SOFTWARE PROCESSES

1. Software specification The functionality of the software and constraints on its operation must be defined.
2. Software design and implementation The software to meet the specification must be produced.
3. Software validation The software must be validated to ensure that it does what the customer wants.
4. Software evolution The software must evolve to meet changing customer needs.

–

they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There are also supporting process activities such as documentation and software configuration management.

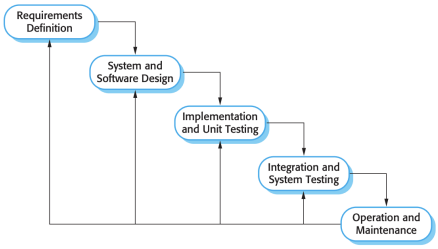
–

Software Process may also include:

1. Products, which are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles, which reflect the responsibilities of the people involved in the process.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed.

The process models:

1. The waterfall model: This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.



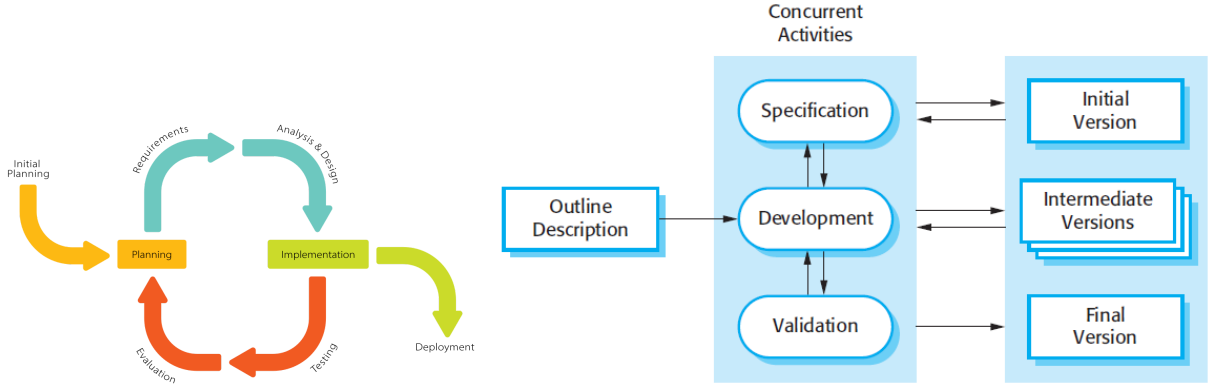
The principal stages of the waterfall model directly reflect the fundamental development activities:

1. Requirements analysis and definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification. They are particularly suited to the development of systems that have stringent safety, reliability, or security requirements. Processes based on formal transformations are generally only used in the development of safety-critical or security-critical systems. They require specialized expertise.
2. Incremental development: This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version. The idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed. (A fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems). Cheaper and easier. Customer feedback rapidly.
User interface should always be developed using an incremental approach.
Problems: -Process is not visible(no documentation)
-System structure tends to degrade as new increments are added.

Best approach may be mix of plan-driven, agile, and incremental. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified but the development of later increments depends on progress and customer priorities.



3. Reuse-oriented software engineering This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

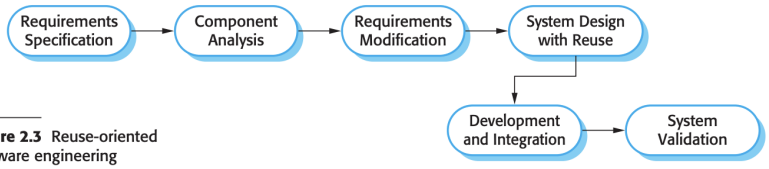
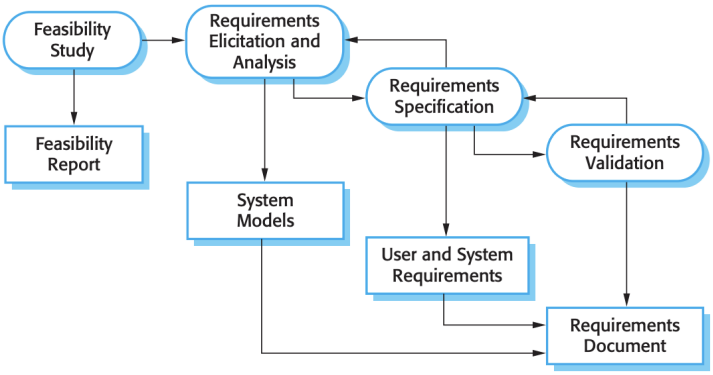


Figure 2.3 Reuse-oriented software engineering

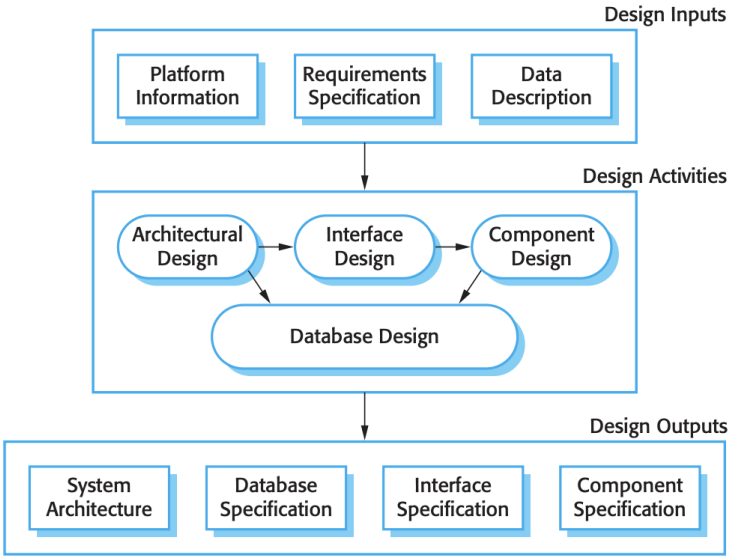
The new project is same/similar to old ones, so the worker modifies them.
1. Component analysis: Searching for similar projects
2. Requirements modification: Requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
3. System design with reuse: The framework of the system is designed and organized for the reused project.
4. Development and integration: Software that cannot be externally procured is developed, and the components and COTS(commercial off the shelf(kullanıma hazır)) systems are integrated to create the new system.
1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.
Obvious advantage of reducing cost risk, and time.
—-------

Software specification or requirements engineering:
There are four main activities in the requirements engineering process:
1. Feasibility study: Can the project be done by using current software and hardware technologies, is the existing budget enough, will profit be made
2. Requirements elicitation and analysis: Deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. Maybe also models and prototypes.
3. Requirements specification:The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. 1- User requirements are abstract statements of the system requirements for the customer and end-user of the system; 2-System requirements are a more detailed description of the functionality to be provided.
4. Requirements validation: Requirements for realism, consistency, and completeness. During this process, errors in the requirements document are fixed.



Software design and implementation:

The implementation stage of software development is the process of converting a system specification into an executable system.



Parts of design process for information systems:

1. 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.
2. Interface design, where you define the interfaces between system components. With a precise interface, a component can be used without other components having to know how it is implemented.
3. Component design, where you take each system component and design how it will operate.
4. Database design, where you design the system data structures and how these are to be represented in a database.

