[L10P2]

One Destination, Many Paths: Software Process Models

SDLC process models

Software development goes through different stages such as *requirements, analysis, design, implementation* and *testing*. These stages are collectively known as the *software development life cycle* (SDLC). There are several approaches, known as *software development life cycle models* (also called *software process models*) that describe different ways to go through the SDLC. Each process model prescribes a "roadmap" for the software developers to manage the development effort. The roadmap describes the aims of the development stage(s), the artifacts or outcome of each stage as well as the workflow i.e. the relationship between stages.

This handout covers,

(a) Two basic process models that are used as the building blocks for more sophisticated process models:

i) sequential model ii) iterative model

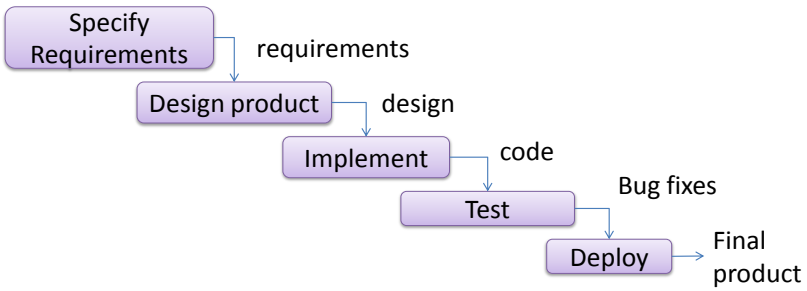
(b) Three popular process models, each a combination of the two basic process models mentioned in (a).

iii) Unified process iv) Extreme programming v) Scrum

(c) A process improvement approach called CMMI

i) Sequential model

The *sequential model*, also called the *waterfall* model, is probably the earliest process model employed. It models software development as a linear process, in which the project is seen as progressing steadily in one direction through the development stages. The name *waterfall* stems from how the model is drawn to look like a waterfall (see below).



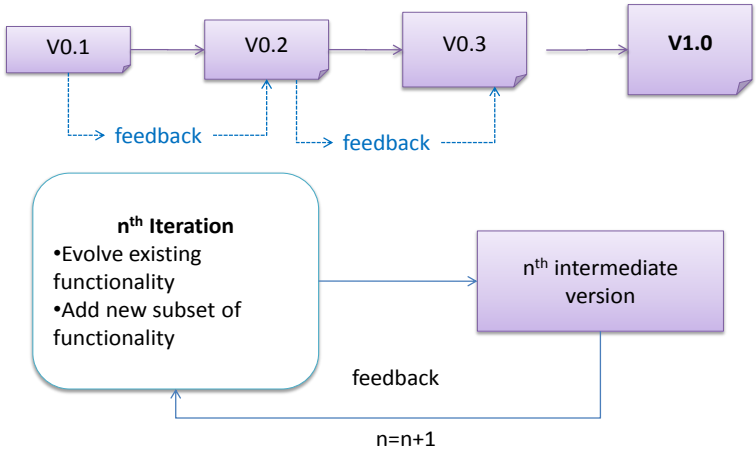
When one stage of the process is completed, it should produce some artifacts to be used in the next stage. For example, upon completion of the requirement stage a comprehensive list of requirements is produced that will see no further modifications. A strict application of the sequential model would require each stage to be completed before starting the next.

This could be a useful model when the problem statement that is well-understood and stable. In such cases, using the sequential model should result in a timely and systematic development effort, provided that all goes well. As each stage has a well-defined outcome, the progress of the project can be tracked with a relative ease.

The major problem with this model is that requirements of a real-world project are rarely well-understood at the beginning and keep changing over time. One reason for this is that users are generally not aware of how a software application can be used without prior experience in using a similar application.

ii) Iterative model

The *iterative model* (sometimes called *iterative* and *incremental*[10]) advocates having several *iterations* of SDLC. Each of the iterations could potentially go through all the development stages, from requirement gathering to testing & deployment. Roughly, it appears to be similar to several cycles of the sequential model.



In this model, each of the iterations produces a new version of the product. Feedback on the version can then be fed to the next iteration. Taking the Minesweeper game as an example, the iterative model will deliver a fully playable version from the early iterations. However, the first iteration will have primitive functionality, for example, a clumsy text based UI, fixed board size, limited randomization etc. These functionalities will then be improved in later releases.

The iterative model can take a *breadth-first* or a *depth-first* approach to iteration planning.

- breadth-first: an iteration evolves all major components in parallel.
- depth-first: an iteration focuses on fleshing out only some components.

The breadth-first approach is considered purely 'iterative', while depth-first is 'incremental'. As mentioned before, most project use a mixture of breadth-first and depth-first iterations. Hence, the common phrase 'an iterative and incremental process'.

