SWEN 6301 Software Construction

Module 3: Agile Development and Requirements Engineering

Ahmed Tamrawi

What does the program print?

```
1  public class JavaPuzzle {
2
3      public static void main(String[] args) {
4          print("Hello");
5
6          /*
7           * TODO: print World in unicode
8           * \u002A\u002F\u0070\u0072\u0069\u006E\u0074\u0028\u0022\u0043\u0072
9           * \u0075\u0065\u006C\u0022\u0029\u003B\u002F\u002A
10          */
11         print("World");
12     }
13
14     private static void print(String s){
15         System.out.print(s + " ");
16     }
17
18 }
```

Agile Software Development

Topics Covered

- Agile methods
- Agile development techniques
- Agile project management
- Scaling agile methods

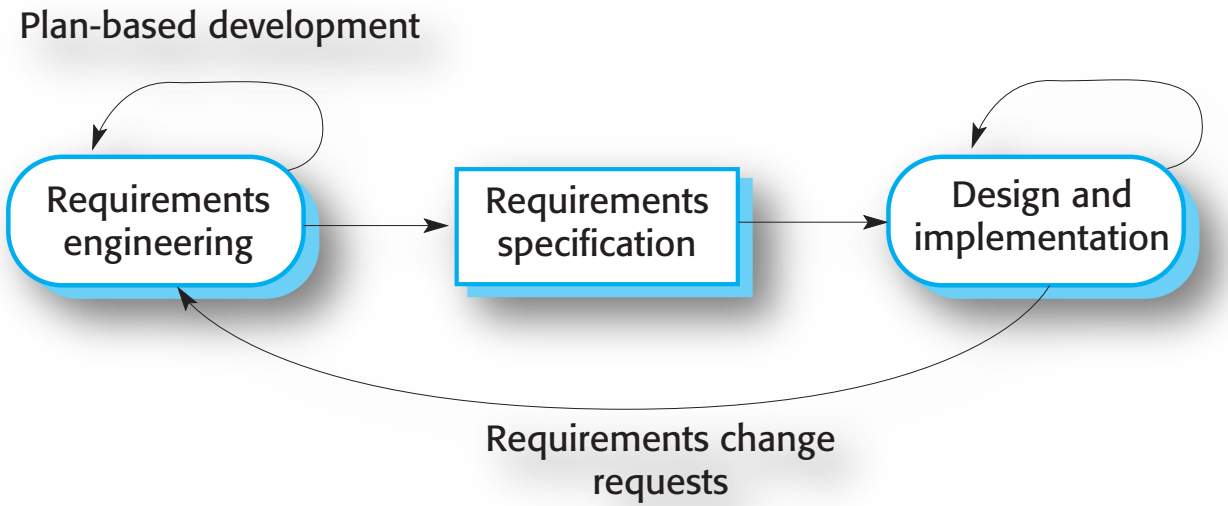
Rapid Software Development

- *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 must **evolve quickly** to reflect changing business needs.
- Plan-driven development is essential for some types of system but does **not** meet these business needs.
- Agile development methods emerged in the late 1990s whose aim was to **radically reduce the delivery time** for working software systems.

Agile Development

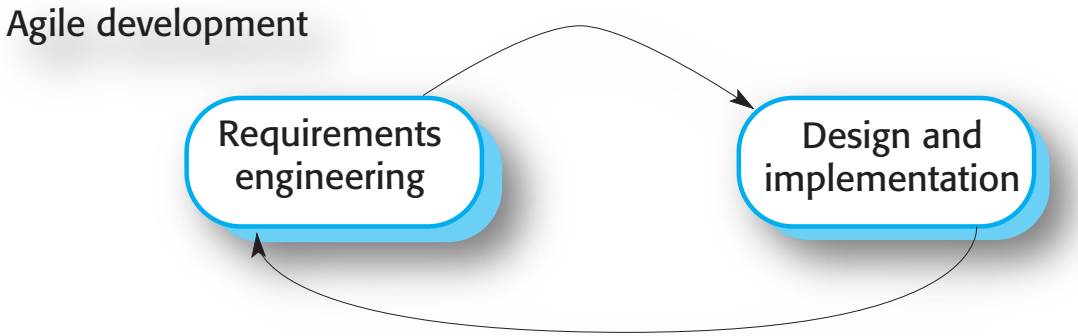
- Program specification, design and implementation are **inter-leaved**.
- The system is developed as a *series of versions or increments* with stakeholders involved in version specification and evaluation.
- **Frequent** delivery of new versions for evaluation.
- Extensive **tool support** (e.g. automated testing tools) used to support development.
- **Minimal documentation** – focus on working code.