–

The development of a program to implement the system follows naturally from the system design processes. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for the later stages of design and program development to be interleaved.

stages of testing

Testing establishes the existence of defects. **Debugging** is concerned with locating and correcting these defects

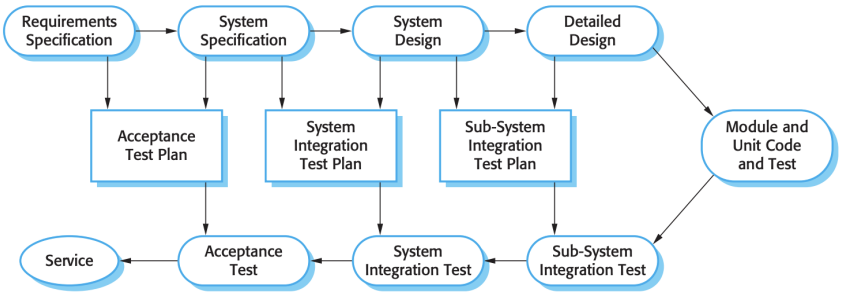
Software validation/verification and validation (V&V)

The stages in the testing process are:

1. Development testing: The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components.
2. System testing: System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements.
3. Acceptance testing: This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. //customer and user needs are tested

(Acceptance testing is sometimes called ‘alpha testing’. Custom systems are developed for a single client. Testing continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.)

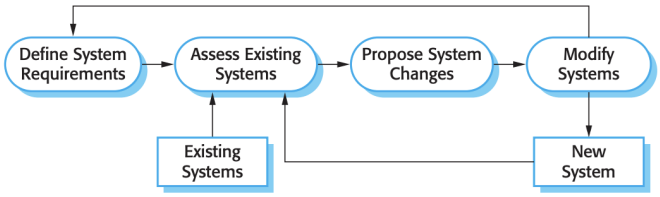
When a system is to be marketed as a software product, a testing process called ‘beta testing’ is often used. Beta users give valuable feedback to developers.



Software evolution:development and maintanance are together not seperate.

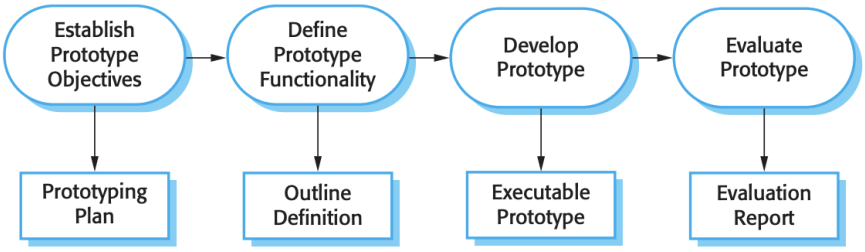
Coping with change:

Change is inevitable.



Cost increases because already finished work has to be repeated.
To reduce this cost:
1. 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. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. 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.

Two ways of coping with change and changing system requirements:
1. System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements.
2. Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.



Prototyping
1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.
Prototype may bring new ideas or reveal error also good for the customer needs, can be reviewed.
Not everything is included in the prototype just the basics.
The prototype may not be the same as the final project.

Developers are sometimes pressured by managers to deliver throwaway prototypes, particularly when there are delays in delivering the final version of the software. However, this is usually unwise.
Sometimes a mockup is a prototype

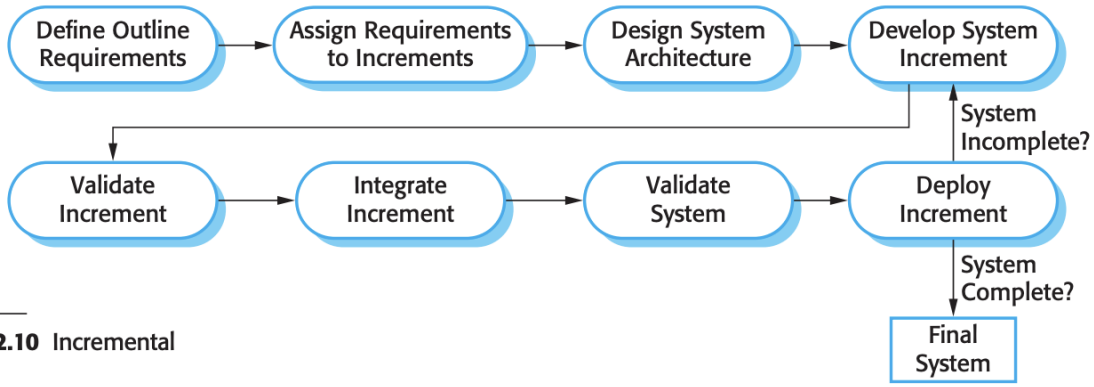


Figure 2.10 Incremental delivery

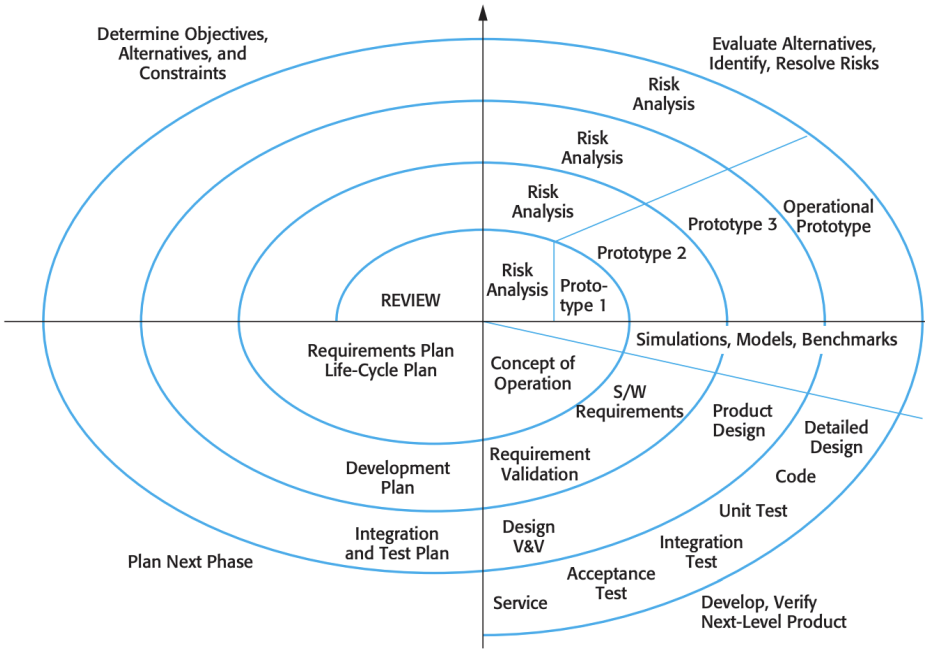
Incremental delivery
Highest prioritized services are implemented first, and so on.
As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.

advantages:
1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system so there is no re-learning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.
4. As the highest-priority services are delivered first and increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery:
1. 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.
2. Iterative development can also be difficult when a replacement system is being developed. Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting useful customer feedback is difficult.

3. 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. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.
Very large systems are not appropriate for incremental developing.

Boehm's spiral model



Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another.
The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.
Each loop in the spiral is split into four sectors: ***????(spiral model olmadı)***
1. Objective setting: Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.
2. Risk assessment and reduction: For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.
3. Development and validation: A development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use.

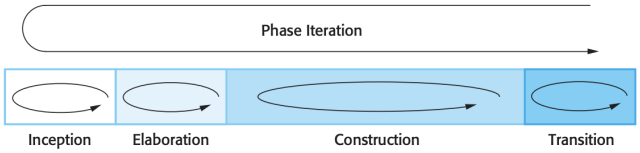
4. Planning: If the decision is to continue, plans are drawn up for the next phase of the project.