In summary,

- The sequential model organizes the project based on activity.
- The iterative and incremental model organizes the project based on functionality.

Most process models used today, including the ones described below, are hybrids of the sequential and iterative models.

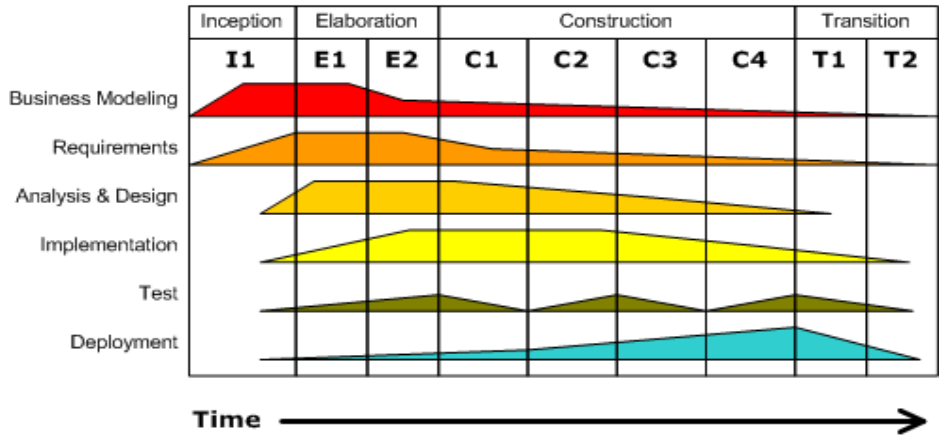
[10] While some authors define *iterative* and *incremental* as two distinct models, the definitions of each vary across authors. In this handout, we define them together as they are most often used together.

iii) Unified process

The *unified process* is developed by the *Three Amigos* - Ivar Jacobson, Grady Booch and James Rumbaugh (the creators of UML).

The unified process consists of four phases: *inception, elaboration, construction* and *transition.* The main purpose of each phase can be summarized as follows:

Phase	Activities	Typical Artifacts
Inception	• Understand the problem and requirements • Communicate with customer • Plan the development effort	• Basic use case model • Rough project plan • Project vision and scope
Elaboration	• Refines and expands requirements • Determine a high-level design e.g. system architecture	• System architecture • Various design models • Prototype
Construction	• Major implementation effort to support the use cases identified • Design models are refined and fleshed out • Testing of all levels are carried out • Multiple releases of the system	• Test cases of all levels • System release
Transition	• Ready the system for actual production use • Familiarize end users with the system	• Final system release • Instruction manual



Given above is a visualization of a project done using the Unified process (source: Wikipedia). As the diagram shows, a phase can consist of several iterations. Each vertical column (labeled "I1" "E1", "E2", "C1", etc.) represents a single iteration. Each of the iterations consists of a set of 'workflows' such as 'Business modeling', 'Requirements', 'Analysis & Design' etc. The shaded region indicates the amount of resource and effort spent on a particular workflow in a particular iteration.

Note that the unified process is a flexible and customizable process model framework rather than a single fixed process. For example, the number of iterations in each phase, definition of workflows, and the intensity of a given workflow in a given iteration can be adjusted according to the nature of the project. Take the *Construction Phase*, to develope a simple system, one or two iterations would be sufficient. For a more complicated system, multiple iterations will be more helpful. Therefore, the diagram above simply

records a particular application of the UP rather than prescribe how the UP is to be applied. However, this record can be refined and reused for similar future projects.

In 2001, a group of prominent software engineering practitioners met and brainstormed for an alternative to documentation-driven, heavyweight software development processes that were used in most large projects at the time. This resulted in something called the *agile manifesto* (a vision statement of what they were looking to do). The agile manifesto (taken from http://agilemanifesto.org/) is given below.

> We are uncovering better ways of developing software by doing it and helping others do it.
> Through this work we have come to value:
>
> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

Subsequently, some of the signatories of the manifesto went on to create process models that try to follow it. These processes are collectively called *agile processes*. Some of the key features of agile approaches are:

- Requirements are prioritized based on the needs of the user, are clarified regularly (at times almost on a daily basis) with the entire project team, and are factored into the development schedule as appropriate.
- Instead of doing a very elaborate and detailed design and a project plan for the whole project, the team works based on a rough project plan and a high level design that evolves as the project goes on.
- Strong emphasis on complete transparency and responsibility sharing among the team members. The team is responsible together for the delivery of the product. Team members are accountable, and regularly and openly share progress with each other and with the user.

