12) Software development life cycle models and methodologies

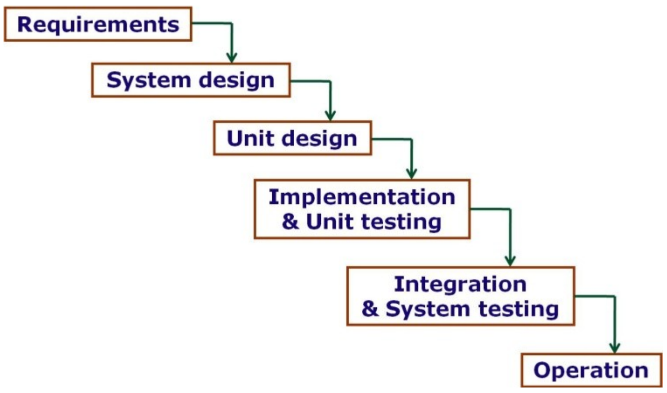
General remarks

As mentioned in chapter 3 ("The life cycle of a project"), there are many models that describe the **sequence of phases in a software development project**. Each particular model emphasizes certain features of a project's organization and dynamics.

The choice of a **model**, which has a definite **impact on project planning**, is dependent on factors and constraints specific to each project, for example the degree of flexibility allowed as regards the requirements specification.
An experienced PM will generally devise a specific model in order to take into account the characteristics of his particular project. Such a "home-made" model may be a variant of a "standard" model. A few standard models are described below.

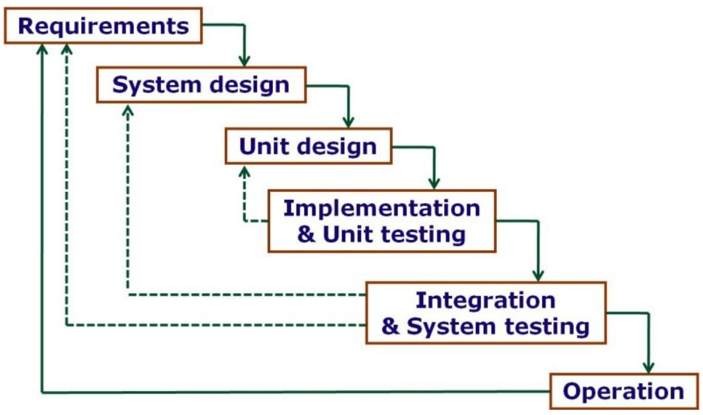
Waterfall model

The Waterfall model emphasizes the fact that each phase of a project must be completed, and its deliverables reviewed and validated, before the next phase can begin. For example, this model excludes going back and forth between product design and requirements, so there is **no flexibility**, which can be a major drawback for certain projects.



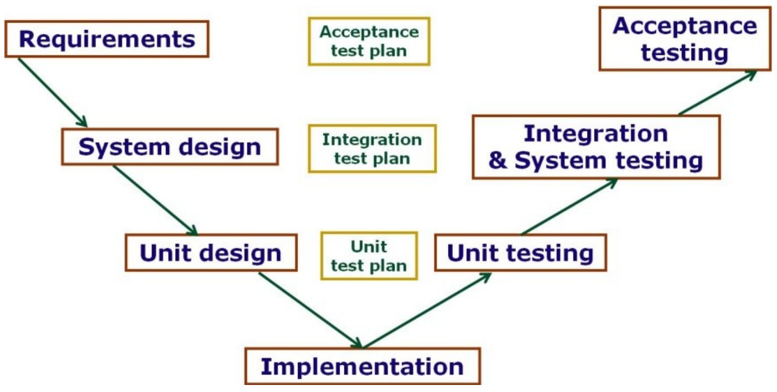
Incremental (or multi-waterfall) model

The Incremental model, which involves a repeated application of the Waterfall model, and therefore provides **more flexibility**, may be used for development projects that are executed in **multiple iterations**. Each iteration delivers a working version of the software (to be tested against specifications), which thus gradually evolves by increments into the final product, with limited risk of deviation from the requirements.
One advantage of this model is that software is delivered for testing in stages, as opposed to being delivered "in bulk" at the end of the development cycle.