Plan-Driven and Agile Development



Plan-Driven Development

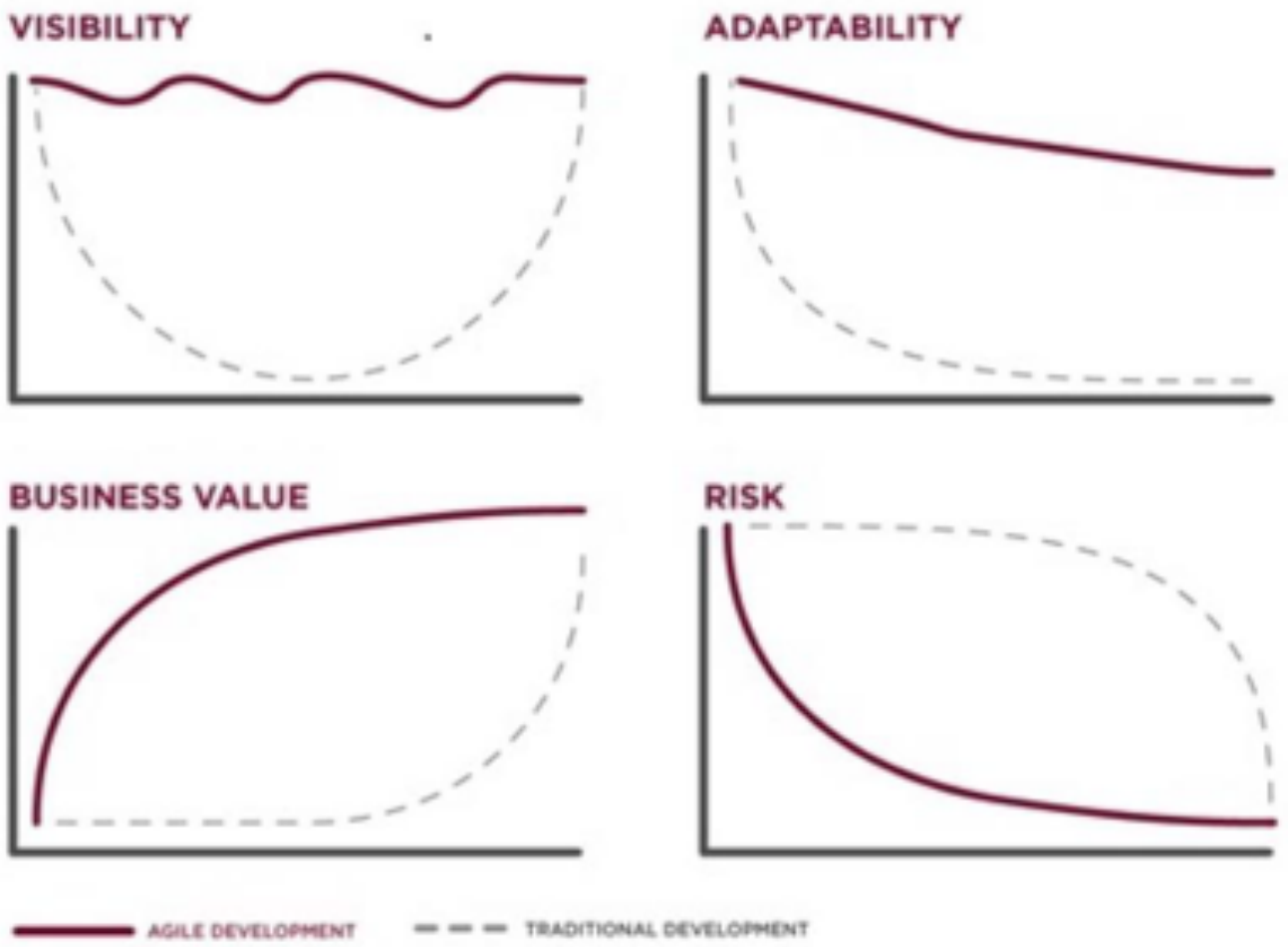
- A plan-driven approach to software engineering is based around **separate** development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible.
- Iteration occurs within activities.



Agile Development

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

Plan-Driven vs Agile Development



Agile Methods

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.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

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**.

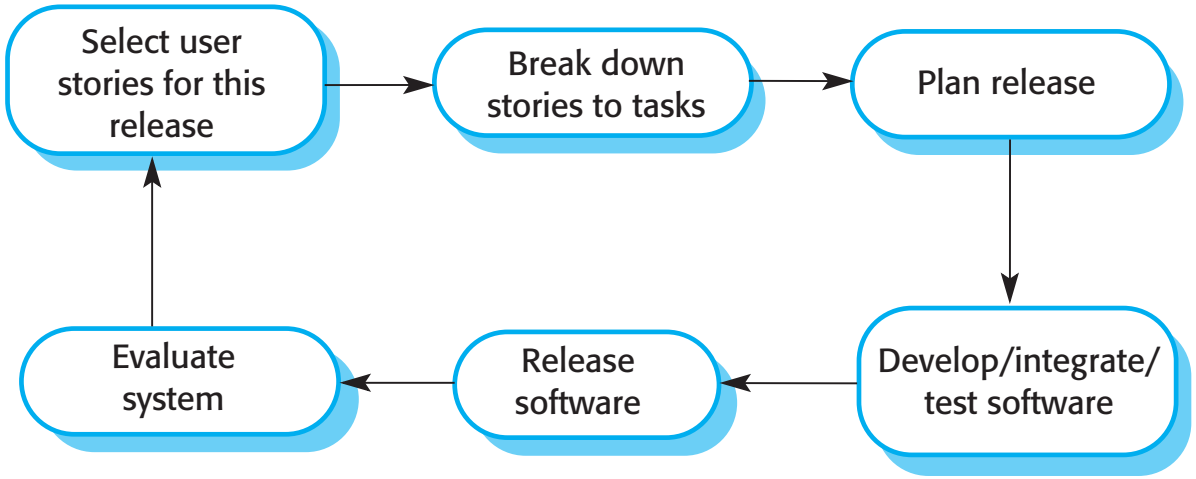
Agile Method Applicability

- **Product development** where a software company is developing a small or medium-sized product for sale.
  - Virtually all software products and apps are now developed using an agile approach
- **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 few external rules and regulations that affect the software.

Agile Development Techniques

Extreme Programming

- A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- 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 Practices

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'**.
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 re-factor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme Programming Practices

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.

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**.

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**.

User Stories for Requirements

- 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.

A ‘prescribing medication’ Story and Tasks

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either ‘current medication’, ‘new medication’ or ‘formulary’.