There are a number of agile processes in the development world today. eXtreme Programming (XP) and Scrum are two of the well-known ones.

iv) eXtreme programming (XP)

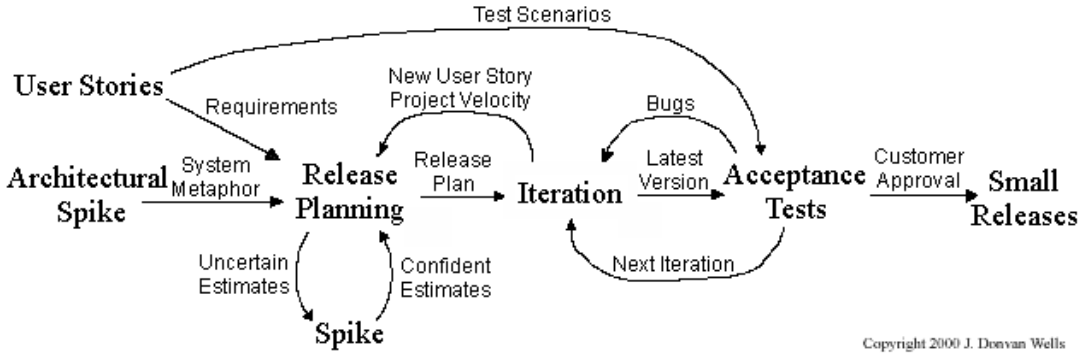
The following description was adapted from http://www.extremeprogramming.org

The first Extreme Programming project was started March 6, 1996. Extreme Programming is one of several popular Agile Processes. Extreme Programming (XP) stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future, this process delivers the software you need as you need it. XP aims to empower developers to confidently respond to changing customer requirements, even late in the life cycle.

XP emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. XP implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

XP aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation, Extreme Programmers are able to courageously respond to changing requirements and technology.

XP has a set of simple rules. XP is a lot like a jig saw puzzle with many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This flow chart shows how Extreme Programming's rules work together.



Pair programming, *CRC cards*, *project velocity*, and *standup meetings* are some interesting topics related to XP. Refer to extremeprogramming.org to find out more about XP.

v) Scrum

This description of Scrum was adapted from Wikipedia [retrieved on 18/10/2011]:

Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:

- The *ScrumMaster*, who maintains the processes (typically in lieu of a project manager)
- The *Product Owner*, who represents the stakeholders and the business
- The *Team*, a cross-functional group who do the actual analysis, design, implementation, testing, etc.

A Scrum project is divided into iterations called *Sprints*. A sprint is the basic unit of development in Scrum. Sprints tend to last between one week and one month, and are a *timeboxed* (i.e. restricted to a specific duration) effort of a constant length.

Each sprint is preceded by a planning meeting, where the tasks for the sprint are identified and an estimated commitment for the sprint goal is made, and followed by a review or retrospective meeting, where the progress is reviewed and lessons for the next sprint are identified.

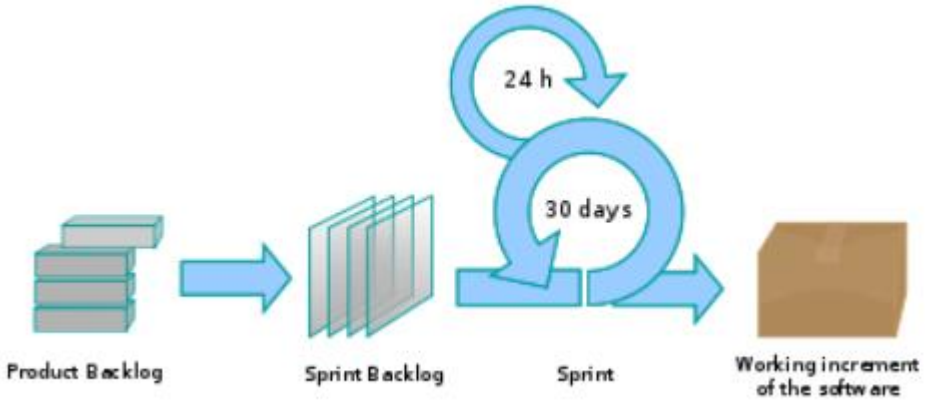
During each sprint, the team creates a potentially deliverable product increment (for example, working and tested software). The set of features that go into a sprint come from the product *backlog*, which is a prioritized set of high level requirements of work to be done. Which backlog items go into the sprint is determined during the sprint

planning meeting. During this meeting, the Product Owner informs the team of the items in the product backlog that he or she wants completed. The team then determines how much of this they can commit to complete during the next sprint, and records this in the sprint backlog. During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Development is timeboxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog. After a sprint is completed, the team demonstrates the use of the software.

