated testing tools) used to support development
- **Minimal documentation** - focus on working code

Plan-based vs agile development





Plan-based vs 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 inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

Plan-based vs agile development

Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on many technical, human, and organizational issues.

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.

Manifesto for Agile Software Development:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

The Principle of Agile Methods:

- **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.

The Principle of Agile Methods:

- **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.

The Principle of Agile Methods:

- **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.
- 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.

Agile Method: *Problems*

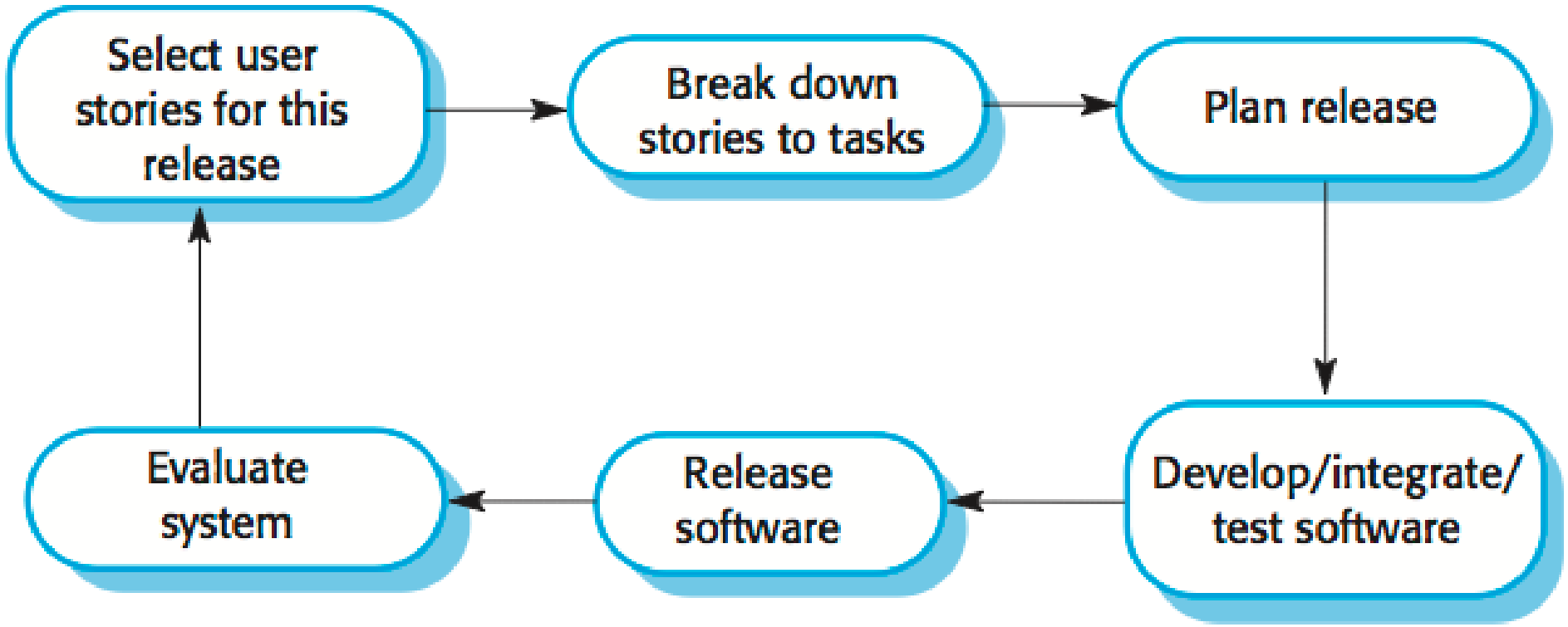
- It can be difficult to keep the interest of **customers** who are involved in the process.
- **Team members** may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are **multiple stakeholders**.
- **Maintaining simplicity**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

Extreme programming

Perhaps the best-known and a very influential 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.

Extreme programming



Extreme programming: *This is how XP supports 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.