If you select ‘current medication’, you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click ‘OK’ or ‘Change’. If you click ‘OK’, your prescription will be recorded on the audit database. If you click ‘Change’, you reenter the ‘Prescribing medication’ process.

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low.
If within the range, enable the ‘Confirm’ button.

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.

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.

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.
- 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-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.

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 Case Description for Dose Checking

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

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.

Problems with Test-First Development

- 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

- 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 1 person.
- It encourages refactoring as the whole team can benefit from improving the system code.
- In pair programming, programmers sit together at the same computer to develop the software.

Pair Programming

- 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 2 programmers working separately.

Agile Project Management

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 practices used in agile methods.

Scrum

- Scrum is an *agile method that focuses 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.

SCRUM

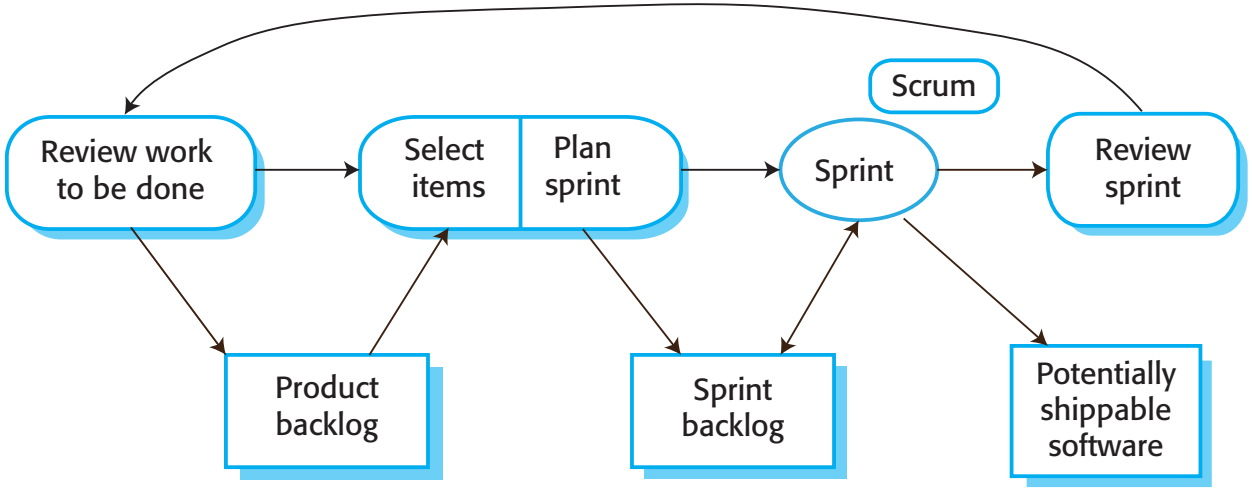
A framework for managing work with an emphasis on software development. It is designed for teams of developers (3 to 9) who break their work into actions that can be completed within timeboxed iterations, called sprints (30 days or less, most commonly two weeks) and track progress and re-plan in 15-minute stand-up meetings, called daily scrums.

Scrum Terminology

Scrum term	Definition
Development team	A self-organizing group of software developers responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be '**potentially shippable**' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

The Scrum Sprint Cycle

- Sprints are fixed length, normally 2–4 weeks.
- 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 from the product backlog 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

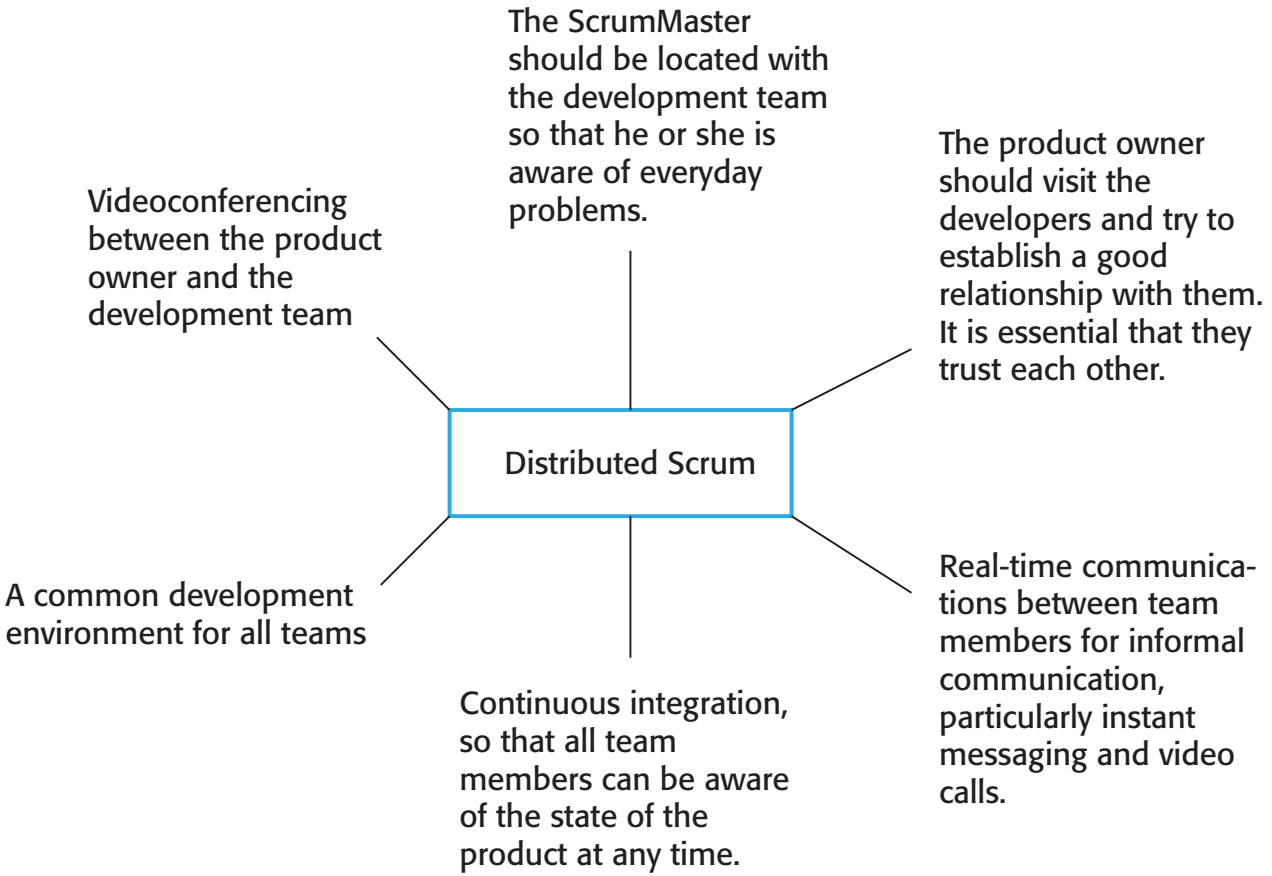
- The ‘Scrum master’ is a **facilitator** who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- The whole team attends short daily meetings (Scrums) where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
  - 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.