Scrum enables the creation of self-organizing teams by encouraging co-location of all team members, and verbal communication between all team members and disciplines in the project.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.

Like other agile development methodologies, Scrum can be implemented through a wide range of tools. Many companies use universal software tools, such as spreadsheets to build and maintain artifacts such as the sprint backlog. There are also open-source and proprietary software packages dedicated to management of products under the Scrum process. Other organizations implement Scrum without the use of any software tools, and maintain their artifacts in hard-copy forms such as paper, whiteboards, and sticky notes. The diagram below illustrates the essence of the Scrum process.

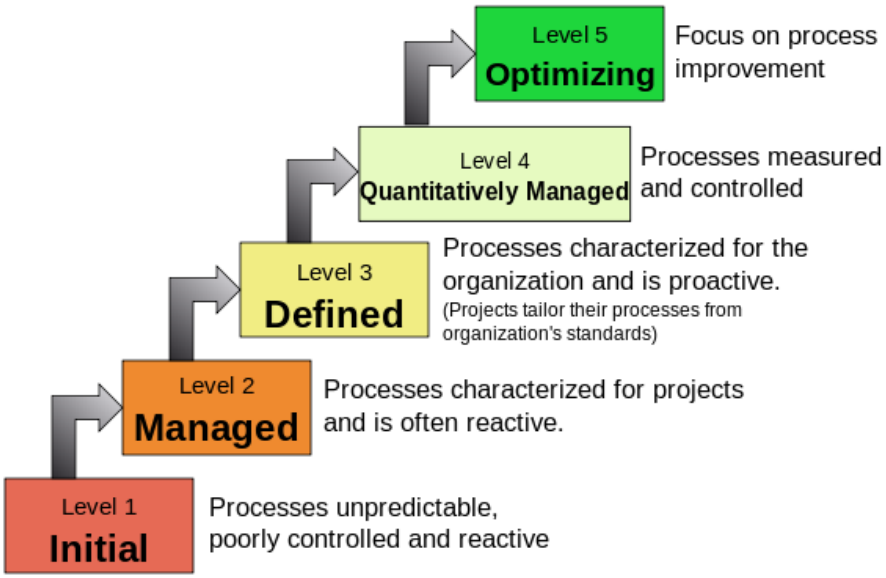


CMMI (Capability Maturity Model Integration)

The following description was adapted from http://www.sei.cmu.edu/cmmi/

CMMI (Capability Maturity Model Integration) is a process improvement approach defined by Software Engineering Institute at Carnegie Melon University. CMMI provides organizations with the essential elements of effective processes, which will improve their performance. CMMI-based process improvement includes identifying an organization's process strengths and weaknesses and making process changes to turn weaknesses into strengths.

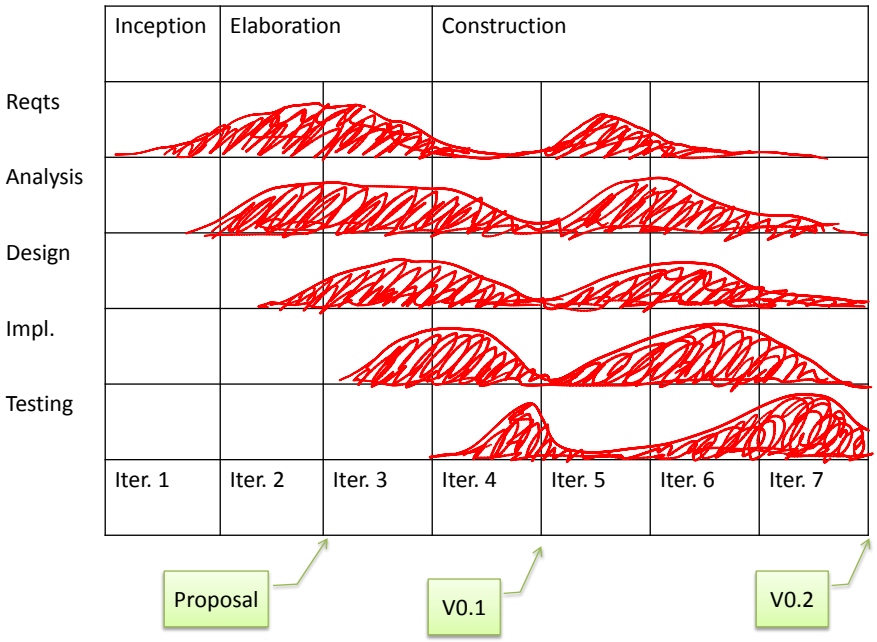
CMMI defines five maturity levels for a process and provides criteria to determine if the process of an organization is at a certain maturity level. The diagram below [taken from Wikipedia] gives an overview of the five levels.