Influential XP practices

Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations. Consequently, while agile development uses practices from XP, the method as originally defined is not widely used. Some key practices of XP include:

- **User stories for specification**

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 user stories or scenarios. 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.

Influential XP practices

Refactoring

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. Programming teams 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 require architecture refactoring and this is much more expensive.

Influential XP practices

Test-first development

Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.

- **Test-driven 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.

Influential XP practices

Test-first development

- **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.

Influential XP practices

Pair programming

Pair programming involves programmers working in pairs, developing code together. 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 one person. It encourages refactoring as the whole team can benefit from improving the system code.

Influential XP practices

Pair programming

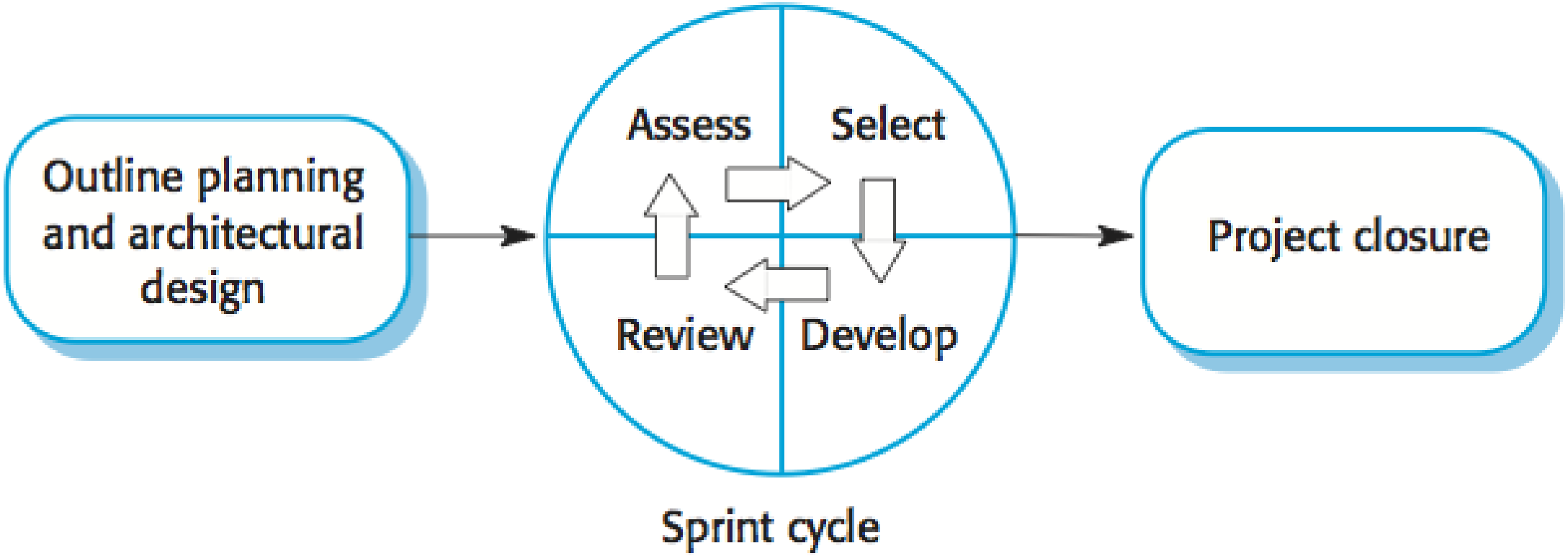
In pair programming, programmers sit together at the same computer to develop the software. Pairs are created dynamically so that all team members work with each other during the development process. The sharing of knowledge that happens during pair programming is very important as it reduces the overall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

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:

1. The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
2. This is followed by a series of ***sprint cycles***, where each cycle develops an increment of the system.
3. 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.