Distributed Scrum

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams, with team members located in different places around the world.



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 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 important to maintain agile fundamentals:
  - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

Practical Problems with Agile Methods

- The informality of agile development is incompatible with the **legal approach to contract definition** that is commonly used in large companies.
- Agile methods are **most appropriate for new software development rather than software maintenance**. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- Agile methods are designed for **small co-located teams** yet much software development now involves worldwide distributed teams.

Contractual Issues

- Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- However, this precludes interleaving specification and development as is the norm in agile development.
- A contract that pays for developer time rather than functionality is required.
  - However, this is seen as a high risk in many legal departments because what has to be delivered cannot be guaranteed.

Agile Methods and Software Maintenance

- Key problems are:
  - Lack of product documentation
  - Keeping customers involved in the development process
  - Maintaining the continuity of the development team
- Agile development relies on the development team knowing and understanding what has to be done.
- For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

Agile and Plan-Driven Methods

- 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.

Key Points

- Agile methods are incremental development methods that focus on rapid software development, frequent releases of the software, reducing process overheads by minimizing documentation and producing high-quality code.
- Agile development practices include:
  - User stories for system specification
  - Frequent releases of the software,
  - Continuous software improvement
  - Test-first development
  - Customer participation in the development team.
- Scrum is an agile method that provides a project management framework.
  - It is centered round a set of sprints, which are fixed time periods when a system increment is developed.

Key Points

- Many practical development methods are a mixture of plan-based and agile development.
- Scaling agile methods for large systems is difficult.
  - Large systems need up-front design and some documentation and organizational practice may conflict with the informality of agile approaches.

Requirements Engineering

Topics Covered

- Functional and non-functional requirements
- Requirements engineering processes
- Requirements elicitation
- Requirements specification
- Requirements validation
- Requirements change

Requirements Engineering

- *The process of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.*
- The system requirements 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.***

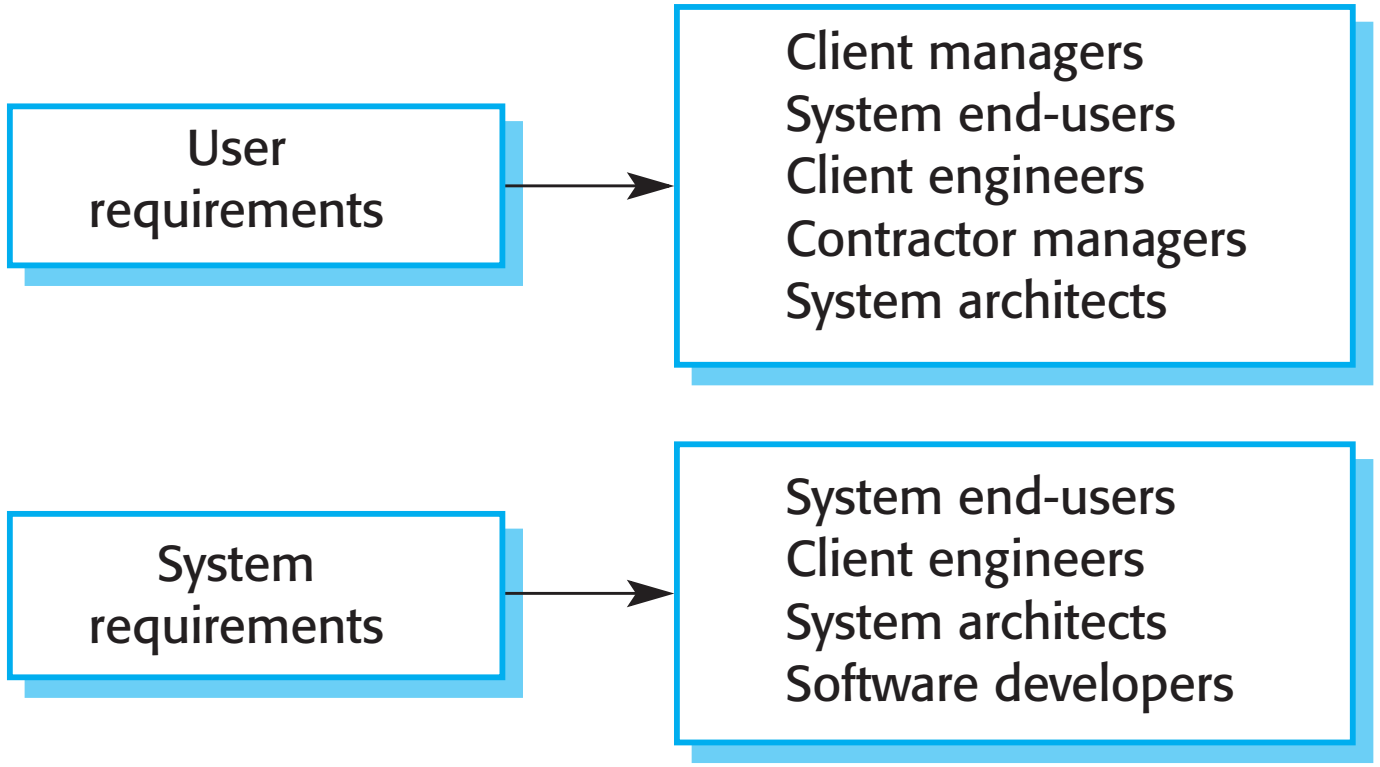
User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Readers of Different Types of Requirements Specification