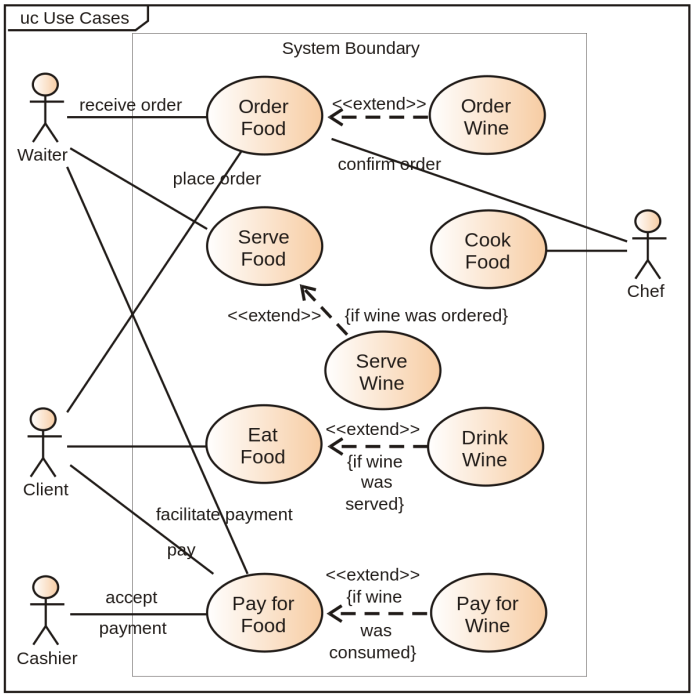
The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives, and dealing with the constraints on each of them, are then enumerated. Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation. Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity.

The Rational Unified Process

It brings together elements from all of the generic process models, illustrates good practice in specification and design and supports prototyping and incremental delivery.
RUP has 3 perspectives:
1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Phases of RUP:
1. Inception(başlangıç): To establish a business case for the system. You should identify all external entities (people and systems) that will interact with the system and define these interactions. You find out if the contribution of the project is profitable or not.
2. Elaboration: The elaboration phase aim to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. You should have a requirements model for the system, which may be a set of UML use cases, an architectural description, and a development plan for the software.
3. Construction: The construction phase involves system design, programming, and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.
4. Transition: The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity. On completion of this phase, you should have a documented software system that is working correctly in its operational environment.



UML USE-CASES: 

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception.

Six fundamental best practices on RUP:
1. Develop software iteratively Plan increments of the system based on customer priorities and develop the highest-priority system features early in the development process.
2. Manage requirements Explicitly document the customer's requirements and keep track of changes to these requirements.
3. Use component-based architectures Structure the system architecture into components.
4. Visually model software Use graphical UML models.
5. Verify software quality Ensure that the software meets the organizational quality standards.
6. Control changes to software Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP is not a suitable process for all types of development, e.g., embedded software development.
The most important innovations in the RUP are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

—-------

E X E R C I S E S 2.1. Giving reasons for your answer based on the type of system being developed, suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems: A system to control anti-lock braking in a car A virtual reality system to support software maintenance A university accounting system that replaces an existing system An interactive travel planning system that helps users plan journeys with the lowest environmental impact 2.2. Explain why incremental development is the most effective approach for developing business software systems. Why is this model less appropriate for real-time systems engineering? 2.3. Consider the reuse-based process model shown in Figure 2.3. Explain why it is essential to have two separate requirements engineering activities in the process. 2.4. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process. 2.5. Describe the main activities in the software design process and the outputs of these activities. Using a diagram, show possible relationships between the outputs of these activities. 2.6. Explain why change is inevitable in complex systems and give examples (apart from prototyping and incremental delivery) of software process activities that help predict changes and make the software being developed more resilient to change.
2.7. Explain why systems developed as prototypes should not normally be used as production systems. 2.8. Explain why Boehm's spiral model is an adaptable model that can support both change avoidance and change tolerance activities. In practice, this model has not been widely used. Suggest why this might be the case. 2.9. What are the advantages of providing static and dynamic views of the software process as in the Rational Unified Process? 2.10. Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology? syf 71 72

AGILE SOFTWARE DEVELOPMENT

So fast changing everything, so the software should keep up with them or they fail. If the development process for a project is long, by the end of it, the customers' needs could have changed and that project may become useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential

Rapid Software Development fundamental characteristics:

1. The processes of specification, design, and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document only defines the most important characteristics of the system.
2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version.
3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface.

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Agile methods

Agile methods universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system.

Probably the best-known agile method is extreme programming. Other agile approaches include **Scrum, Crystal, Adaptive Software Development, DSDM, and Feature Driven Development**.

The success of these methods has led to some integration with more traditional development methods based on system modeling, resulting in the notion of agile modeling and agile instantiations of the Rational Unified Process.

Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this.

Agile methods have been very successful: 1. Product development, where a software company is developing a small or medium-sized product for sale. 2. 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.

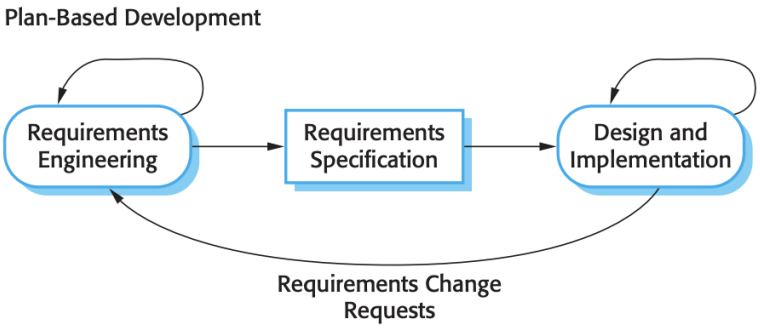
What they have in common:

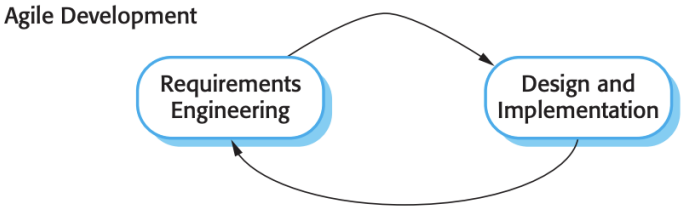
Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer 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.

Bad sides of agile methods:
1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders.
2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
3. Prioritizing changes can be extremely difficult, every stakeholder can prioritize sth else.
4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.
5. Many organizations, especially large companies, may have problems adapting to this new methods as they are so used to the old ones.
So, the contracts and time are hard but if everything goes well, super methodologies.
Two questions when developing by agile methods and maintanance:
1. Are systems that are developed using an agile approach maintainable, given minimal documation?-minimal document but well structured well written code, but still document would be better to see all the requirements.
2. Can agile methods be used effectively for evolving a system in response to customer change requests?-hard because if the customers isnt there full time, anlaşmazlıklar will happen.

Also, the team members should get along really well and communicate so much.

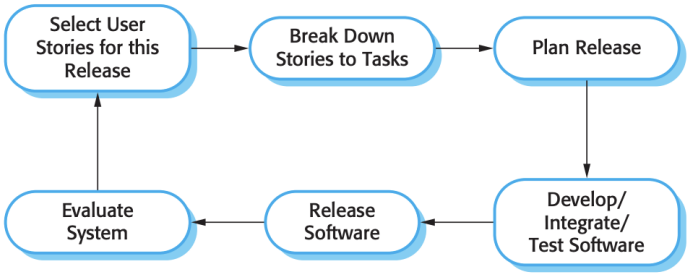
Plan-driven and agile development





Most software projects include practices from plan-driven and agile approaches, how to balance you have to answer a range of technical, human, and organizational questions.

Extreme programming(XP)
Best known and widely used. Very fast.



In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system.
The requirements are not specified as lists of required system functions. Rather, story.
1. Incremental development is supported through small, frequent releases of the system.
2. Customer involvement is very good, customer representative is also in development and creates tests.
3. People, not process, are supported through pair programming, collective ownership of the system code, and good working hours.
4. Change is embraced through regular system releases to customers, test-first development, and refactoring to avoid code degeneration.
5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

spike: an increment where no programming is done. Extreme programming takes an ‘extreme’ approach to incremental development. Very rapidly new versions of the software, and release deadlines are never missed.
In XP, the software should always be refactored.This means that the programming team look for possible improvements to the software and implement them immediately. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code etc. But ofc sometimes implementation of new functionality is more important then refactoring so it is delayed.

