Software Life-cycles

Peter Wittmann, peter@wittmannclan.com, www.wittmannclan.com

During my time at the university and in my professional career I have come to known quite a number of ways to handle software life cycles. This document is a short overview of some of the most popular approaches and discusses each of these models.

Preface

I wrote this document in order to compare some of the most common software life cycles (SLC). This document is neither considered a complete review nor a summary of all the SLC's used today. For further details please refer to some books on object oriented software design.

It is probably a good idea to open each of the images to the corresponding texts in a second window to support the explanations.

Introduction

Software life cycles (SLC) are meant to present an outline of software development teams can cling to. These life cycles can be distinguished by their way of dealing with the different activities in the evolution of the software as well as their way of handling problems that arise during such a development. There are two types of models: those that go through each of the activities sequentially only once and those that go through each of the phases many times. Those that execute the stages in a life cycle iteratively produce different artefacts (such as prototypes as well as documents) after each iteration. The major benefit is that even if the whole project is cancelled there are pieces left that could be used in further projects or developments.

Please be aware that there is no one best practice for every development process. The following is just supposed to list the most common ways of managing software development processes. Depending on the company structure, complexity of the project, customer requirements and team size as well as knowledge in and usage of tools each of these models has advantages and disadvantages.

Waterfall model

This is one of the sequential activity centered models. All of the software development activities are performed in sequence and there is no iteration. All requirements for one activity are completed and reviewed before the next activity starts. The goal is to never turn back once an activity is completed.

This model is quite easy to understand by each of the parties involved in the process.

However, due to its sequential nature, this model is not capable of dealing with iterations and evolutions. It can't deal with changes and problems that arise during one of the activities since it does not consider a second iteration in one stage. Furthermore, tests aren't mapped to the corresponding activity. Once a phase has been considered complete the results are not going to be changed and are used as input for the next phase. Most of the time, this is

too rigid.

Once a project is cancelled, no prototypes or solutions are present that could be reused since the whole development process has to be aborted once a problem is found that had to be dealt with in an activity before the problem arose. The whole development process is lost and no money can be made out of single components or modules that were already developed.

V-model

This is another sequential activity-centered model, but in contrast to the waterfall model, all of the activities are mapped to tests. Like the activities of the waterfall model, each activity has to be completed before moving on to the next activity. Whereas each of the subsequent activities specify the system in more detail and with less abstraction until it is implemented. This model also deals with the verification of each of these activites through the mapping of tests to the corresponding activities. Mapping tests starting from unit-tests to acceptance tests ensure dependencies between development activities and verification activities.

One advantage is the presence of tests: to show where exactly the system has to be changed or where the system is not fulfilling its desired state.

But in general the V-model is too simple. It does not discuss how to deal with problems. Therefore the tests could only give a clue where the problem might be but the whole process might still need to be terminated. That's also because it doesn't deal with iterations which are almost always needed in the real world.

Sawtooth

The Sawtooth model is actually an extension of the V-model. The only difference between the Sawtooth and the V-model is, that prototypes are created and shown to the client for validation. The prototypes are present right after the analysis phase and in between the design and the implementation phase. By making sure that the client is getting an insight into the progress, ceckpoints should ensure that development is going in the right direction.

The major benefit comes from the validations in between the critical phases. The client is involved which is supposed to guarantee that the project will become a success.

The huge problem is the time consumption and costs associated with presenting the prototypes to the client. Depending on the complexity of the project, the time and costs might be considerable.

Sharktooth

This is another, more detailed view of the Sawtooth model. In contrast to the sawtooth model the sharktooth model puts the manager into account. By presenting the manager a certain abstraction it also introduces new activities.

As it's pretty similar to the Sawtooth model it has all of the advantages and disadvantages of the Sawtooth model.

Evolutionary model

The evolutionary model could be seen as one of the classic iterative activity based models. Each of the activities are handled in sequential order. After each of the iterations an evolutionary prototype is produced. This prototype is used to validate the iteration it's just coming from and forms the basis of the requirements for the next phase. It's usually applied when the requirements are unclear. There are two well-known forms: exploratory programming and prototyping. A prototype is used to discover the requirements and is discarded once the real system is developped. In exploratory programming a system is implemented and functionality is added iteratively.

One of the major problems here is the way of dealing with the development of software. The prototypes produced are being used in the next phase. Because many prototypes are produced that are redifned further and further, the whole software mighjt become a patchwork of features. Hence the whole approach might only work for small projects. Large or complex projects can't be handled this way as the software might become too hard to manage.