Scrum


Outline planning and architectural design
Assess
Select
Review
Develop
Sprint cycle
Project closure

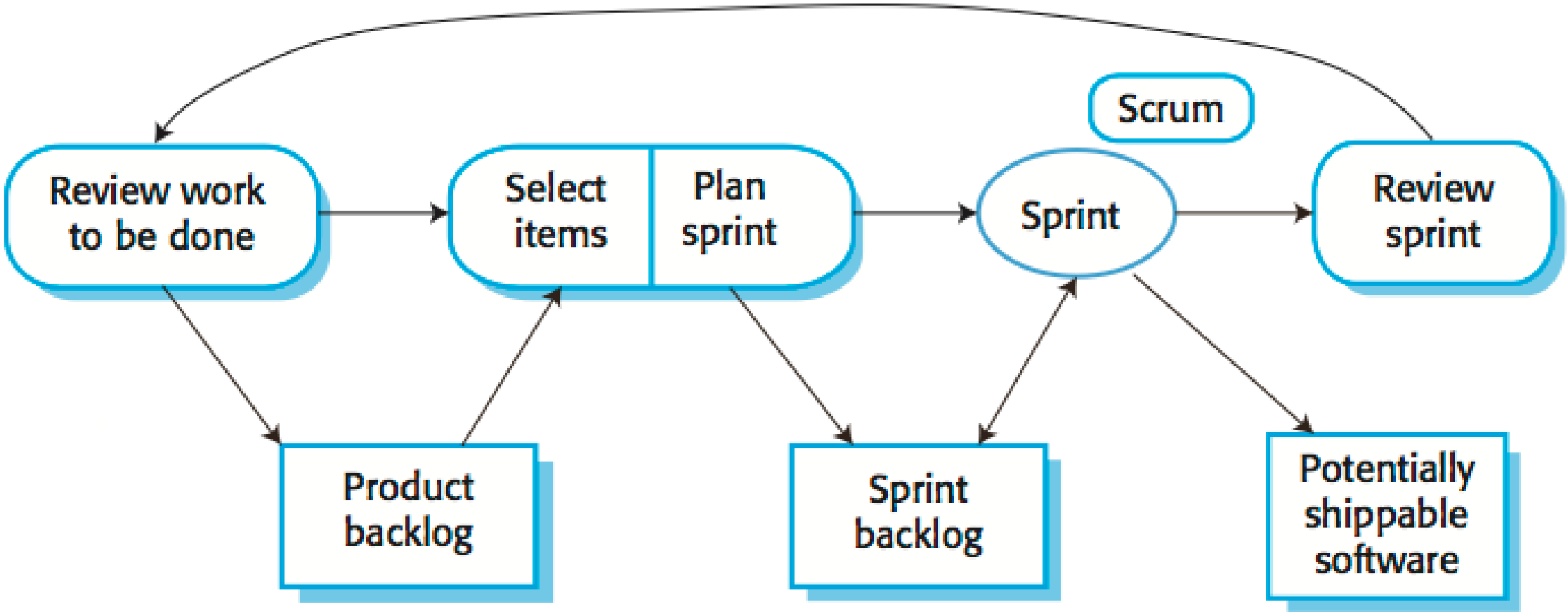
Scrum: *The Sprint cycle*

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 (***product owner***) to select the features and functionality to be developed during the sprint. Once these are agreed, the team organize themselves to develop the software. During this stage the team is relatively isolated from the product owner and the organization, with all communications channeled through the ***ScrumMaster***.

Scrum: *The Sprint cycle*

The role of the ScrumMaster is to protect the development team from external distractions. At the end of the sprint the work done is reviewed and presented to stakeholders (including the product owner). *Velocity* is calculated during the *sprint review*; it provides an estimate of how much product backlog the team can cover in a single sprint. Understanding the team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring and improving performance. The next sprint cycle then begins.

Scrum: *The Scrum Master*



Scrum: *The Scrum Master*

The ScrumMaster is a facilitator who arranges short daily meetings *(daily scrums),* tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with the product owner and management outside of the team. The whole team attends daily scrums 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.

Scrum: *Advantages*

- 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

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.