Testing in XP
The key features of testing in XP are:
1. Test-first development, /you write the test before, then the code. (you can run the test while writing the code.)
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.
Customer develops acceptance tests. But usually they cant and dont have time.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:
1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally.
3. It is 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
Programmers work in pairs. They are all in the same workstation, pairs are created dynamically so that all team members work with each other during the development process.

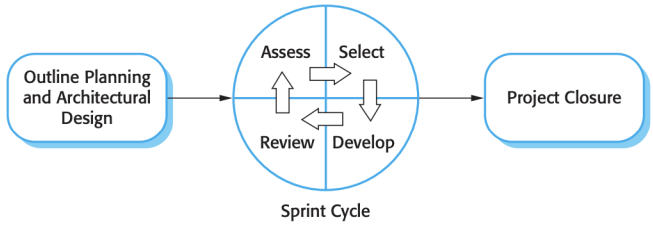
The use of pair programming has a number of advantages:
1. It supports the idea of collective ownership and responsibility for the system.
2. It acts as an informal review process because each line of code is looked at by at least two people, but time consuming to organize, and chaper
3. It helps support refactoring, which is a process of software improvement.

They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits but these did not fully compensate for the pair-programming overhead.

Agile project management

The Scrum approach is a general agile method but its focus is on managing iterative development rather than specific technical approaches to agile software engineering. Scrum does not prescribe the use of programming practices such as pair programming and test-first development. It can therefore be used with more technical agile approaches, such as XP, to provide a management framework for the project.

There are three phases in Scrum. The first 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. Finally, 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.



A Scrum sprint is a planning unit in which the work to be done is assessed, features are selected for development, and the software is implemented. At the end of a sprint, the completed functionality is delivered to stakeholders.

Key characteristics of this process are as follows:

1. Sprints are fixed length, normally 2–4 weeks.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The customer is closely involved in this process.
3. 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.
4. Once these are agreed, the team organizes themselves to develop the software. Short daily meetings involving all team members are held to review progress and if necessary, reprioritize work. 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. The way in which the work is done depends on the problem and the team.
5. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

‘Scrum master’ is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog..

Advantages:
1. The product is broken down into a set of manageable and understandable chunks.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything and consequently team communication is improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works.
5. 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 were small or middle-sized companies that could work together in the same room. But large companies also need faster delivery of software .
Large companies: –gözden geçir ve kısalt!
1. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.
2. 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.
3. 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. This is not necessarily compatible with incremental development and frequent system integration.
4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
5. 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.
6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

There are two perspectives on the scaling of agile methods:
1. A 'scaling up' perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team.
2. A 'scaling out' perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

Agile methods have to be adapted to cope with large systems engineering.
It is essential to maintain the fundamentals of agile methods—flexible planning, frequent system releases, continuous integration, test-driven development, and good team communications.
1. 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. The software architecture has to be designed and there has to be documentation produced to describe critical aspects of the system, such as database schemas, the work breakdown across teams, etc.
2. 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. A range of communication channels such as e-mail, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
3. It is essential to maintain frequent system builds and regular releases of the system. This may mean that new configuration management tools that support multi-team software development have to be introduced.

It is difficult to introduce agile methods into large companies for a number of reasons:
1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. 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.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are also people that arent skilled as much.
4. There may be cultural resistance to agile methods.

—---
3.1. Explain why the rapid delivery and deployment of new systems is often more important to businesses than the detailed functionality of these systems.
3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.
3.3. When would you recommend against the use of an agile method for developing a software system?
3.4. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.
3.5. Explain why test-first development helps the programmer to develop a better understanding of the system requirements. What are the potential difficulties with test-first development?
3.6. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.
3.7. Compare and contrast the Scrum approach to project management with conventional

plan-based approaches, as discussed in Chapter 23. The comparisons should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.

3.8. You are a software manager in a company that develops critical control software for aircraft. You are responsible for the development of a software design support system that supports the translation of software requirements to a formal software specification (discussed in Chapter 13). Comment on the advantages and disadvantages of the following development strategies:

a. Collect the requirements for such a system from software engineers and external stakeholders (such as the regulatory certification authority) and develop the system using a plan-driven approach.

b. Develop a prototype using a scripting language, such as Ruby or Python, evaluate this prototype with software engineers and other stakeholders, then review the system requirements. Redevelop the final system using Java.

c. Develop the system in Java using an agile approach with a user involved in the development team.

3.9. It has been suggested that one of the problems of having a user closely involved with a software development team is that they ‘go native’; that is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem and discuss the advantages and disadvantages of each approach.

3.10. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using agile methods, which rely on close team working and pair programming. Discuss the difficulties that this new policy might cause and how you might get around these problems.

REQUIREMENTS ENGINEERING

The requirements for a system are the descriptions of what the system should do— the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.

'user requirements' to mean the high-level abstract requirements and 'system requirements' to mean the detailed description of what the system should do

1. User requirements are 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.
2. System requirements are more detailed descriptions of 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.

User Requirement Definition

1. The MHC-PMS 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 automatically 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., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

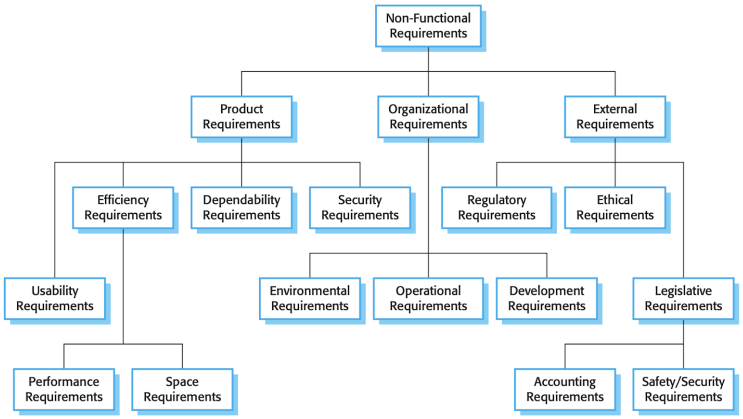
Functional and non-functional requirements

1. Functional requirements: These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.

2. Non-functional requirements: These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards.
In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest.

Functional requirements
The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
Example MHC-PMS: (mental health care - patient management system)
1. A user shall be able to search the appointments lists for all clinics. 2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day. ..
Non-functional requirements
such as performance, security, or availability
Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable.
For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation.
The implementation of these requirements may be diffused throughout the system. There are two reasons for this:
1. 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.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation.



PRODUCT REQUIREMENT
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS 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.

A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express 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::wrong.
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::correct
Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems.

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

The software requirements document

//Software requirements specification(SRS)

An official document about what the developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements.

Agile development methods argue that document is a large effort thats wasted.

Requirements documents are essential for outsider developers.

XP collects user requirements incrementally and write user stories. The user then prioritizes requirements for implementation in the next increment of the system.(bussiness systems)

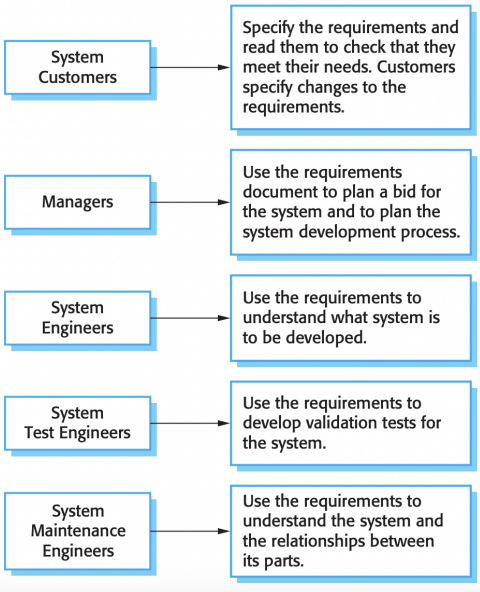


Figure 4.6 Users of a requirements document

The diversity of possible users(reader of the document); the document should be clear for everyone who reads it, defining the requirements in precise detail for developers and testers, and including information about possible system evolution.
Critical systems: detailed document (safety and security have to be analyzed in detail)
The information that is included in a requirements document depends on the type of software being developed and the approach to development that is to be used.
IEEE standard for requirements documents:

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 non-functional 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.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, 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.