System Stakeholders

- *Any person or organization who is affected by the system in some way and so who has a legitimate interest*
- Stakeholder types:
  - End users
  - System managers
  - System owners
  - External stakeholders

Stakeholders in the Mentcare System

- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.
- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Health care managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Agile Methods and Requirements

- Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly.
- The requirements document is therefore **always out of date**.
- Agile methods usually use incremental requirements engineering and may express requirements as '**user stories**'.
- This is practical for business systems but problematic for systems that require pre-delivery analysis (e.g. critical systems) or systems developed by several teams.

Functional and Non-functional Requirements

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.

Mentcare System: *Functional Requirements*

- A user shall be able to **search** the appointments lists for all clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements Imprecision

- Problems arise when functional requirements are *not precisely stated*.
- **Ambiguous** requirements may be interpreted in different ways by developers and users.
- Consider the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

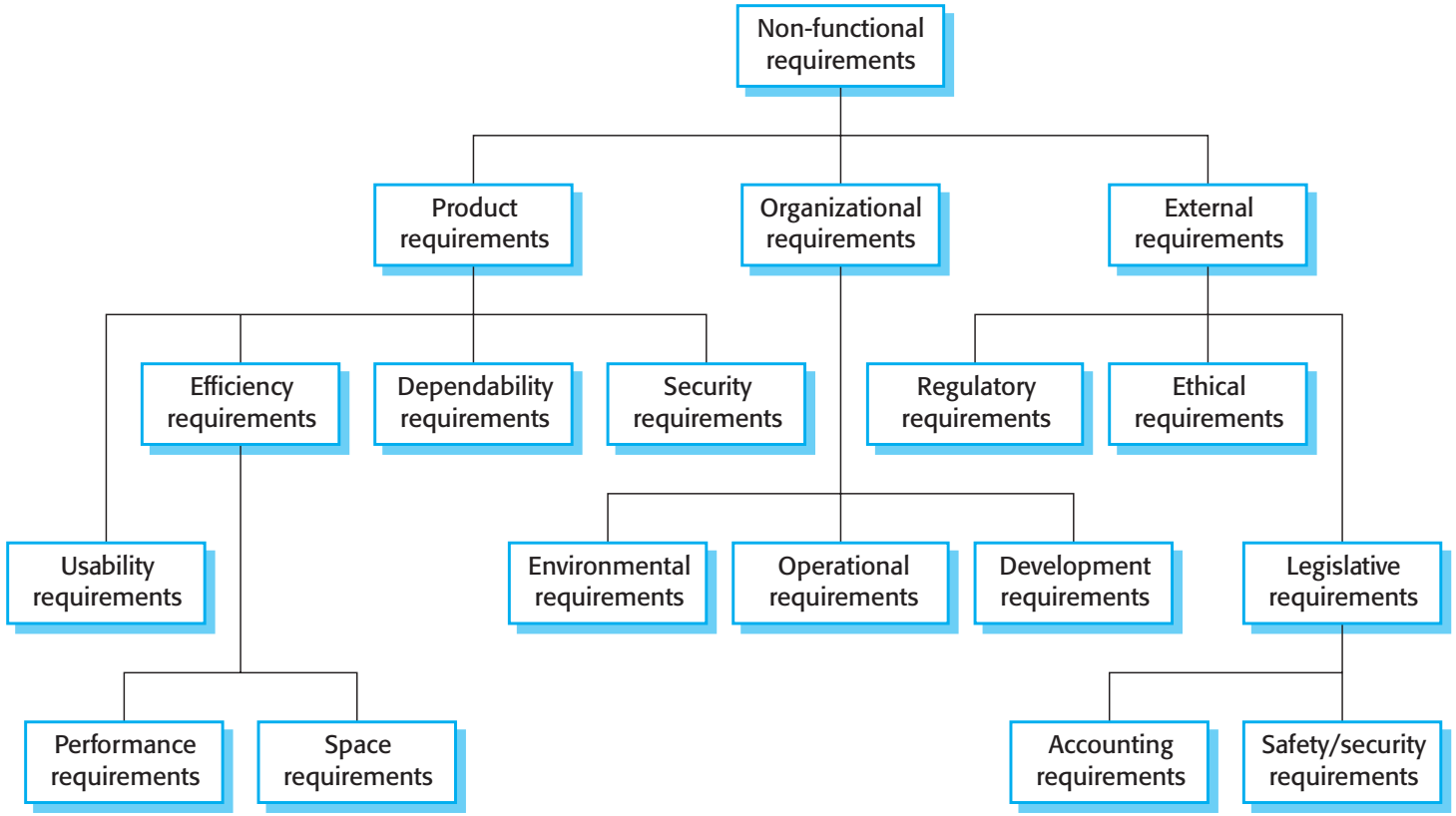
Requirements Completeness and Consistency

- 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, because of system and environmental complexity, 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 Non-functional 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.

