|  |
| --- |
|  |
| Software Engineering |
| Part I. Software Engineering Models |

|  |
| --- |
| Dr. Moghadampour, Ghodrat  Spring 2023 |

**Contents**

1. [General](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Gen)
2. [Software Engineering](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#SEAO)
   1. [Software Development Process Models](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#SDPM)
      1. [Pure Waterfall Model](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#PWM)
      2. [Modified Waterfall Model](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#MWM)
      3. [The Spiral Model](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#TSM)
      4. [Evolutionary Prototyping](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#EP)
      5. [Rational Unified Process](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#RUP)
         1. [Develop Software Iteratively](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#DSI)
         2. [Manage Requirements](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#MR)
         3. [Use Component Based Architecture](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#UCBA)
         4. [Visually Model Software](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#VMS)
         5. [Verify Software Quality](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#VSQ)
         6. [Control Changes to Software](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#CCTS)

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html" \l "Con)  
**Software Engineering, 5 SPs**  
  
**Objectives**  
The main goal of this unit is to give the student knowledge of software development process in general and analysis phase especially. After the unit the student will be able to apply Unified Modelling Language (UML) of software analysis, and will have sufficient capability to utilise other similar methods. The unit emphasises the impact of systematic, well-disciplined software work on productivity and the successful completion of projects.

**Contents**  
Software development process models. Analysis process and software requirements specification report. Feasibility study. Implementation methods of object oriented analysis. Unified Modelling Language (UML). Object and behavior models. Descriptions of state behavior using state transition models. Graphical tools for software specification. Rapid prototyping in software analysis. Validation of software.  
  
**Previous Knowledge**  
Basics of Programming.  
**Assessment**  
Lectures, team works, project work and examinations.  
  
**References**

* **Software Requirements & Specifications**: A Lexicon of Practice, Principles and Prejudices (Acm Press Books), Michael Jackson , Addison-Wesley Pub Co; 1st edition (August 23, 1995) , 0201877120
* **Applying UML and Patterns**: An Introduction to Object- Oriented Analysis and Design and the Unified Process, Craig Larman, Prentice-Hall, Inc., second/2002, 0130925691
* **Software Engineering, A Practitioner's Approach** Roger S. Pressman, adapted by Darrel Ince. Fifth Edition/2000. McGraw-Hill International (UK) Limited.
* **Managing The Software Enterprise**, Software Engineering and Information Systems in Context. Pat Hall & Juan Fernandez-Ramil. Thomson Learning, 2007.
* [OMG Site](http://www.omg.org/uml/)
* [SmartDraw Site for Downloading Drawing Software](http://www.smartdraw.com/index.htm)

**1. Software Engineering**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity that is to be engineered, the following questions must be asked and answered:

* What is the problem to be solved?
* What are the characteristics of the entity that is used to solve the problem?
* How will the entity ( and the solution) be realized?
* How will the entity be constructed?
* What approach will be used to uncover errors that were made in the design and construction of the entity?
* How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity?

To engineer software adequately, a software development process must be defined. The work associated with software engineering can be categorized into three generic phases:

1. The **definition phase** focuses on what. During the definition The key requirements of the system and the software are identified. The software developer attempts to identify: what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. During the definition three major tasks will occur in some form:
   1. system or information engineering
   2. software project planning
   3. requirements analysis
2. The **development phase** focuses on how. During development a software engineer attempts to define:
   1. how data are to be structured,
   2. how function is to be implemented as a software architecture,
   3. how procedural details are to be implemented,
   4. how interfaces are to be characterized,
   5. how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed.

The methods applied during the development phase will vary, but three specific technical tasks should always occur:

* 1. software design,
  2. code generation, and
  3. software testing.

1. The **maintenance phase** focuses on change that is associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancement brought about by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software. Four types of change are encountered during the maintenance phase:
   1. **Correction:** Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects.
   2. **Adaptation:** Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was developed is likely to change. **Adaptive maintenance** results in modification to the software to accommodate changes to its external environment.
   3. **Enhancement:** As software is used, the customer/user will recognize additional functions that will provide benefit. Perfective maintenance extends the software beyond its original functional requirements.
   4. **Prevention:** Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

The phases and related steps described in the generic view of software engineering are complemented by a number of activities:  
Reviews are conducted to ensure that quality is maintained as each step is completed. Documentation is developed and controlled to ensure that complete information about the system and software will be available for later use. Change control is instituted so that changes can be approved and tracked.