– include a comprehensive table of contents and document index so that readers can find the information that they need.

Requirements specification
The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, discriminating architecture and software of the project is almost impossible.
Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts, show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances.

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.*)

Natural language specification
natural language will continue to be the most widely used way of specifying system and software requirements
To minimize misunderstandings when writing natural language requirements:
1. Invent a standard format and ensure that all requirement definitions adhere to that format. (for ex: all requirements are one sentence)
2. Use language consistently to distinguish between Mandatory - "shall" , Desirable - "should."
3. Use text highlighting (bold, italic, or color) for key parts.
4. Do not assume that readers understand technical software engineering language. Avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement.

Structured specifications

Defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band;

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.

Structured specifications

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 understandability of natural language but ensures that some uniformity is imposed on the specification.

VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements..

Use of standard templates.

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

Tabular specification of computation for an insulin pump

Standard form:

1. A description of the function or entity being specified.

2. A description of its inputs and where these come from.
3. A description of its outputs and where these go to.
4. Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
5. A description of the action to be taken.
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.
+ tables or graphical models of the system.

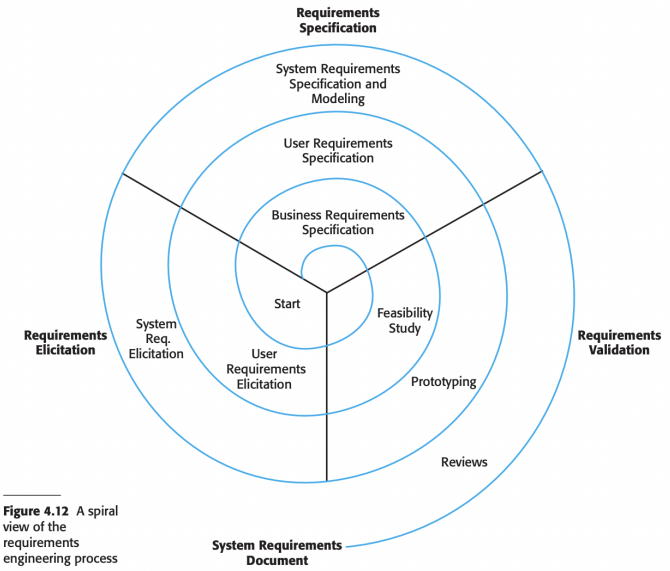
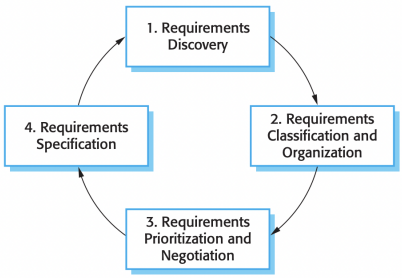


Figure 4.12 A spiral view of the requirements engineering process

Requirements engineering processes may include four high-level activities. These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation). But, requirements engineering is an iterative process in which the activities are interleaved. Iterative process around a spiral.

Requirements elicitation and analysis

In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.



A system stakeholder is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include endusers who will interact with the system and anyone else in an organization who will be affected by it. Other system stakeholders might be engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.
The process activities are:
1. Requirements discovery: Interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. Requirements classification and organization: The unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters.
3. Requirements prioritization and negotiation: Inevitably, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.
4. Requirements specification: The requirements are documented and input into the next round of the spiral.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:
1. Stakeholders often don't know what they want from a computer system except in the most general terms. Also, they naturally express requirements in their own terms and with implicit knowledge of their own work.
3. Different stakeholders have different requirements and they may express these in different ways.
4. Political factors may influence the requirements of a system. (for ex, a manager's influence on the firm)
5. The economic and business environment in which the analysis takes place is dynamic.
Requirements discovery is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.
Sources: documentation, system stakeholders, and specifications of similar systems. You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.

For example, system stakeholders for the mental healthcare patient information system include:
Patients, doctors, nurses, medical receptionists, IT staff, medical ethics manager, healthcare managers, medical records staff.
requirements sources (stakeholders, domain, systems)
Interviewing In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. 1. Closed interviews, where the stakeholder answers a pre-defined set of questions. 2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs. Very good to understand the project but not efficient to understand the requirements from the application domain. Because, terminology and jargon should be used but stakeholders dont know. Sometimes stakeholders dont explain sth becuse they think everybody knows it.
Effective interviewers: open-minded, ask good questions that keep the flow of the interview.
Scenarios Real life examples are relatable, easier to understand. Scenarios can be particularly useful for adding detail to an outline requirements description. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system. XP uses them.
At its most general, a scenario may include:
1. A description of what the system and users expects when the scenario starts.

2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

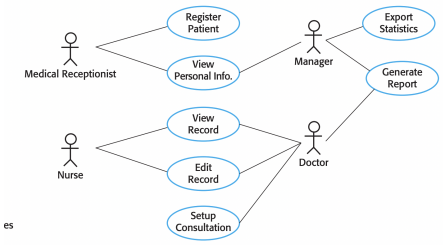
Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

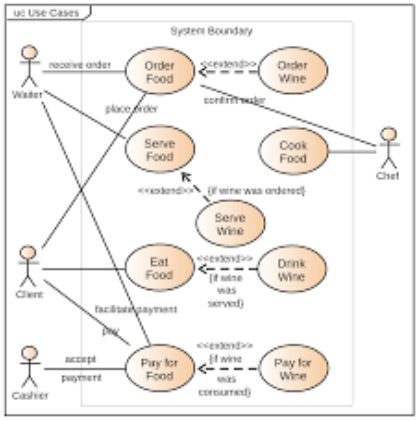
When a new patient attends a clinic, a new record is created by a medical receptionist and personal information (name, age, etc.) is added to it. A nurse then interviews the patient and collects medical history. The patient then has an initial consultation with a doctor who makes a diagnosis and, if appropriate, recommends a course of treatment. The scenario shows what happens when medical history is collected.

Use cases A fundamental feature of the unified modeling language. Identifies the actors involved in an interaction and names the type of interaction + additional info.

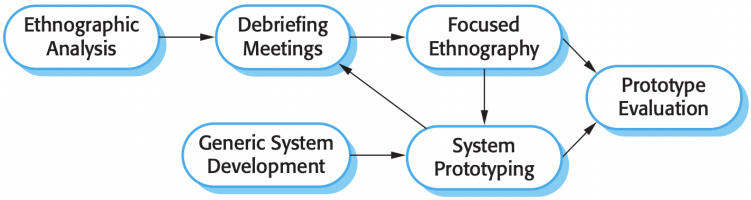


es

A use case is a written description of how users will perform tasks on your website. It outlines, from a user's point of view, a system's behavior as it responds to a request. Each use case is represented as a sequence of simple steps, beginning with a user's goal and ending when that goal is fulfilled.

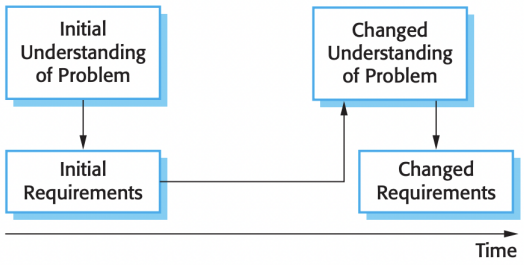


Ethnography Satisfying these social and organizational requirements is often critical for the success of the system.