Examples of Non-functional Requirements in the Mentcare System

Product requirement
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement
Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

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.

Usability Requirements

- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement)

Metrics for Specifying Non-functional Requirements

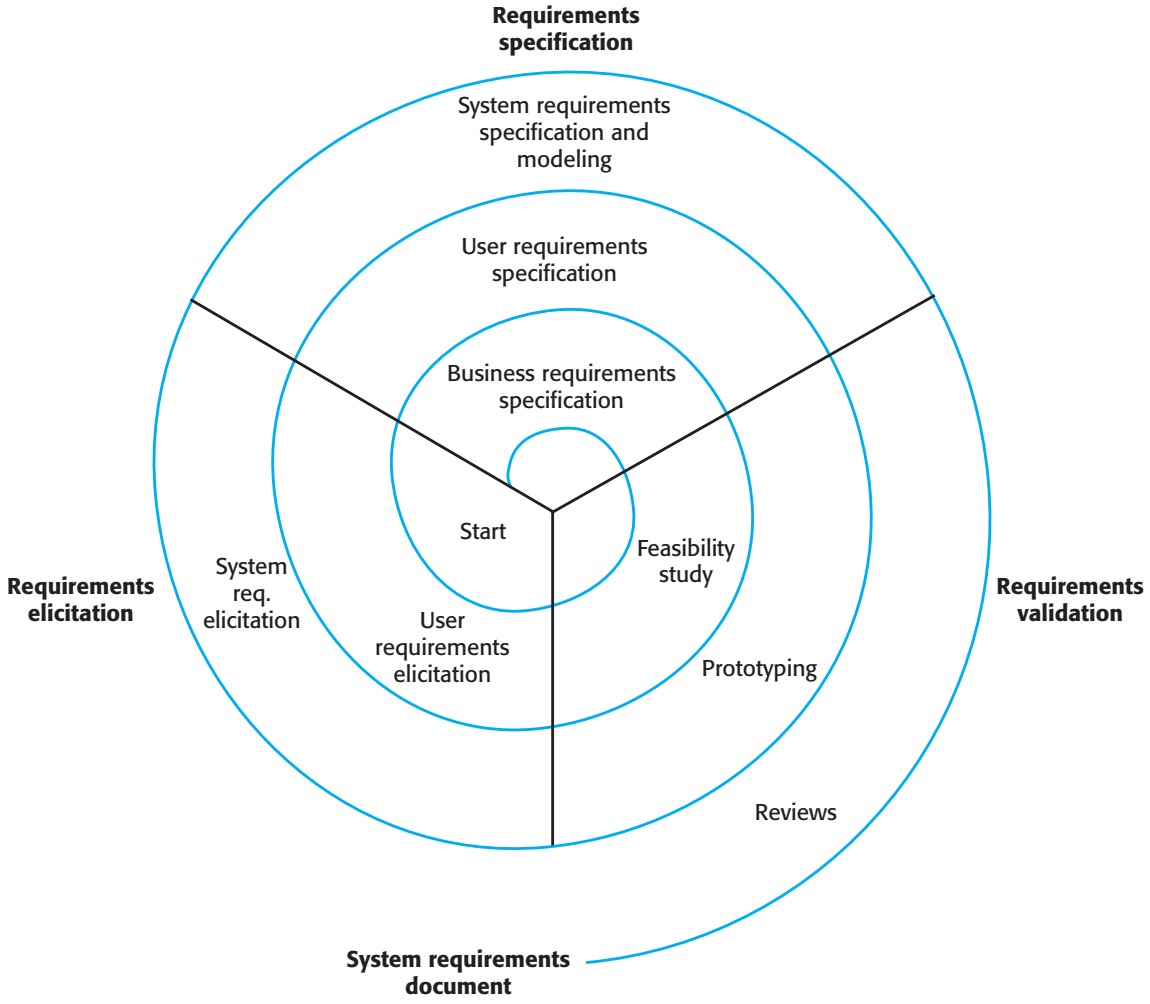
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart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

Requirements Engineering Processes

Requirements Engineering Processes

- The processes used for RE **vary widely** depending on the **application domain**, the **people** involved and the **organization** 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.