**1.1 Software Development Process Models**

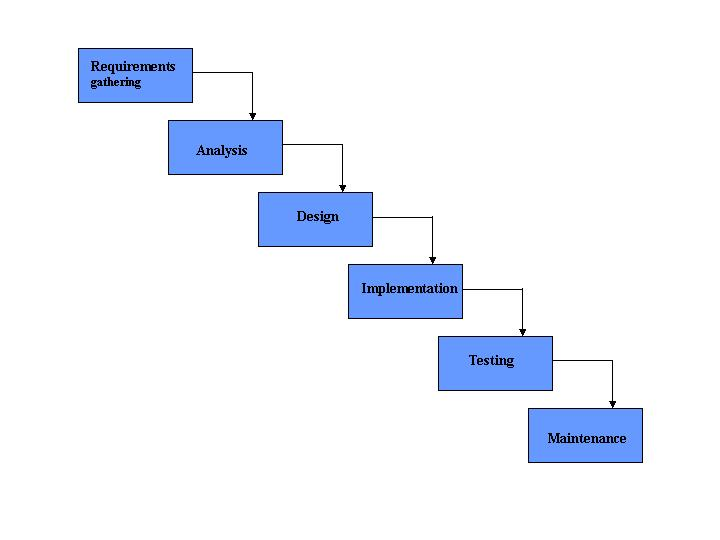
[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
The system development process has been modelled in different manners. In the following we go through some most common models.

**1.1.1 Pure Waterfall Model**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Pure waterfall model is the classical system development model. It consists of discontinuous phases:

* Concept
* Requirements
* Architectural design
* Detailed design
* Coding and development
* Testing and implementation

The pure waterfall performs well for products with clearly understood requirements or when working with well understood technical tools, architectures and infrastructures. It's weaknesses frequently make it inadvisable when rapid development is needed. In those cases, modified models may be more effective.



**Figure 1:** Classic life cycle or waterfall model for software development process. Below are the strengths and weaknesses of classical life cycle model.

|  |  |
| --- | --- |
| **Strengths** | **Weaknesses** |
| Minimizes planning overhead since it can be done up front. | Inflexible |
| Structure minimizes wasted effort, so it works well for technically weak or inexperienced staff. | Only the final phase produces a non-documentation deliverable. |
|  | Backing up to address mistakes is difficult. |

**1.1.2 Modified Waterfall Model**

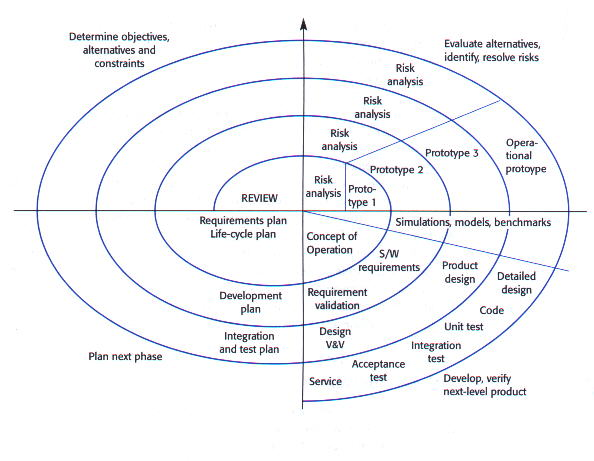
[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
The modified waterfall uses the same phases as the pure waterfall, but is not done on a discontinuous basis. This enables the phases to overlap when needed. The pure waterfall can also split into subprojects at an appropriate phase (such as after the architectural design or detailed design). Risk reduction spirals can be added to the top of the waterfall to reduce risks prior to the waterfall phases. The waterfall can be further modified using options such as prototyping, JADs ([Joint Application Design/Development](http://www.umsl.edu/~sauter/analysis/JAD.html)) or CRC (Class-Responsibility- Collaborators) sessions or other methods of requirements gathering done in overlapping phases.  
  