Ethnography is particularly effective for discovering two types of requirements:
1. Requirements that are derived from the way in which people actually work.
2. Requirements that are derived from cooperation and awareness of other people's activities.

Requirements validation is the process of checking that requirements actually define the system that the customer really wants.
What to check:
1. Validity checks 2. Consistency checks 3. Completeness checks 4. Realism checks 5. Verifiability

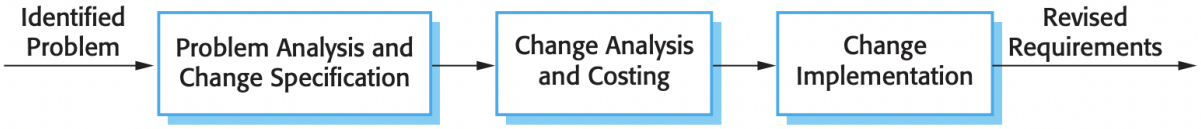


Requirements Validation Techniques:
1. Requirements reviews A team of reviewers.
2. Prototyping A model of the system in question is demonstrated to end-users and customers.
3. Test-case generation Requirements should be testable. If not, the requirement is hard, should be changed.

Requirements management
The requirements for large software systems are always changing. Reason, these systems are "wicked-problems" – problems that cannot be completely defined.

why change is inevitable:
1. The business and technical environment of the system always changes after installation. New hardware may be introduced etc.
2. 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.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

Requirements management planning



1. Requirements identification Each requirement must be uniquely identified.
2. A change management process This is the set of activities that assess the impact and cost of changes.
3. Traceability policies These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
4. Tool support Requirements management involves the processing of large amounts of information about the requirements.

Requirements change management

Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation

1. Problem analysis and change specification The problem or change proposal is analyzed to its validity.
2. Change analysis and costing Cost of making the change, is it worth it?
3. Change implementation The requirements document and the system design and implementation are modified.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document.

Agile development processes, such as extreme programming, have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped.

—

4.1. Identify and briefly describe four types of requirement that may be defined for a computerbased system.
4.2. Discover ambiguities or omissions in the following statement of requirements for part of a ticket-issuing system:
An automated ticket-issuing system sells rail tickets. Users select their destination and input a credit card and a personal identification number. The rail ticket is issued and their credit card account charged. When the user presses the start button, a menu display of potential destinations is activated, along with a message to the user to select a destination.

Once a destination has been selected, users are requested to input their credit card. Its validity is checked and the user is then requested to input a personal identifier. When the credit transaction has been validated, the ticket is issued.

4.3. Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in an appropriate way.

4.4. Write a set of non-functional requirements for the ticket-issuing system, setting out its expected reliability and response time.

4.5. Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

■ An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

■ The cash-dispensing function in a bank ATM.

■ The spelling-check and correcting function in a word processor.

4.6. Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

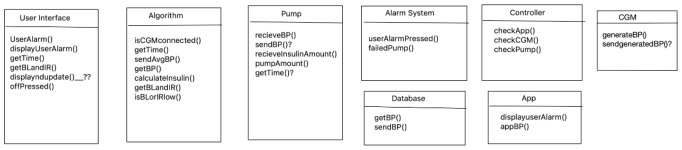
4.7. Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.

4.8. Who should be involved in a requirements review? Draw a process model showing how a requirements review might be organized.

4.9. When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.

4.10. You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. However, you have also a responsibility of confidentiality to your previous employer.

INSULIN PUMP DOCUMENTATION



pump keeps recevinig the info until x3 - for loop

max insulin amount, "max insulin has been given for the day" warning

variables-types and names

if(notified){
update App();
}

Controller:connection checker

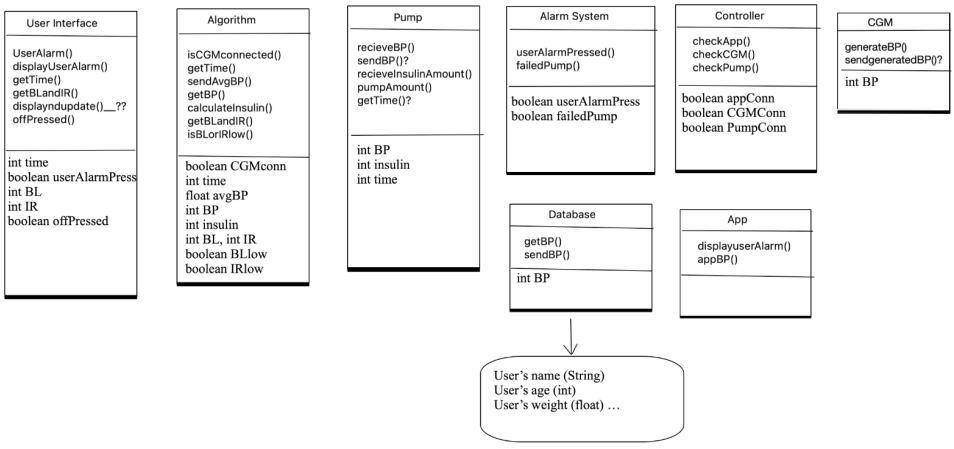
displayndupdate() -
for displaying all the necessary info in the user interface and to keep the user updated

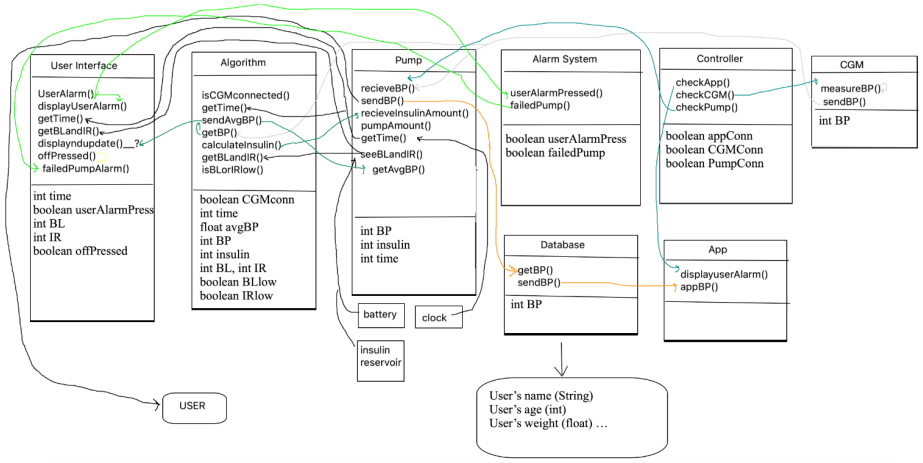
send metodu var mı yoksa sadece recieve mi

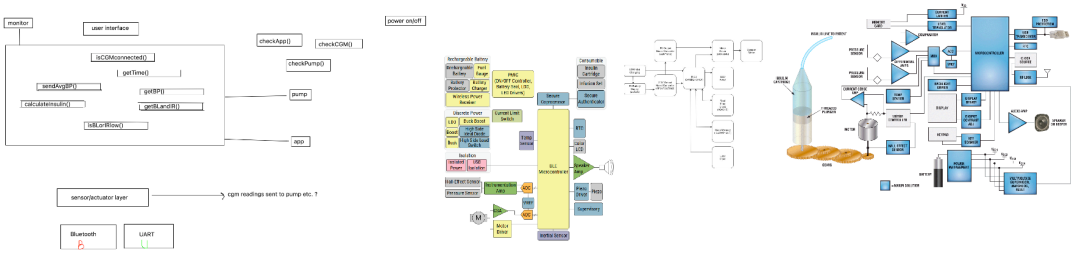
-user turns on (connects the gcm to the device) -gcm connected or not checker (algorithm)
-cgm sends blood level every 5 minutes -pump should recieve the info every 5 minutes, and sends the info to user interface at the same time
-after 3 blood levels, the algorithm devreye girmek and calculates the needed insulin to be pumped
-sends the amount information to the pump, the pump pumps that amount. Input: amount output: I have pumped that much insulin.
-after every pump, the battery level and insulin reservoir info are sent to user interface and algorithm. The algorithm is checking the battery level and insulin reservoir -if the reseorvior level is less then the max insulin amount that could be pumped – alarm is sent to the alarm system.
-the user interface; only the screen that the user sees can be closed. On/off switch. How to do that, because only the screen goes dark but everything is occurring the same.
-the hardware comes with embedded clock (rtos), the clock information is always displayed on the user interface
-errors may occur in; pumping,
According to the blood pressure level information gotten from the CGM, basal insulin delivery is adjusted, and/or a bolus is recommended or given automatically.
-on the user interface, if the user presses alarm, the alarm system is evoked
-when the app's user presses "güncelle the information" and there is Bluetooth connection (if not, error comes), the pumps last infos are sent to the app , the apps database consists of up to 60 days.