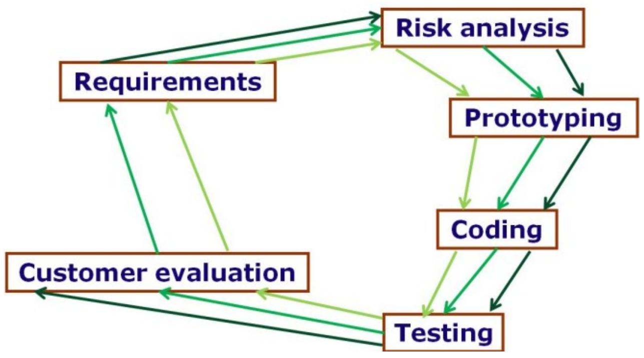
V (or V-shaped) model

The V model **emphasizes testing**: for each level of product specification, a test plan and test cases are prepared before implementation is started. Testing is performed at each level according to the test plan. The tests must first confirm that the "units" (or "modules") that have been developed comply with the unit design specifications. Compliance of the "system" resulting from the integration of all units, and content, if any, with the system design specifications must then be tested. Finally, compliance of the complete product with the requirements specification must be tested.
As with the Waterfall model, there is, in principle, no going back and forth between requirements and design, therefore **no flexibility**, which can be a major drawback for certain projects.
A **"double-V" (VV) model** may be applied to a project involving the development of a prototype then of a complete product; in this case the V model is applied successively to each development cycle.



Spiral model

The Spiral model, which is comparable to the incremental model, is an **iterative process that emphasizes risk analysis and resolution**. Risks (of a technical nature) are evaluated at an early stage and minimized as **multiple prototypes** are produced then tested and evaluated by the customer. The requirements specification may thus be adjusted to take into account what is technically feasible (or not). In this respect, the spiral model allows **more flexibility** than the three models described above.
However, the more "revolutions" in the spiral, the higher the cost of the project!



> See the following sites for **more information on Project life cycle models**:

>> codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/

>> en.wikipedia.org/wiki/Waterfall_model

>> en.wikipedia.org/wiki/V_model

The above-mentioned models, which are useful for describing the successive phases of a project, feature various degrees of flexibility but are globally somewhat rigid. There are **other methodologies for software development projects** which allow **more flexibility and interactivity**, and which you, as a PM, should be aware of.
These methodologies need not be applied in a strict manner, but their general principles should be kept in mind. Some of the principles are actually featured to a certain extent in the models presented above, and many of these principles are just "**common sense**".

One particularly interesting methodology is "**Agile software development**", and "**eXtreme Programming**" (**XP**) is one of its applications. They are described hereafter.

Other methodologies (referenced by links at the end of this chapter) include "**Scrum**" (another application of "agile" principles) and "**CMMI**" (Capability Maturity Model Integration).

Agile software development

This methodology is based on **ten key principles**, listed and described below.

1. Actively involve users.
2. Empower the development team to make decisions.
3. Allow requirements to evolve but keep the project's timescale fixed.
4. Capture requirements at the highest level of description.
5. Develop small incremental releases, and iterate.
6. Make frequent delivery of product (to test…).
7. Complete a feature before moving on to the next.
8. Apply the "80/20 rule".
9. Integrate testing throughout the project's life cycle.
10. Rely on a collaborative approach between the project's stakeholders.

1. Actively involve users: this may seem obvious, but it is not always easy or even possible to involve users in all stages of a project, in particular during development. The more opportunities there are for users to test the product before its completion and to provide feedback, the better.

2. Empower the development team to make decisions: giving development team members a certain degree of decision-making authority, along with responsibility, contributes to motivation and efficiency, which is positive. A prerequisite is the involvement of the team at the early stages of the project, in particular the review and analysis of the requirements specification.
In most cases however, there will need to be a decision-making hierarchy within the development team so that if a consensus cannot be reached a decision can nevertheless be made.