JAD is a process used in the prototyping life cycle area of the Dynamic Systems Development Method (DSDM) to collect business requirements while developing new information systems for a company. It also includes approaches for enhancing user participation, expediting development, and improving the quality of specifications. It consists of a workshop where knowledge workers and IT specialists meet, sometimes for several days, to define and review the business requirements for the system.  
  
The strengths and weaknesses of the spiral model can be summarized as follows:

|  |  |
| --- | --- |
| **Strengths** | **Weaknesses** |
| More flexibility than the pure waterfall model. | Milestones are more ambiguous than for the pure waterfall. |
| If there is personnel continuity between the phases, documentation can be substantially reduced. | Activities performed in parallel are subject to miscommunication and mistaken assumptions. |
| Inplementation of easy areas do not need to wait for the hard ones. | Unforseen interdependencies can create problems. |

**1.1.3 The Spiral Model**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
For projects with risky elements, it's beneficial to run a series of risk-reduction iterations which can be followed by a waterfall or other non-risk-based lifecycle. The spiral is a risk-reduction oriented model that breaks a software project up into mini-projects, each addressing one or more major risks. After major risks have been addressed, the spiral model terminates as a waterfall model. Rouglly speaking the spiral model involves six steps:

* Determine objectives, alternatives and constraints.
* Identify and resolve risks.
* Evaluate alternatives.
* Develop the deliverables for that iteration and verify that they are correct.
* Plan the next iteration.
* Commit to an approach for the next iteration.

The following is a spiral model of software development process:  


**Figure 2:** The spiral model for software development process ([Reference](http://www.kendall-consulting.com/projmgt.html)).  
  
The spiral approach provides participants with a series of iterations and connections to the project, which encourages participation, shared understanding and sponsorship, and eases implementation. Each connection affords project participants the opportunity to share ideas and test concepts as the project progresses, rather than waiting until the end of a more linear project life cycle, when changes and rework can become extremely costly. To successfully employ this approach, it is important that project goals and scopes be flexible enough to accommodate revisions that will enhance the ultimate outcome. The new objective of projects is to learn, and if we do that right, the deliverables get produced as a natural by-product of the process.  
The steps in the spiral model can be identified in more details as follows:

1. The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.
2. A preliminary design is created for the new system.
3. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.
4. A second prototype is evolved by a fourfold procedure:
   1. evaluating the first prototype in terms of its strengths, weaknesses, and risks
   2. defining the requirements of the second prototype
   3. planning and designing the second prototype
   4. constructing and testing the second prototype
5. At the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation, or any other factor that could, in the customer's judgment, result in a less-than- satisfactory final product.
6. The existing prototype is evaluated in the same manner as was the previous prototype, and, if necessary, another prototype is developed from it according to the fourfold procedure outlined above.
7. The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired.
8. The final system is constructed, based on the refined prototype.
9. The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

The strengths and weaknesses of the spiral model can be summarized as follows:

|  |  |
| --- | --- |
| **Strengths** | **Weaknesses** |
| Early iterations of the project are the cheapest, enabling the highest risks to be addressed at the lowest total cost. This ensures that as costs increase, risks decrease. | It is complicated and requires attentive and knowledgeable management to pull it off. |
| Each iteration of the spiral can be tailored to suit the needs of the project. |  |

**1.1.4 Evolutionary Prototyping**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Evolutionary prototyping uses multiple iterations of requirements gathering and analysis, design and prototype development. After each iteration, the result is analyzed by the customer. Their response creates the next level of requirements and defines the next iteration.  
The strengths and weaknesses of the evolutionary prototyping model can be summarized as follows:

|  |  |
| --- | --- |
| **Strengths** | **Weaknesses** |
| Customers can see steady progress. | It is impossible to know at the outset of the project how long it will take. |
| This is useful when requirements are changing rapidly, when the customer is reluctant to commit to a set of requirements, or when no one fully understands the application area. | There is no way to know the number of iterations that will be required. |

**1.1.5 Rational Unified Process**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
The Rational Unified Process (RUP) is a software design methodology created by the Rational Software Corporation. It describes how to effectively deploy software using commercially proven techniques, and is a heavyweight process (also described as a Thick methodology), and hence particularly applicable to larger software development teams working on large projects.  
  