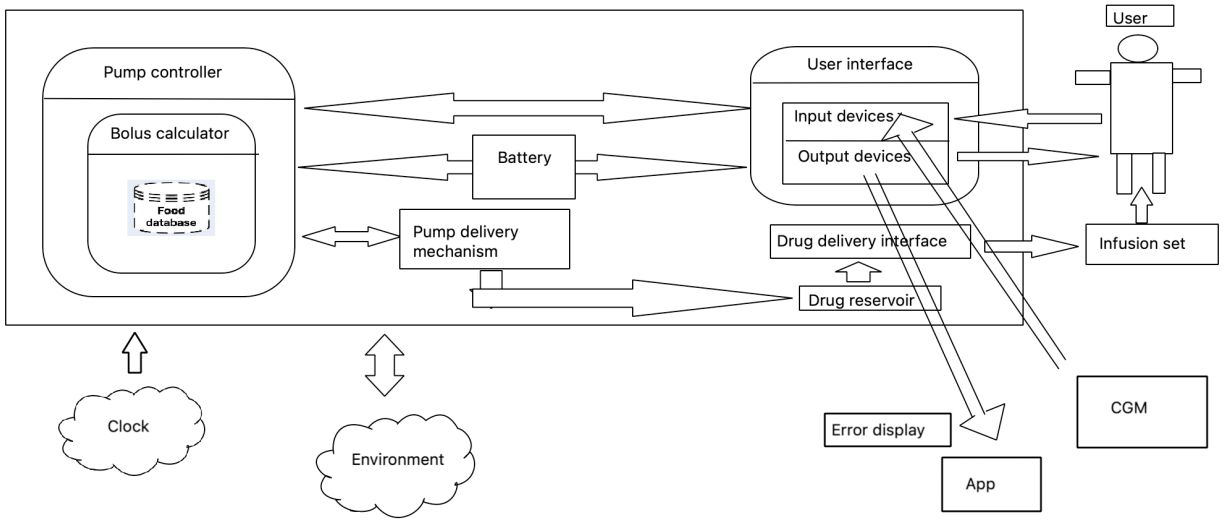
-alarm system
-algorithm: battery and reservoir level + calculations of needed pump + takes the average of the pumped insulin amount every hour and sends it? To the pump's user interface + cgm connected or not checker
-pump: responsible for pumping the emredilen amount + has user interface that shows the remaining levels and the insulin level read from cgm and the 3 last insulin levels are shown + user interface also shows how much insulin is pumped on average every hour its güncellenir (by the algorithm)
-database: (different) last 3? insulin levels, insulin amount thats pumped
-error control: battery and insulin reservoir if there is error immediately goes to alarm system + connection error with either devices bluetoth or uart
-user interface

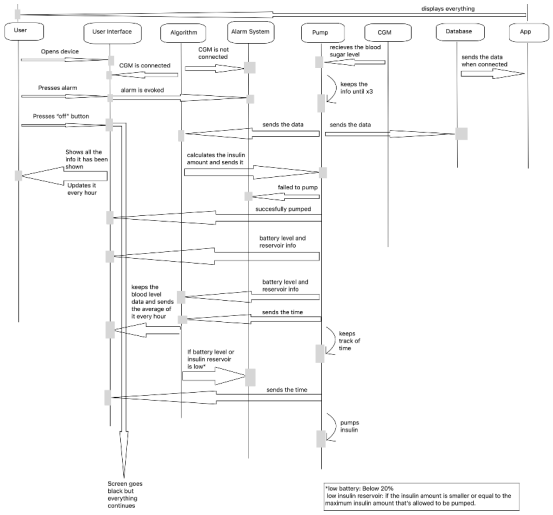
isCGMconnected(){boolean true(UI) false(AS);;}
userAlarm() ->when called, calls alarms systems UserAlarmPressed() method
userAlarmPressed() -> sends alarm notification to UI and app that "user pressed alarm!"
getTime() -> gets time info from the pump's embedded clock (this method is always working)?
sendAvgBP() -> keeps the BP info and takes the avg of it and sends it to UI each hour ?
recieveBP() -> gets from CGM
getBP() -> gets from pump
calculateInsulin() -> if-else statements to calculate, and sends the amount into the pump
pumpAmount() -> pumps that amount in —— secs/minutes? after this, if else to check successful pump or not?
failedPump() -> sends alarm note to UI and app.
isBLorIRlow() -> if else.. sends to alarm system.
offPressed() -> screen goes black but everything else keeps working..?
BLlow() -> sends alarm note to UI and app.
IRlow() -> sends alarm note to UI and app.
updateApp() -> send the most recent info when connection is made (checkApp method has if else.)

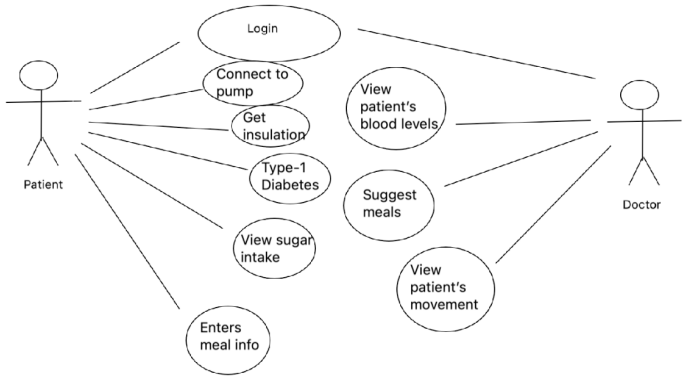


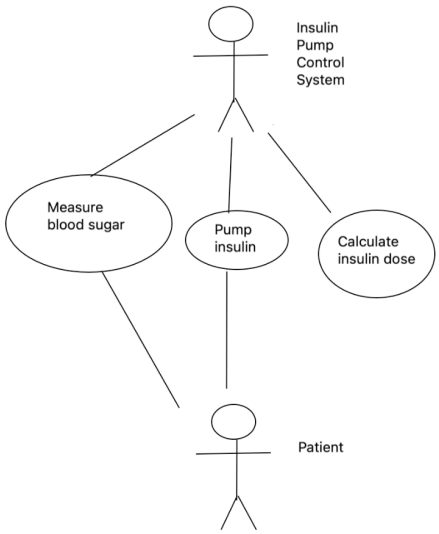

User Interface
UserAlarm()
displayUserAlarm()
getTime()
getBLandIR()
displayndupdate()__?
offPressed()
failedPumpAlarm()
int time
boolean userAlarmPress
int BL
int IR
boolean offPressed
Algorithm
isCGMconnected()
getTime()
sendAvgBP()
getBP()
calculateInsulin()
getBLandIR()
isBLorIRlow()
boolean CGMconn
int time
float avgBP
int BP
int insulin
int BL, int IR
boolean BLlow
boolean IRlow
Pump
recieveBP()
sendBP()
recieveInsulinAmount()
pumpAmount()
getTime()
seeBLandIR()
getAvgBP()
int BP
int insulin
int time
Alarm System
userAlarmPressed()
failedPump()
boolean userAlarmPress
boolean failedPump
Controller
checkApp()
checkCGM()
checkPump()
boolean appConn
boolean CGMConn
boolean PumpConn
CGM
measureBP()
sendBP()
int BP
Database
getBP()
sendBP()
int BP
App
displayuserAlarm()
appBP()
battery
clock
insulin reservoir
USER
User's name (String)
User's age (int)
User's weight (float) …