Scaling agile methods: *Two perspective on scaling agile methods*

'Scaling up'

Using agile methods for developing large software systems that cannot be developed by a small team. 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, which 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 agile methods: *Two perspective on scaling agile methods*

'Scaling out'

How agile methods can be introduced across a large organization with many years of software development experience. 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.

Requirements Engineering: The Big picture

Requirements engineering (RE) is 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. Requirements may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. As much as possible, requirements should describe what the system should do, but not how it should do it.

Requirements Engineering: *Two kinds of requirements based on the intended purpose and target audience*

User requirements

High-level abstract requirements written as statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

System requirements

Detailed description of what the system should do including the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Requirements Engineering: *Classes of Requirement*

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 derived from the domain of operation.

Functional requirements

Functional requirements describe ***functionality*** or system services. They 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.

Functional requirements

Problems arise when requirements are not precisely stated. Ambiguous requirements may be interpreted in different ways by developers and users. In principle, requirements should be both

- **Complete**: they should include descriptions of all facilities required, and
- **Consistent**: there should be no conflicts or contradictions in the descriptions of the system facilities.

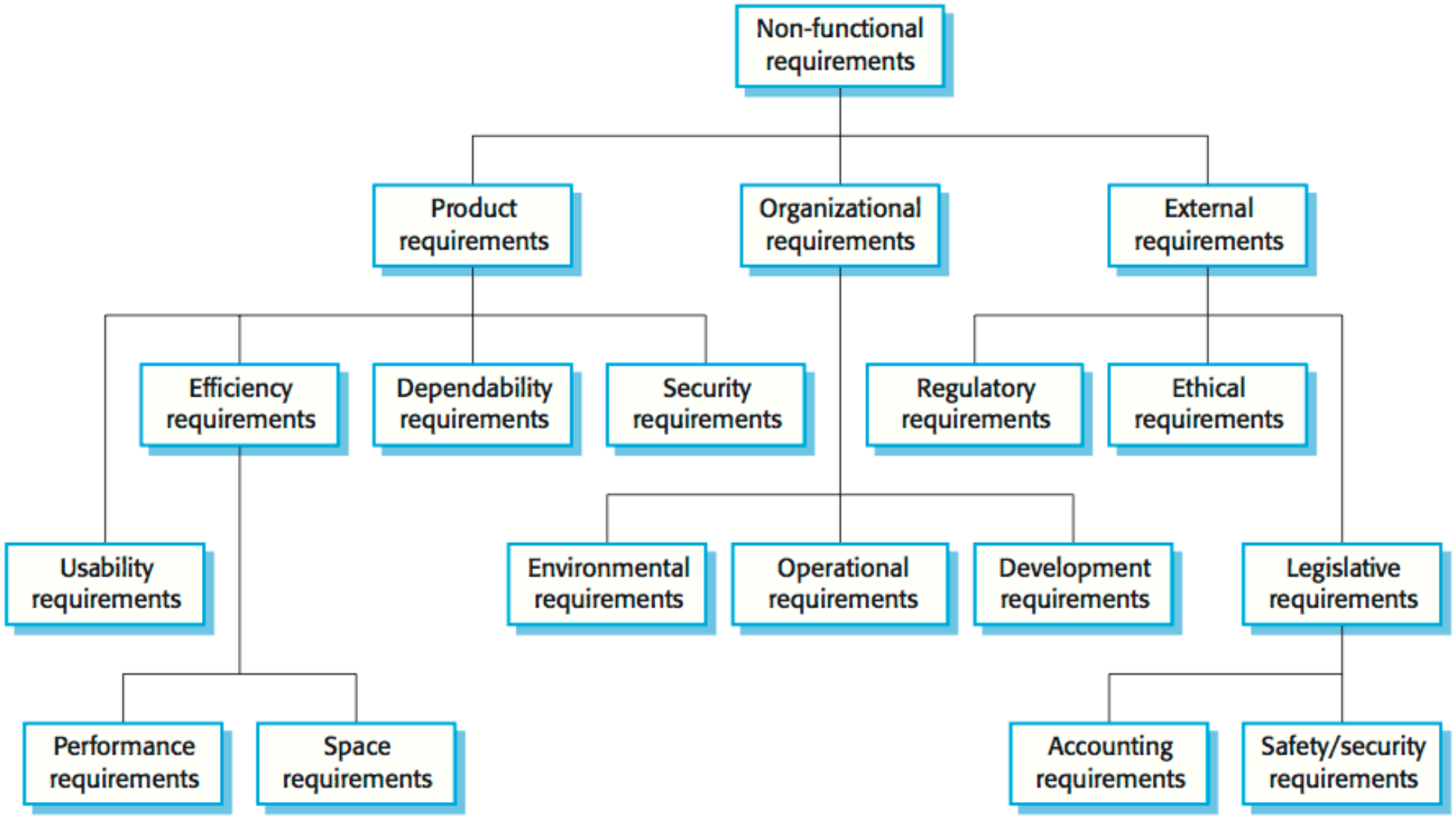
In practice, it is impossible to produce a complete and consistent requirements document.

