Collaborative Software Engineering of the Life-Critical Systems

DENISS KUMLANDER
Department of Informatics
Tallinn University of Technology
Raja St.15, 12617 Tallinn
ESTONIA
http://www.kumlander.eu

Abstract: - A collaborative software engineering approach for the life-critical systems is proposed in this paper. Life-critical systems are systems, where any fault can produce a danger for humans or can be extremely expensive. The life-critical systems' development is well known as expensive and slow one. Mostly it is because of its nature – fault-free development. The approach to be proposed includes two main elements: an effective feedback between neighbour steps of the development work cycle and a collaborative work team either global for the entire project or within neighbour steps. In the result, a development team can increase their productivity keeping products on the same quality level by eliminating certain extra-time spent on local verification by using next phases to support such evaluation.

Key-Words: - Collaborative software engineering, and life-critical systems

1 Introduction

Life critical software systems are software packages that should work correctly to ensure safety of people, in other words each failure of such systems could be dangerous for people lives. One example of such system could be software controlling different submarine systems. If any element will fail then the submarine crew will be in a sufficient danger. Sometimes life critical systems are also extended to systems controlling other important or extremely expensive mechanisms and equipment because loosing those systems due software failures is also too expensive. An example of such equipment could be a satellite and its software could be a launch system's module. A historical example of software failure in such systems could be the Mariner I space probe, which was launched in 1962 and was destroyed during several minutes after its start since an error occurred in the controlling software. All these demonstrates that software engineering have certain restrictions on methodologies to be used since should guarantee the highest quality unlike commercial software development, where such high quality is compromised by the end product price.

On the other hand methodologies used so far in the life-critical systems development can be defined as "heavy-weight"-ed as requires quite a lot of additional work and are not flexible enough. It results in many cases in very ugly software with an enormous cost. Following the main principle of nowadays software development looking for approaches ensuring the simpler and more flexible development cycle and increasing quality of the resultant software we could define an increasing demand for life-critical software engineering improvements that will not compromise software end quality. Here a collaborative software engineering principle is proposed to be applied, which is derived from a commercial software development principle called supporting software engineering that was described in series of our previous works.

The paper is organised as follows. The section 2 briefly describes the software development work cycles of different high-level approaches. This section describes an approach to which the collaborative principle will be proposed and an approach from which certain parts of the supporting design will be inherited. The following section introduces the supporting software engineering principles and demonstrates how those are transformed into the collaborative software engineering for the life-critical systems. The forth section lists some potential danger that a software engineers should be aware of implementing the proposed approach and gives some guidelines on avoiding those. The last section concludes the paper.

2 Software Engineering Approaches

Software engineering approaches can be divided into two major categories:

- Classical or traditional;
- Modern or iterational.

Below we are going to demonstrate the main difference by building a general model from which this difference can be derived.

There are a lot of models for software development and some of them are quite basic ones like the waterfall or spiral software development [1, 2, 3, 4] methodologies. Therefore it is a bit hard to build a model

that could adequately reflect all those on a general level, but the following very simple one should demonstrate basic principles from most of them:

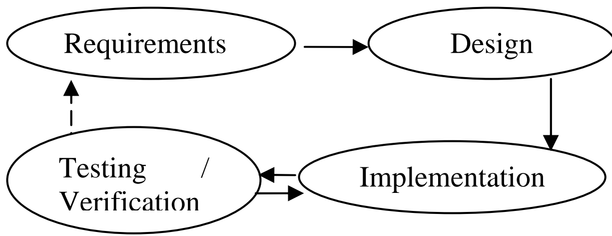

Fig. 1 High-level software development work cycle

Notice here the dashed arrow that denotes difference between main approaches. The traditional one says that all steps should be done completely before moving to another one so there is no cycle during one complete software implementation, although different versions could still exist, but each version should be ready to use on 100% and a time interval between different versions normally is sufficient [1]. Therefore the last dashed arrow is missed in this approach. Notice also that the testing and verification step, which is the last one and normally interact with the development step returning errors and getting new releases and fixes. Now let's examine the second main approach. The modern iterative software implementation and engineering develops a complete package in several cycles [1, 5], so the transition between the testing phase and requirements formulation exists in this type approaches and therefore is followed several times. It means that the dashed arrow becomes "visible" and is followed several times as software implementation contains several cycles.

It is widely adopted that life critical system's software engineering follows the traditional approach since there is no possibility to do and test intermediate releases as a potential loss is too big. At the same time the impossibility to do a cycle and fix mis-designs and other errors produces a need to spent much more time verifying each step than normally it will be needed.