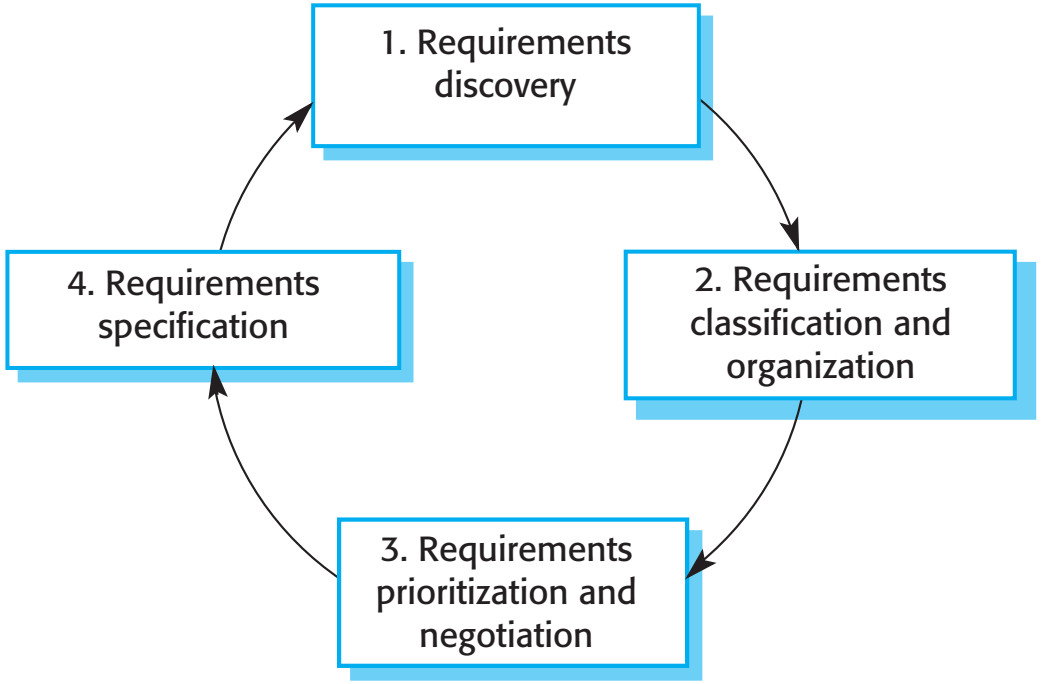
Spiral View of Requirements Engineering Process



Requirements Elicitation

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.*



Process Activities

- **Requirements discovery**
  - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage. (**Interviewing and Ethnography**)
- **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 may change.

Requirements Specification

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.

Writing 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

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 Specification of a Requirement for an Insulin Pump

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

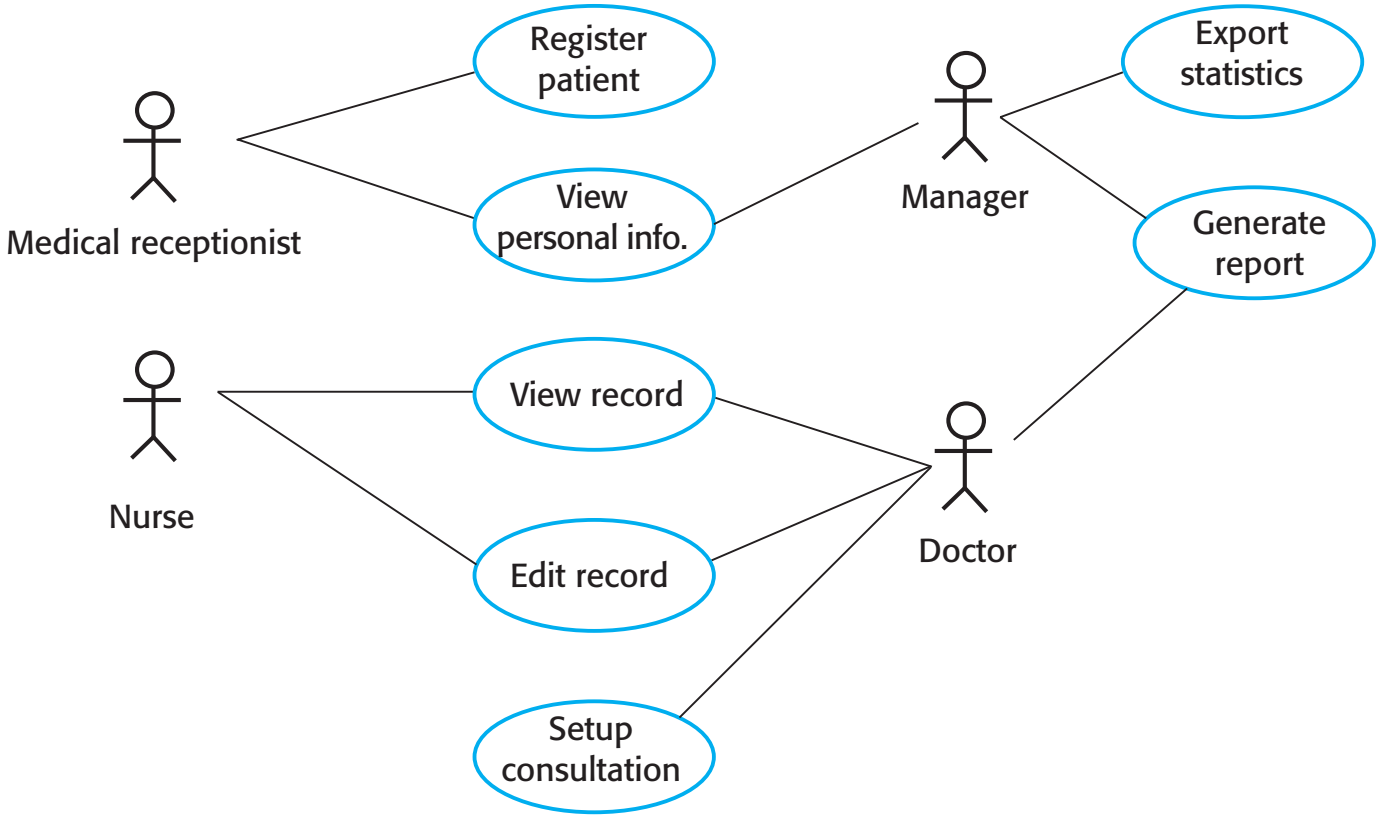
Tabular Specification of Computation for an Insulin Pump

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) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Use Cases

- Use-cases are a kind of scenario that are included in the UML.
- Use cases 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.
- **UML sequence diagrams** may be used to add detail to use-cases by showing the sequence of event processing in the system.