The RUP defines the following guidelines and templates for team members to follow during a product's lifecycle.

1. Develop Software Iteratively
2. Manage Requirements
3. Use Component Based Architecture
4. Visually model software
5. Verify software quality
6. Control changes to software

**1.1.5.1 Develop Software Iteratively**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Given the time it takes to develop large sophisticated software systems it not possible to define the problem and build the solution in a single step. Requirements will often change throughout a projects development, due to architectural constraints, customer's needs or a greater understanding of the original problem. Iteration allows greater understanding of a project through successive refinements and addresses a projects highest risk items at every stage of its lifecycle. Ideally each iteration ends up with an executable release; this helps reduce a projects risk profile, allows greater customer feedback and help developers stay focused.

**1.1.5.2 Manage Requirements**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
A documentation framework is essential for any large project; hence RUP describes how to document functionality, constraints, design decisions and business requirements. Use Cases and Scenarios are examples of artifacts prescribed by the process and have been found to be very effective at both capturing functional requirements and providing coherent threads throughout the development and deployment of the system.

**1.1.5.3 Use Component Based Architecture**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Component Based Architecture creates a system that is easily extensible, promotes software reuse and intuitively understandable. A component often relates to an object in Object Orientated Programming. The RUP provides a systematic way to build this kind of system, focusing on producing an early executable architecture before committing full scale resources on a project. These components are often assembled within existing infrastructures such as [CORBA](http://www.fact-index.com/c/co/corba.html) (Common Object Request Broker Architecture) and [COM](http://www.fact-index.com/c/co/component_object_model.html) (Component Object Model).

**1.1.5.4 Visually Model Software**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Abstracting your programming from it's code and representing it using graphical building blocks is an effective way to get an overall picture of a solution. It can also allow less technically competent individuals who may have a better understanding of the problem to have a greater input. Unified Modelling Language (UML) is the industry standard way of representing projects and is hence usually used by the RUP.

**1.1.5.5 Verify Software Quality**

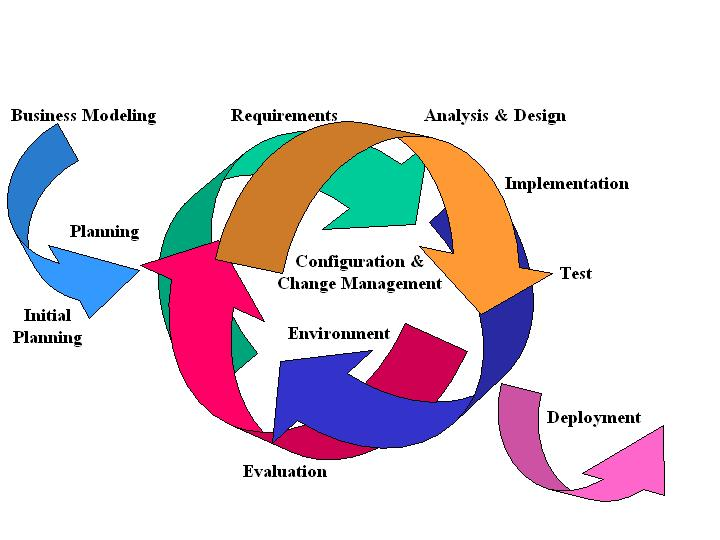
[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Quality Assessment is the most common failing point of all software projects, often an afterthought in such projects and even handled by a different team. The RUP assists in planning quality control and assessment built into the entire process involving all members of a team.

**1.1.5.6 Control Changes to Software**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
In all software projects change is inevitable, the RUP defines methods to control track and monitor changes. As a seemingly small change can affect applications in entirely unpredictable ways this is essential for a successful project. The RUP also defines secure workspaces allowing a programmer to be guaranteed that changes in another system will not affect his system. This ties in heavily with Component based architectures.  
  