Non-functional requirements

Non-functional requirements 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.

Non-functional requirements may affect the overall architecture of a system rather than the individual 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 requirements



Non-functional requirements: *Three Classes*

Product requirements

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

Organizational requirements

Requirements which are a consequence of organizational 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.

Non-functional requirements

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify. If they are stated as a **goal** (a general intention of the user such as ease of use), they should be rewritten as a **verifiable** non-functional requirement (a statement using some **quantifiable metric** that can be objectively tested). Goals are helpful to developers as they convey the intentions of the system users.

Domain requirements

The system's operational domain imposes requirements on the system. Domain requirements may be new functional or non-functional requirements, constraints on existing requirements, or define specific computations. If domain requirements are not satisfied, the system may be unworkable. Two main ***problems*** with domain requirements:

- ***Understandability***

  Requirements are expressed in the language of the application domain, which is not always 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.

Requirements engineering process

Processes vary widely depending on the application domain, the people involved and the organization developing the requirements. In practice, requirements engineering is an *iterative process*, in which the following generic activities are interleaved:

- Requirements *elicitation*;
- Requirements *analysis*;
- Requirements *validation*;
- Requirements *management*.

Requirements elicitation and analysis

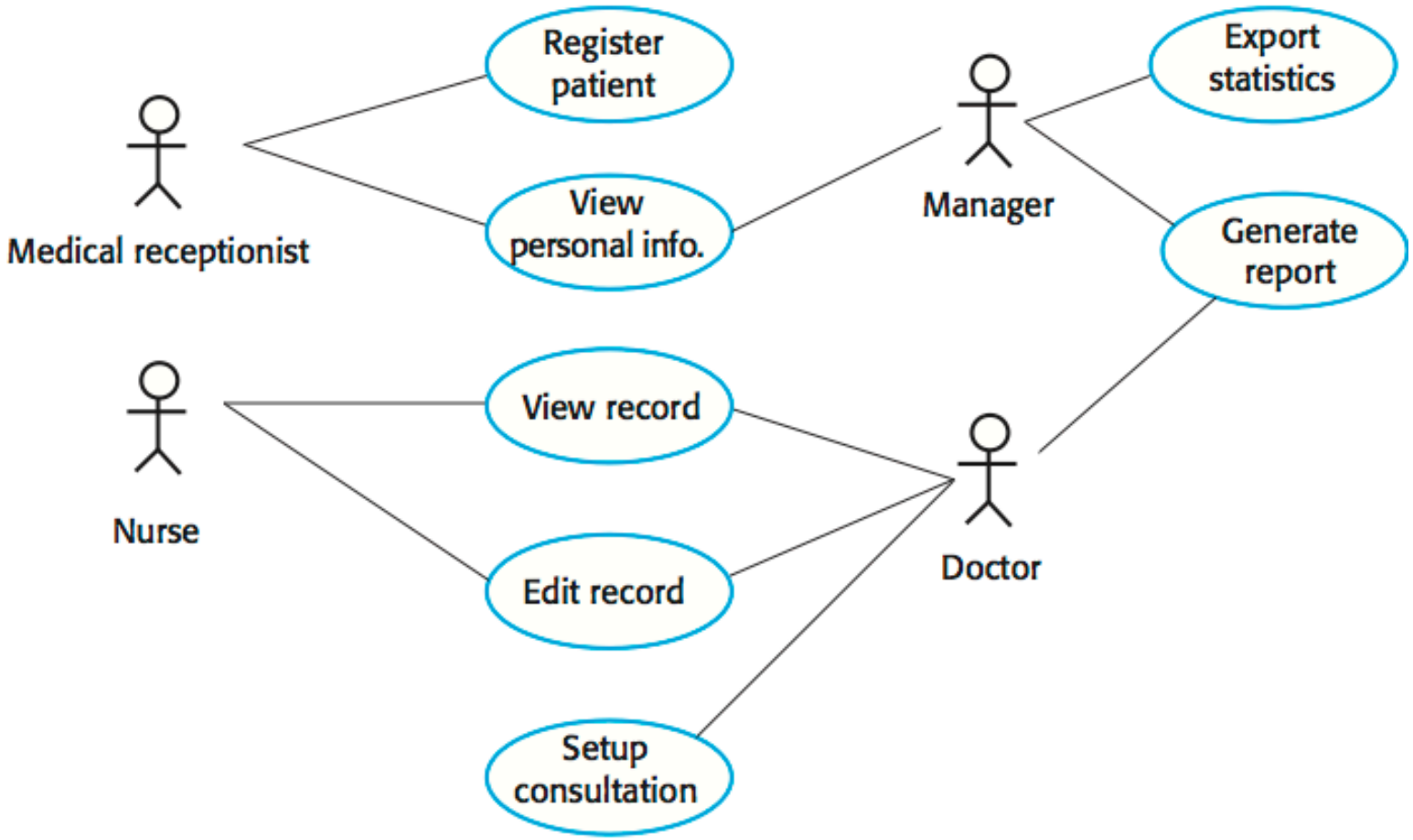
Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.
Stages include:

- ***Requirements discovery***
  Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- ***Requirements classification and organization***
  Groups related requirements and organizes them into coherent clusters.
- ***Prioritization and negotiation***
  Prioritizing requirements and resolving requirements conflicts.
- ***Requirements specification***
  Requirements are documented and input into the next round of the spiral.