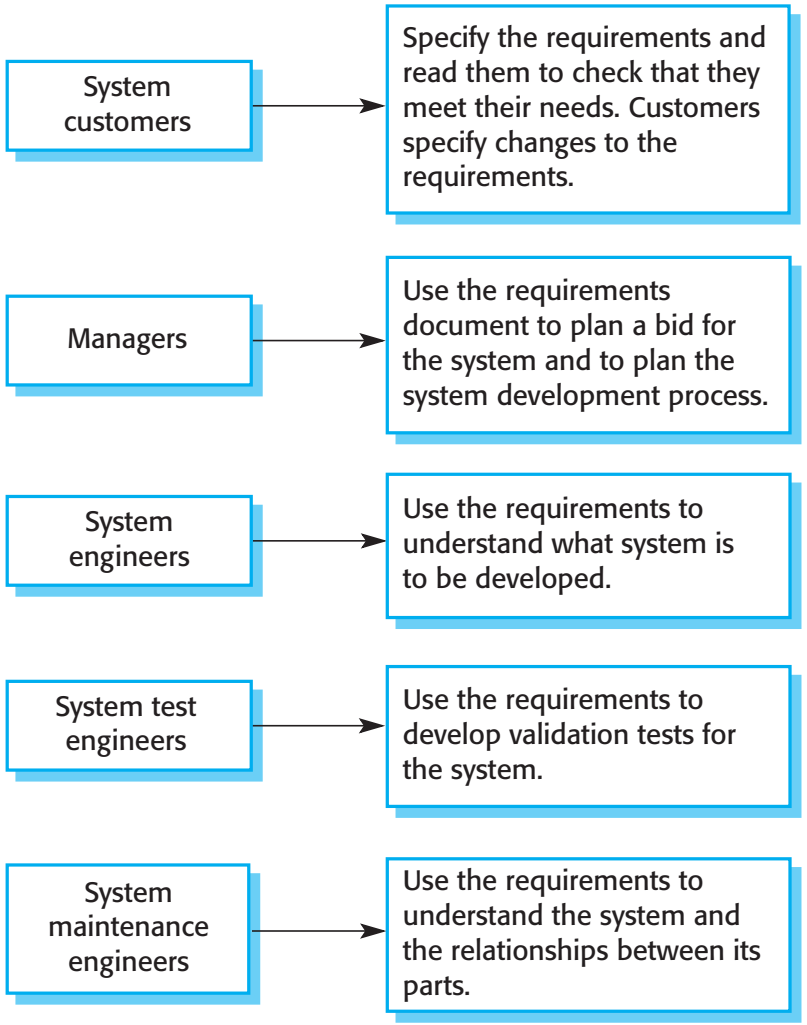
Use Cases for the Mentcare System



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.

Users of a Requirements Document



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.

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.

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 Validation

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.*
  - **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

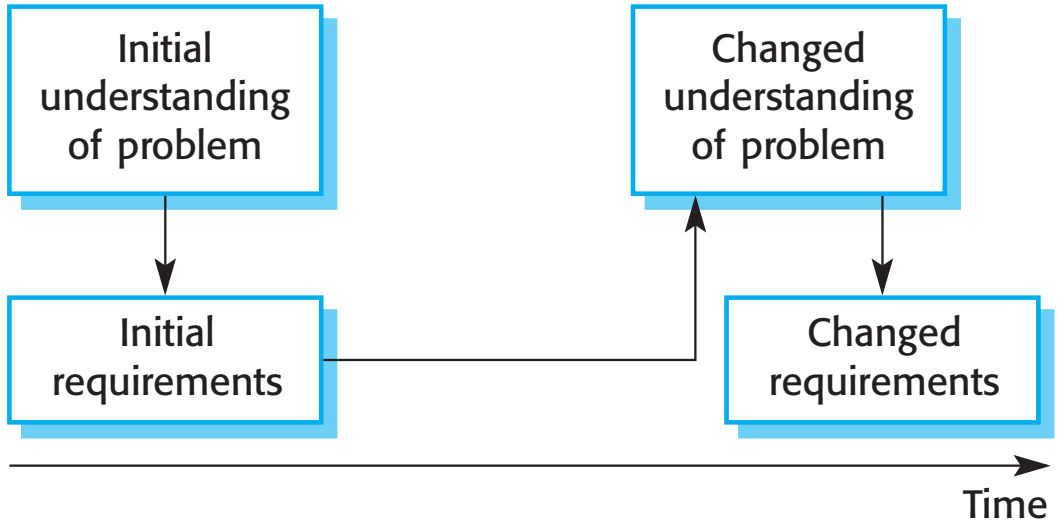
Changing Requirements

- 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

- 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**.

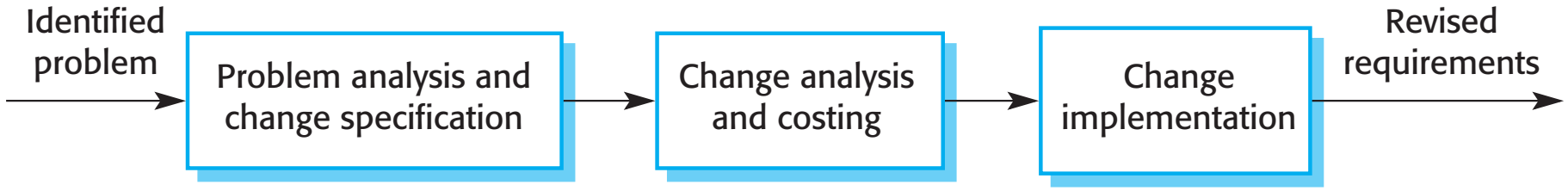
Requirements Management Planning

- 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.
  - *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



Key Points

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.

Key Points

- Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
- You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.
- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

Key Points

- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
- Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.