So far these guidelines are general, to be adhered to throughout a project's lifecycle. To capture the time dimension of a project the RUP divides a project into four distinct phases:

1. Inception
2. Elaboration
3. Construction
4. Transition

The following is a graphical demonstration of Rational Unifed Process:



**Figure 3:** Iterative Development in the RUP. The Rational Unifed Process promotes an iterative approach. In each iteration a little of requirements, analysis, design, implementation and testing is done. Each iteration builds on the work of the previous iteration to produce an executable that is one step closer to the final product.  
  
For more information refer to:

* [Rational Software Corporation homepage](http://www-306.ibm.com/software/rational/)
* [Excellent description of the process from The Rational Edge website](http://www-106.ibm.com/developerworks/rational/rationaledge/)

**1.1.6 Agile Software Development**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Agile software development is a group of software development methodologies based on **iterative** and **incremental** development, where requirements and solutions evolve through collaboration between **self-organizing**, **cross-functional** teams.  
  
Lightweight (opposed to heavyweight) development evolved in the mid-1990s. Heavyweigth methods are typically criticized as heavily regulated, regimented, micromanaged, waterfall model of development. Lightweight methods (agile methods) are considered by their practitioners as a return to the early history of software development practices.

**1.1.6.1 Agile Manifesto**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
The Manifesto for Agile Software Development defines the approach known as agile software development. The followings are the main principles of the Agile Software Development:

* Customer satisfaction by rapid delivery of useful software
* Welcome changing requirements, even late in development
* Working software is delivered frequently (weeks rather than months)
* Working software is the principal measure of progress
* Sustainable development, able to maintain a constant pace
* Close, daily cooperation between businesspeople and developers
* Face-to-face conversation is the best form of communication (co-location)
* Projects are built around motivated individuals, who should be trusted
* Continuous attention to technical excellence and good design
* Simplicity
* Self-organizing teams
* Regular adaptation to changing circumstances

There are many specific agile development methods. Most promote development, teamwork, collaboration, and process adaptability throughout the life-cycle of the project. The main elements of Agile software development are explained below.

**1.1.6.1.1 Small increments**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Agile methods break tasks into small increments with minimal planning, and do not directly involve long-term planning. Iterations are short time frames (timeboxes) that typically last from one to four weeks. Each iteration involves a team working through a full software development cycle including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This minimizes overall risk and allows the project to adapt to changes quickly. Documentation is produced as required. An iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration. Multiple iterations may be required to release a product or new features.

**1.1.6.1.2 Cross-functional and self-organizing Teams**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Team composition in an agile project is usually cross-functional and self-organizing without consideration for any existing corporate hierarchy or the corporate roles of team members. Team members normally take responsibility for tasks that deliver the functionality an iteration requires. They decide individually how to meet an iteration's requirements.

**1.1.6.1.3 Face-to-face communication**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Agile methods emphasize face-to-face communication over written documents when the team is all in the same location. Most agile teams work in a single open office (called a bullpen), which facilitates such communication. Team size is typically small (5-9 people) to simplify team communication and team collaboration. Larger development efforts may be delivered by multiple teams working toward a common goal or on different parts of an effort. This may require a coordination of priorities across teams. When a team works in different locations, they maintain daily contact through videoconferencing, voice, e-mail, etc.

**1.1.6.1.4 Customer representative**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
No matter what development disciplines are required, each agile team will contain a customer representative. This person is appointed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment (ROI) and ensuring alignment with customer needs and company goals. Most agile implementations use a routine and formal daily face-to-face communication among team members. This specifically includes the customer representative and any interested stakeholders as observers. In a brief session, team members report to each other what they did the previous day, what they intend to do today, and what their roadblocks are. This face-to-face communication exposes problems as they arise.

**1.1.6.1.5 Working software**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Agile development emphasizes working software as the primary measure of progress. This, combined with the preference for face-to-face communication, produces less written documentation than other methods. The agile method encourages stakeholders to prioritize wants with other iteration outcomes based exclusively on business value perceived at the beginning of the iteration. Specific tools and techniques such as continuous integration, automated or xUnit test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance project agility.