Requirements elicitation and analysis

Closed (based on pre-determined list of questions) and open *interviews* with stakeholders are a part of the RE process. *User stories* and *scenarios* are real-life examples of how a system can be used, which are usually easy for stakeholders to understand. Scenarios should include descriptions of the starting situation, normal flow of events, what can go wrong, other concurrent activities, the state of the system when the scenario finishes.

Requirements elicitation and analysis: *Use Cases*



Requirements elicitation and analysis: *Problems*

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organizational 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 specification

Requirements specification is 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 and it is important that these are as complete as possible.

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***.

Requirements specification

User requirements are almost always written in *natural language* supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Natural language is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. This approach maintains most of the expressiveness and understand-ability of natural language but ensures that some uniformity is imposed on the specification.

Requirements validation

Requirements validation is concerned with demonstrating that the requirements define the system that the customer really wants.
Requirements error costs are high so validation is very important.
What *problems* to look for:

- *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: *Requirements validation techniques*

Requirements reviews

Systematic manual analysis of the requirements. Regular reviews should be held while the requirements definition is being formulated. What to look for:

- *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?

Prototyping

Using an executable model of the system to check requirements.

Test-case generation

Developing tests for requirements to check testability.

Requirements change

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. Reasons why requirements change after the system's deployment:

- The business and technical environment of the system always changes after installation.
- The people who pay for a system and the users of that system are rarely the same people.
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

End of Session