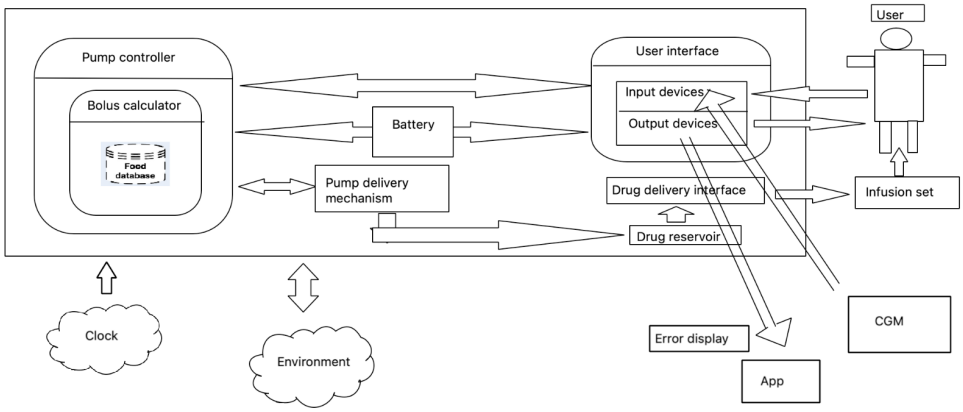

monitor
user interface
isCGMconnected()
getTime()
sendAvgBP()
getBP()
calculateInsulin()
getBLandIR()
isBLorIRlow()
checkApp()
checkCGM()
checkPump()
pump
app
sensor/actuator layer
cgm readings sent to pump etc. ?
Bluetooth
UART
power on/off

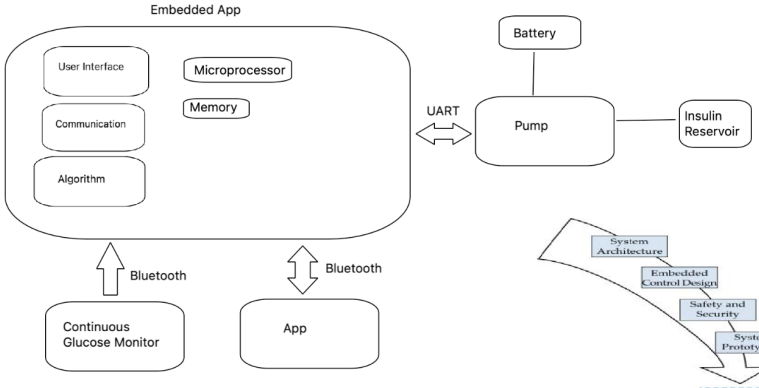

Pump controller
Bolus calculator
Food database
Battery
Pump delivery mechanism
User interface
Input devices
Output devices
Drug delivery interface
Drug reservoir
User
Infusion set
CGM
Error display
App
Clock
Environment

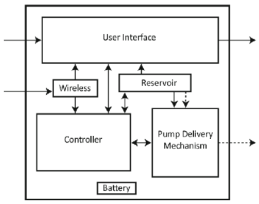

displays everything
User
User interface
Algorithm
Alarm System
Pump
CGM
Database
App
Opens device
CGM is connected
CGM is not connected
recieves the blood sugar level
sends the data when connected
Presses alarm
alarm is evoked
keeps the info until x3
Presses "off" button
sends the data
sends the data
Shows all the info it has been shown
Updates it every hour
calculates the insulin amount and sends it
failed to pump
succesfully pumped
battery level and reservoir info
battery level and reservoir info
keeps the blood level data and sends the average of it every hour
sends the time
keeps track of time
If battery level or insulin reservoir is low*
sends the time
pumps insulin
Screen goes black but everything continues
*low battery: Below 20%
low insulin reservoir: If the insulin amount is smaller or equal to the maximum insulin amount that's allowed to be pumped.

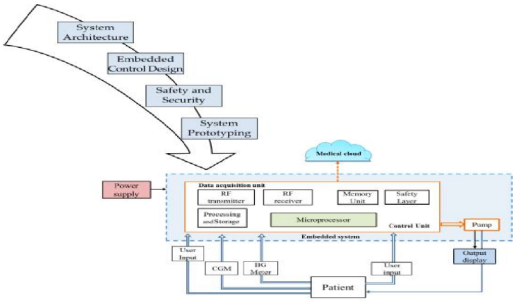

Login
Connect to pump
Get insulation
Type-1 Diabetes
View sugar intake
Enters meal info
Patient
View patient's blood levels
Suggest meals
View patient's movement
Doctor


Insulin Pump Control System
Measure blood sugar
Pump insulin
Calculate insulin dose
Patient


Pump controller
Bolus calculator
Food database
Battery
Pump delivery mechanism
User interface
Input devices
Output devices
User
Drug delivery interface
Drug reservoir
Infusion set
Clock
Environment
Error display
App
CGM


Embedded App
User Interface
Microprocessor
Memory
Communication
Algorithm
Bluetooth
Bluetooth
Continuous Glucose Monitor
App
Battery
UART
Pump
Insulin Reservoir


User Interface
Wireless
Reservoir
Controller
Pump Delivery Mechanism
Battery


System Architecture
Embedded Control Design
Safety and Security
System Prototyping
Medical cloud
Power supply
Data acquisition unit
RF transmitter
RF receiver
Memory Unit
Safety Layer
Processing and Storage
Microprocessor
Control Unit
Pump
Embedded system
User Input
CGM
BG Meter
User input
Output display
Patient

Comparisons

PUMPS	Medtronic 770G/780G	Tandem T:Slim x2	Omnipod DASH	Insulin Pump
Approved For	All Ages	>6 yo	All Ages	Suitable for all ages, but the design is for children.
Weight	95.7 g	112 g	26 g	105 g
Tubing	45-110 cm	60-110 cm	Tube-free	40-120 cm
Cannula	6-17 mm	6-13 mm	9mm with 6.5mm under the skin at 50 degree angle	6-12 mm
Reservoir	1.8 and 3 ml	0.95-3.0 ml	2.0 ml	3.0 ml
Battery	AA lithium/Alkaline	Rechargeable	Rechargeable	Rechargeable
Waterproof	3.6m, 24hrs (IPX8)	1m, 30mins (IPX7)	7.6m, 60mins (IP28)	3.6m, 24hrs (IPX8)
Glucose Meter	Accu-Chek Guide Link Meter	Verio Meter provided – Not Linke	None	Dexcom G6
Bolus Calculator	In-built in pump	In-built in pump	In-built in PDM	In-built in pump
Basal Range	0.0-35 units/hr	0.1-15 units/hr	0.0-40 units/hr	0.0-35 units/hr
Bolus Range	0.0 – 25 units	0.05 – 25 units	0.05 – 30 units	0.0 – 30 units
Basal Increment	0.025, 0.05, 1.0	0.001	0.05	0.01, 0.025, 0.05,1.0
Uploading	Automatic uploads via MiniMed Mobile App	Tandem USB cable	USB cable	Automatic uploads via Mobile App
Software	Carelink Web based	Diasend	Diasend	Carelink Web Based
CGM Integration	Yes	Yes	No	Yes
Loop Type	Hybrid-Closed	Hybrid-Closed	Hybrid-Closed	Closed