Worked examples

[Q1]
Explain the process you followed in your project using the unified process framework. You may also include smaller iterations that you followed within the two iterations we specified.

[A1]
Sample only. Will be different from project to project. We assume that you are building a Minesweeper game as your project.



Iter 1: Project kick-off. We decide to do Minesweeper

Iter2: Produce the proposal

Iter 3: Modify proposal on feedback, construct a minimal V0.0

Iter 4: Construct version V0.1

V0.2 is produced in three short iterations (5,6,7).

[Q2]
Discuss how sequential approach and the iterative approach can affect the following aspects of a project.

a) Quality of the final product.
b) Risk of overshooting the deadline.
c) Total project cost.
d) Customer satisfaction.
e) Monitoring the project progress.
f) Suitability for CS2103 project

[A2]
a) Quality of the final product:

- Iterative: Frequent reworking can deteriorate the design. Frequent refactoring should be used to prevent this. Frequent customer feedback can help to improve the quality (i.e. quality as seen by the customer).

- Sequential: Final quality depends on the quality of each phase. Any quality problem in any phase could result in a low quality product.

b) Risk of overshooting the deadline.

- Iterative: Less risk. If the last iteration got delayed, we can always deliver the previous version. However, this does not guarantee that all features promised at the beginning will be delivered on the deadline.
- Sequential: High risk. Any delay in any phase can result in overshooting the deadline with nothing to deliver.

c) Total project cost.

- Iterative: We can always stop before the project budget is exceeded. However, this does not guarantee that all features promised at the beginning will be delivered under the estimated cost. (The sequential model requires us to carry on even if the budget is exceeded because there is no intermediate version to fall back on).
  Iterative reworking of existing artifacts could add to the cost. However, this is "cheaper" than finding at the end that we built the wrong product.

d) Customer satisfaction

- Iterative: Customer gets many opportunities to guide the product in the direction he wants. Customer gets to change requirements even in the middle of the product. Both these can increase the probability of customer satisfaction.
- Sequential: Customer satisfaction is guaranteed only if the product was delivered as promised and if the initial requirements proved to be accurate. However, the customer is not required to do the extra work of giving frequent feedback during the project.

e) Monitoring project progress

- Iterative: Hard to measure progress against a plan, as the plan itself keeps changing.

- Sequential: Easier to measure progress against the plan, although this does not ensure eventual success.

f) Suitability for CS2103 project:

Reasons to use iterative:

- Requirements are not fixed.
- Overshooting the deadline is not an option.
- Gives a chance to learn lessons from one iteration and apply them in the next.

Sequential:

- Can save time because we minimize rework.

[Q3]
Find out more about the following three topics and give at least three arguments for and three arguments against each.

(a) Agile processes, (b) Pair programming, (c) Test-driven development

[A3]

(a) Arguments in favor of *agile processes*:
- More focus on customer satisfaction.
- Less chance of building the wrong product (because of frequent customer feedback).
- Less resource wasted on bureaucracy, over-documenting, contract negotiations.

Arguments against agile processes (not necessarily true):
- It is 'just hacking'. Not very systematic. No discipline.
- It is hard to know in advance the exact final product.
- It does not give enough attention to documentation.
- Lack of management control (gives too much freedom to developers)

(b) Arguments in favor of *pair programming*:
- It could produce better quality code.
- It is good to have more than one person know about any piece of code.
- It is a way to learn from each other.
- It can be used to train new programmers.
- Better discipline and better time management (e.g. less likely to play Farmville while working).
- Better morale due to more interactions with co-workers.

Arguments against pair programming:
- Increase in total man hours required
- Personality clashes between pair-members
- Workspaces need to be adapted to suit two developers working at one computer.
- If pairs are rotated, one needs to know more parts of the system than in solo programming

(c) Arguments in favor of *TDD*:
- Testing will not be neglected due to time pressure (because it is done first).

- Forces the developer to think about what the component should be before jumping into implementing it.
- Optimizes programmer effort (i.e. if all tests pass, there is no need to add any more functionality).
- Forces us to automate all tests.

Arguments against TDD (not necessarily true):

- Since tests can be seen as 'executable specifications', programmers tend to neglect others forms of documentation.
- Promotes 'trial-and-error' coding instead of making programmers think through their algorithms (i.e. 'just keep hacking until all tests pass').
- Gives a false sense of security. (what if you forgot to test certain scenarios?)
- Not intuitive. Some programmer might resist adopting TDD.

--- End of handout ---