Spiral model

This is one of the activity based iterative models. It deals with iterations and changes in activities. All of the waterfall's activities are extended into a cycle. Each cycle consists of four phases. In the first phase determination of objectives, alternatives and constraints happens. Alternatives are evaluated as well as risks being identified and resolved in the second phase. In the third phase the development and verification of the current cycle happens. The last phase is used to plan the next cycle. For each cycle the distance from the center measures its proximity to the final product and cost whereas its angular coordinates show the overall progress.

An advantage of this model is its ability to connect objectives to product developments. Risks management is also considered as well as iterations of tasks. It actually combines both development and evolutionary approaches. But the most important advantage is that there are prototypes even if the whole project has to be cancelled.

Nevertheless, it's quite hard to apply in real world scenarios. This model demands many activities in each cycle and requires quite some knowledge in risk management to consider all of the necessary circumstences. Changes in between the cycles are possible but not within one cycle.

RUP

The Rational Unified Process is by far the most complex model. The RUP looks at a project in terms of cycles. One cycle consists of four phases: inception (i.e. concept exploration), elaboration (i.e. planning and allocating ressources), construction (i.e. implementation) and transition (i.e. installation). In each of these phases many different workflows (like management, environment, design, implementation workflow, etc.) can be addressed simultaneously. At the end of each cycle some kind of prototype or artefact are produced. The phases can be repeated many times (i.e. iterations), producing one or many prototypes or artefacts. In each of the iterations other workflows can be addressed and involved.

During the cycles the requirements are stable which offers possibilites to plan the development process for this cycle. But in between the cycles the requirements change. Adressing many different workflows in each of the cycles and phases it's possible to include all of the participating parties at the right time.

Entitiy-based

The entity-based model is supposed to overcome the issues in a software development process that has to deal with lots of changes during the lifetime of the project which is longer than the actual time in between different changes of technologies or issues. An activity-based model wouldn't be the right choice: by the time the project is done, the requirements have changed due to changes in technology or other issues (user demands, substitute products, etc.). Therefore an entity-based model is used. As the name suggests it consists of entities and issues. Those can be closed and re-opened at any time. Once an issues is considered done it's not unlikely that it could be re-opened again. The entity-based model addresses these issues is by dealing with them simultaneously and not in any special predefined order.

XP

eXtreme Programming - I guess most people have heard of this already as it's been relaitvely new and most controversially discussed. However, it might also be the one closest to real life scenarios.

There are only four activities: coding, testing, designing and communicating. All of these activities are performed by each of the programmers involved in the project. No documentation is produced as the code represents all design decisions and thus all the knowledge. Pair programming is one of the key concepts in XP: two programmers sit in front of the screen and share both knowledge and decisions. While one of them is writing code the other one is keeping an eye on design errors or implementation errors. This way small parts of the system are developed, errors are supposed to be small and there is another person at hand who knows how to move on when one gets ill. The customer is always ready to hand, so that if requirements problems or other problems arise there's always someone to ask.

Together with the manager the customer divides the program into sub-projects that can be dealt with in a certain amount of time (normally one week). This way the customer has the possibility to control the development, costs and can change requirements in between each of the phases. By letting the customer write acceptance tests first, the programmers understand what their goal is, what their program is supposed to do. But the programmers write tests, too, to see if smaller components of the system act the way they're supposed to. After each week the programmers and the manager meet in order to discuss the status and the achievements of the developers. In those meetings the other members are only supposed to listen. After those meetings the members could discuss certain things or have a look at the code.

One of the main advantages is that the customer is involved during the whole development process. And by dividing the project into smaller projects the customer is even able to redefine goals in between the phases. The programmers concentrate on programming and deal with what they're supposed to deal with most of the time: the code. Repeated and also regression tests are used to guarantee quality code.

The drawbacks are obvious: as all of the programmers are dealing with the complete code and all of them are responsible for the whole code, it's hard to catch up at large projects. Empirical studies have shown that XP only scales up to 10 programmers. Everything above that creates an immense amount of work for each of the developers.

Summary

As mentioned before: there is neither a worst nor best model. And there is also no generally good or bad approach to managing software development. Each model has its own advantages and disadvantages. And each model is meant to deal with (sometimes only

slightly) different aspects than the other. Which of these models should be applied to a certain problem depends on the number of persons involved, the complexity of the software to be developed and the goals and even more aspects. The above introduction might already help evaluating the correct choices for each case. Most of the time the model is chosen indirectly by the way the software is supposed to be delivered and developed. If you really have the chance to chose a model for a future project, it might be best to not only think about the software lifecycle at hand but also about the tools, frameworks, and knowledge in need to support this kind of development.

Links