3 Collaborative Software Engineering

A central idea of the supporting software engineering approach of the iterative software development is establishing an efficient feedback inside a software development cycle between neighbour steps into addition to the global feedback that is an iteration / cycle by itself [6]. The feedback is provided from the next step to the previous one in case any error is done on the previous step. The "efficient" term means that it is not enough to establish such feedback, but it should be guaranteed to be working using different approaches, rules and an infrastructure. The efficient feedback's goal is to provide information and correct mistakes faster than using the global feedback, i.e. inside a cycle instead of waiting until the next cycle will start and the previous step as a part the next style will be activated.

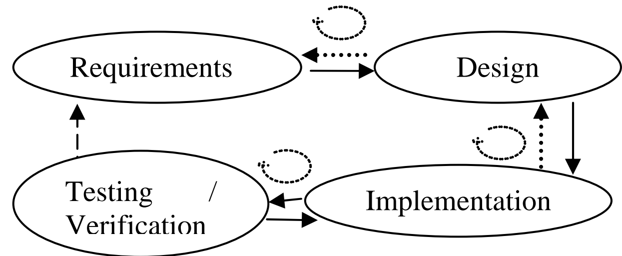

Fig. 2 High-level software development work cycle with concurrent feedback connections

The figure 2 demonstrates changes done by the new approach in the software development work cycle in compare to the earlier version presented on the figure 1. The dotted arrows represent a feedback cycle, which is concurrent internal cycle as it is denoted by the curved arrows. This feedback allows to move back as many times as it is needed, but the number of internal iterations is selected by each team / company separately to avoid staying on the same step / version in the development progress schedule.

A work cycle for the life-critical systems, basing on the previously described restrictions and properties would look like the one on the figure 3.

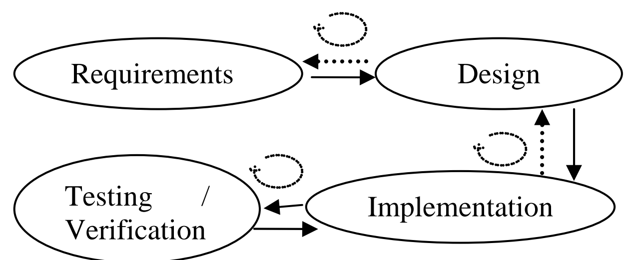

Fig. 3 Collaborative software engineering's work cycle for the life-critical systems

There is no transition from the last work cycle stage to the first of a new work cycle as a product is produced in one cycle. Therefore internal feedbacks and cycles become much more important than in iterative approaches and are the main proposition of the paper. So far one cycle software engineering required some extra time to verify the step output before moving to the next stage. Despite of that errors, missed information, un-designed and un-specified parts where still existing. The only transition where the feedback was established long time ago and was always working was between testing

and verifying and software development phases. The remaining steps was "independent" from the feedback point of view and therefore many software engineers had to solve the described earlier problems arising on their stages either by ignoring those or having some kind of self-made fixes basing on the previous stage documents logic. Notice that such approach could be dangerous in many cases. The other way was to cancel the step fully and re-run the project from scratch (in case of finding sufficient problems) although each stage team could still re-use their previous work the next time. All this resulted in sufficient time losses and decreased usability of the system (both human and computer interfaces). The proposed feedbacks' system enables:

- Return to a previous step (at least partly) and correct mistakes. This could include raising an event (error request, question etc) to more than one level, for example up-to to the first stage. Notice that initial requirements gathering can be greatly verified during the implementation stage done on a lower level on some aspects (notice that only some as other techniques [7] have to be applied to ensure overall success of the requirements formulation step from elements that can not be developed verified). Therefore the efficient feedback leads to a new opportunity within a project: using next steps to verify correctness on previous, i.e. feedback routines are used to ensure correctness of a product via all work cycle stages. All this will stabilize the development and design process;

- Improve or establish collaboration between different teams [8] that normally results in a higher performance of outputs as different teams see topics from different points of view and therefore have better overview of advantages, disadvantages, opportunities and problems. An extra effect of the increased collaboration is a smother and faster work flow as different phases teams are starting to formulate one big project team of persons willing to work together in a comprehensive manner.

4 Potential Dangers of the Approach and Ways to avoid those

Any more or less complex approach has its own potential dangers that have to be foreseen and evaluated to ensure that the approach can be effectively applied. The collaborative software engineering of the life-critical systems isn't an exclusion from this rule. This chapter describes certain conditions/environment where the approach cannot be applied and dangers that should be closely monitored and avoided.