**1.1.6.6 Differences with other methods**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Agile methods are considered to be the opposite end of the spectrum from "plan-driven" or "disciplined" methods; agile teams may, however, employ highly disciplined formal methods.  
  
A more accurate distinction is that methods exist on a continuum from **"adaptive"** to **"predictive"**. Agile methods lie on the "adaptive" side of this continuum. Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well.  
  
**Predictive methods**, in contrast, focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can require completed work to be started over. Predictive teams will often institute a change control board to ensure that only the most valuable changes are considered.  
  
**Formal methods**, in contrast to adaptive and predictive methods, focus on computer science theory with a wide array of types of proves. A formal method attempts to prove the absence of errors with some level of determinism. Some formal methods are based on model checking and provide counter examples for code that cannot be proven Agile methods have much in common with the "Rapid Application Development" techniques.

**1.1.6.7 SCRUM**

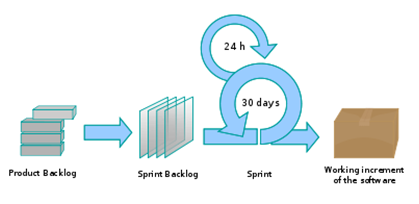
[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Scrum is an Agile programming method, which divides the task into small iteration cycles, which normally take 2-4 weeks. What should be implemented in a cycle must be defined before the iteration, while the needs are fixed during the iteration. A short meeting will be hold everyday (which called a Scrum), people will report what they have done and what will do the next day. In scrum, three characters are defined as the "Pig" role.

* Scrum Master: Maintains the processes
* Product Owner: who represents the stakeholders and the business
* Team: a cross-functional group who do the actual analysis, design, implementation, testing, etc.

The "Pigs" are the ones committed to the project in the Scrum process�they are the ones with "their bacon on the line" and performing the actual work of the project. Another role, which called "Chicken" roles, is made up by the following characters:

* Stakeholders: These are the people who enable the project and for whom the project will produce the agreed-upon benefits, which justify its production. They are only directly involved in the process during the sprint reviews.
* Managers: People who will set up the environment for the product development organizations.

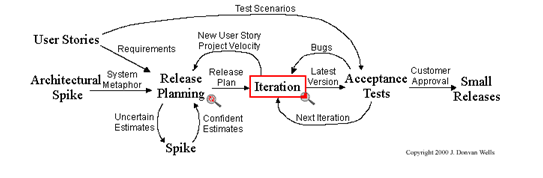
Chicken roles are not part of the actual Scrum process, but must be taken into account. They are people for whom the software is being built.



**Figure 4:** Scrum Procedure(Reference).

**1.1.6.4 Extreme Programming**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.



**Figure 5:** XP procedure(Reference). Other elements of extreme programming include:

* programming in pairs or doing extensive code review, unit testing of all code
* avoiding programming of features until they are actually needed
* a flat management structure
* simplicity and clarity in code
* expecting changes in the customer's requirements as time passes and the problem is better understood
* frequent communication with the customer and among programmers

The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels, on the theory that if some is good, more is better. It is unrelated to "cowboy coding", which is more free-form and unplanned. It does not advocate "death march" work schedules, but instead working at a sustainable pace.

**1.1.6.5 Feature Driven Development**

[Contents](https://portal.vamk.fi/pluginfile.php/341751/mod_resource/content/6/Lectures/software_engineering.html#Con)  
Feature Driven Development (FDD) focuses on short-term iteration and useable features. In FDD, the iteration cycle is usually 2 weeks. The following five tasks are involved in the whole process:

1. Build the overall model
2. Specify the feature list
3. Make plans for the individual tasks
4. Feature design
5. Implementation

Steps 1-3 will be done before the beginning of the task. Steps 4 and 5 will be done during each iteration. In FDD, developers are categorized to "Chief Programmers" and "Class Owners". Chief programmers task the responsibility in designing and coordinating, while class owners in charge of implementation.