3. Allow requirements to evolve but keep the project's timescale fixed: this means that flexibility is allowed with respect to the initial requirements specification. The specification may be adjusted as the project moves forward, in full agreement with the "client", in order to take into account changes that implementation has revealed to be necessary.
The timescale, however, should not be modified (unless the client agrees to such a change…), so that the final deliverable arrives on schedule (and within budget).

4. Capture requirements at the highest level of description: this means that requirements should not be specified in great detail (just enough to provide a general understanding of what they mean) at the beginning of a project; they will be refined at a later stage, after a number of iterations, each iteration resulting in deliverables that can be tested then possibly adjusted to match any requirements changes that have been decided and agreed upon.

5. Develop small incremental releases, and iterate: this is a very practical approach to development with iterations/increments ("analyze; develop; test"), one feature at a time (broadly speaking), thus limiting the risk of developing a product that does not meet requirements.

6. Make frequent delivery of product (to test…): this principle, which is included to a certain extent in the previous one, promotes a "perpetual beta" approach, so to speak, and is particularly applicable to website development, in which the first version of a site can be "open for business" within a few weeks after the start of the project, then can be adjusted and enhanced as required, integrating user/market feedback on a given version into the next version.

7. Complete a feature before moving on to the next: this principle emphasizes the fact that having developed a feature does not mean that it is complete; it needs to be fully tested and accepted before it can be considered "done"!
Of course, in some development projects, multiple features may be developed in parallel, in which case the principle applies to each "batch" of features.

8. Apply the "80/20 rule": this is an instance of Pareto's principle ("For many events, 80% of the effects come from 20% of the causes"), which applies to many areas of activity. In the area of software development, it may be interpreted as "80% of results (may) come from 20% of the development effort", which is often true. The general message is "focus your efforts on what is really important and provides maximum leverage!".

9. Integrate testing throughout the project's life cycle: this principle is consistent with at least two of the previous ones, namely "Develop small incremental releases, and iterate" and "Complete each feature before moving on to the next". The main objective here is product quality.

10. Rely on a collaborative approach between the project's stakeholders: this principle is actually a "must", given some of the above principles. Indeed, evolving requirements, iterative development, testing throughout the project cycle, etc. cannot be achieved without close cooperation between the development team and the other stakeholders (project owner, business analyst, user representatives...).

> See the following sites for **more information about Agile Software Development**:

>> agile-software-development.com

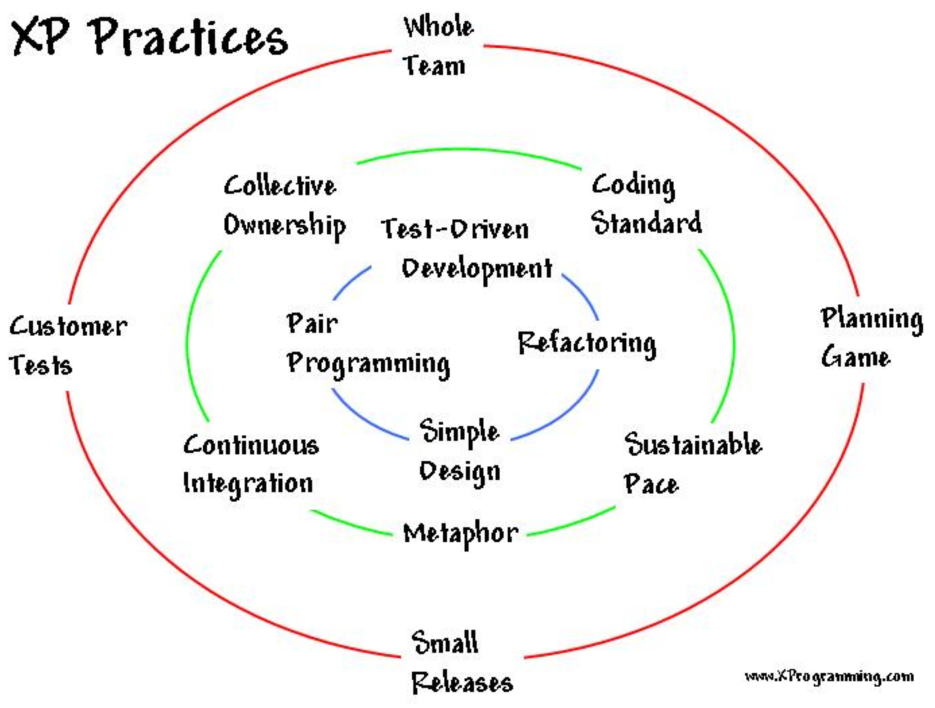
>> https://www.agilealliance.org/

eXtreme Programming (XP)

This methodology is a good example of application of Agile Software Development basic principles. It is summarized by the following quotation and diagram from a website referenced at the end of this chapter.

- ➢ XP improves a software project in four essential ways: communication, simplicity, feedback and courage.
- ➢ XP programmers 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.
- ➢ They deliver the system to the customers as early as possible and implement changes as suggested.
- ➢ With this foundation, XP programmers are able to courageously respond to changing requirements and technology.

Note: in the last sentence above, the adverb "courageously" is used as a synonym of "fearlessly" or, by extension, "aggressively".



The keywords in the diagram represent the general principles of XP, which are briefly explained below.

Whole team: all contributors to the project form a single team, including at least one business and/or user representative.

Planning game: the emphasis is on steering the project rather than on exactly predicting what needs to be done and how long it will take. This involves two types of planning:

- Release planning: provides due dates for deliverables that are supposed to meet the requirements specification; the release plan may be revised by the "whole team", as necessary.
- Iteration planning: the direction given to the team is adjusted on a regular basis (for example every couple of weeks).

Simple design: software design is kept as simple as possible, for the initial release as well as for the following more complete releases, in order to always match the required functionality of the "system" being built (as opposed to wasting time on features that need not be included).

Metaphor: the development team should imagine and use a metaphor, or a set of metaphors, to describe in very simple and evocative terms how the software should work. The corresponding vocabulary should be agreed upon by the team.

Refactoring is a continuous process of design improvement that focuses on avoiding duplication and achieving full "cohesion" of the code developed, lowering the risk of problems that often arise when "coupling" (or interfacing) software units.

Continuous integration: the system must be kept "fully integrated" at all stages of development in order to maintain its cohesion. System builds are produced on a very frequent basis (several times a day if necessary).

Small releases: a working version of the software is delivered to the customer following each iteration; it is tested by "actual" users or user representatives, and may be accepted and even put into operation in order to gather as much real-life feedback as possible.

Customer tests are performed, as stated above, for each "small release" of the software. Furthermore, it is highly recommended to design and develop automatic acceptance tests that can be run over and over again to make sure there is no regression between releases. Automatic tests may complement manual tests, which are sometimes carried out too hastily.

Test-driven development: unit tests are systematic, with full coverage of the features that have been developed; as the system grows, so does the number of unit tests which need to be run successfully; feedback from the tests drives further development work.

Coding standard: code written by any member of the development team should comply with a general, unique standard, as if it were written by a single programmer. This approach ensures the cohesion of the system and makes code maintenance easier.

Pair programming: each software unit is developed by two programmers working together. The principle here is that "pairing" results in better code than would be produced by two programmers working singly, in about the same time frame, if not faster.

Collective (code) ownership: code produced by (pairs of) programmers should in principle be owned by all members of the development team, who are required to pay attention to code written by other members and to contribute to improving its quality.

Sustainable pace: developers should work with maximum productivity at a pace that can be sustained for long periods of time, thus minimizing the "burn-out" factor.
Note: this is easier said than done!

> See the following sites (from which the above quotation, diagram and explanations have been extracted) for **more information about eXtreme Programming**:

>> ronjeffries.com/xprog/articles/expdocumentationinxp/

>> extremeprogramming.org

> Visit the following sites to **learn about Scrum and CMMI methodologies**:

>> scrumalliance.org

>> cmmiinstitute.com