First of all an organisation applying the collaborative software engineering by establishing feedbacks should ensure that this feedbacks routine is not formal and is really working. The following problems could arise:

- The feedback channel should not be overfilled, otherwise feedbacks can be lost, ignored or will not be posted any more. This situation could be produced by two opposite circumstances:
  - The previous step team is unprofessional and the next team generates a lot of error requests on a permanent base;
  - The next stage team doesn't understand an output generated by the previous team adequately and starts to generate faulty (incorrect) error requests;

- The feedback channel is not used or is ignored. It could happen because nobody is informed about the feedbacks possibility. Another case: nobody believes in any reaction of a feedback messages and therefore nobody posting messages. This could happen because of a pure communication and attitude between teams. The collaborative work is possible only in an innovating supporting organisations with well-established communication channels, personnel that is willing to cooperate and is informed on those possibilities;

- The collaborative work and the feedbacks system are not supported by a required infrastructure. An organisation should ensure that the approach processes are supported by dedicated services ensuring that none of the following is true:
  - It is hard to publish a feedback; usability of the feedback publishing/ tracking system is low;
  - Feedbacks can be lost in the system or are not associated to any person;
  - There is no possibilities to track feedbacks and get an efficient information on its status, activities and persons who either are dealing or worked with each actual feedback post;

- Communication gaps: An efficient feedback impossible if information is corrupted during the communication process. It can be or not related to the communication. The first case is already described, so here communicated persons problems will be discussed. The corruption of information can occur because of inequality in knowledge, experience, background etc of the involved persons (senders, receivers, and messengers). It can be produced by impossibility to provide full information communicating by phones (loss of visual information), slow or bad lines including internet communication forcing to compact messages. The

most common scenario of this case is a distributed organisation with branches forced to communicate over long (extra long) distances [9, 10]. The following methods could be applied to avoid described problems:

- o Communication channel is supported by a dedicated infrastructure;
- o Teams that are communicating mostly locates as close to each other as possible;
- o Feedback are verified by a proper documentation routines (avoiding unnecessary bureaucracy);
- o Good timing for the communication and "right" persons that are able send/receive information to be transmitted;
- o Additional methods improving cross-team collaboration like informal contacts etc.

- Security restrictions: this is a problem that is specific for the life-critical systems' development environment especially in the military sector. There can be some complex security rules and restrictions dictating communication rules and potentially disabling moving of information back, collaborating of teams from different phases' and so forth. Those restrictions are a "physical" barrier on the feedback communication flow, which can not be avoided and collaborative software engineering cannot be used in such organisations.

5 Conclusion

A collaborative software engineering approach for the life-critical systems was proposed in this paper. This approach includes two main elements: an effective feedback between neighbour steps of the development work cycle and a collaborative work team either global for the entire project or within neighbour steps. The life-critical systems' development is well known as expensive and slow one. Mostly it is because of its nature – fault-free development. At the same time the collaborative software engineering can increase productivity of the development team keeping products on the same quality level by eliminating certain extra-time spent on local verification by using next phases to support such evaluation. At the same time the organisation should ensure that the established feedback routine is really effective and working; is supported by the required infrastructure (dedicated services) and is traceable. The approach cannot be used if there is any kind of restrictions on communicating between the development stages due, for example, security reasons.

References:

[1] B.W. Boehm, A spiral model of software development and enhancement, *Computer*, Vol. 21, No. 5, 1988, pp. 61-72.

[2] I. Somerville, R. Jane, An Empirical study of industrial requirements assessment and improvement, *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 1, 2005, pp. 85-117.

[3] O. Forsgren, Churchmanian co-design – basic ideas and application examples, *Advances in Information systems development: bridging the gap between academia and industry,* Springer, 2006, pp. 35-46.

[4] M. Rauterberg, O. Strohm, Work organisation and software development, *Annual Review of Automatic Programming*, Vol. 16, 1992, pp. 121-128.

[5] P.R. Reed, Developing applications with Visual Basic and UML, Addison-Wesley, 1999.

[6] D. Kumlander, Supporting Software Engineering, *WSEAS Transactions on Business and Economics*, Vol. 3, No. 4, 2006, pp. 296-303.

[7] I. Somerville, P. Sawyer, Requirements Engineering – A good Practice Guide, Wiley, 1997.

[8] D. Kumlander, Bridging gaps between requirements, expectations and delivered software in information systems development, *WSEAS Transactions on Computers*, Vol. 5, No. 12, 2006, pp. 2933-2939.

[9] C.D. Cramton, S.S. Weber, Relationship between geographical dispersion, team processes, and effectiveness in software development work teams, *Journal of Business Research*, Vol. 58, No. 6, pp. 758-765

[10] D. Kumlander, *Providing a correct software design in an environment with some set of restrictions in a communication between product managers and designers, Advances in Information systems development: bridging the gap between academia and industry*, Springer, 2006, pp. 181-192.