BRIDGING UNCERTAINTIES GAPS IN SOFTWARE DEVELOPMENT PROJECTS

Deniss Kumlander
Department of Informatics, Tallinn University of Technology, Raja St.15, 12618 Tallinn, Estonia

Keywords: Software engineering, uncertainties.

Abstract: The uncertainties is a well known factor affecting the final result of nearly any software project nowadays. Their negative impact is either a misfit between customers' expectations and released software or extra efforts that software vendors have to invest into the development process. The paper presents some novel approaches to uncertainties handling including an ambassador driven communication, discussion groups and varying length internal cycles with software demonstration meetings.

1 INTRODUCTION

The ultimate goal of software engineering process is to provide customers with tools that will help them to automate their activities or achieve desired goals. The modern software development faces new challenges as customers demand much higher quality of the released software, shorter development cycle and increased flexibility of defining requirements. The flexibility requirement and quickly changing business environment produces a lot of uncertainties for software implementation as it becomes very hard to match expectations and the released software after several months of development. This huge pressure on software vendors produces a relatively high level of software projects fails. Researches show that up to 27% of all projects fail because customers are not satisfied with the delivered software (Bennatan and Emam, 2005) and a lot of other projects fail since those do not fit into budgets. Sometimes it happens since those projects are having difficulties with meeting customers' requirements during final stages and are rebuilding the software again and again. A situation with the actual percentage of functionality in use doesn't look much better: just 20% of functionality in average is used "often" or "always" and 16% "sometimes". The remaining 64% is either never used or used just occasionally (Khan, 2004). Therefore the uncertainties management becomes very important in order to ensure software engineering projects success (Kumlander, 2006a).

2 UNCERTAINTIES IN SOFTWARE PROJECTS

Uncertainties in modern software development projects are well-known elements although the name could vary depending on whether the cause of problems or consequences are discussed. Those can be produced by external world – business environment, wrong initial propositions of customers formulating requirements, or by the internal project environment – requirements that are faulty, unclear, corrupted during implementation by missing important details and so forth. Under uncertainties in this paper we mean the lack of certainty about all kind of requirements to software to be developed by its release date and software properties (like technology, included functionality and so forth). Notice that the software in result target expectation arising by the end date of the project rather than existing at the moment. The uncertainty is not the same as the risk since here we deal with a lack of information (certainty). Of course the negative uncertainties' impact is usually measured and treated as a risk, but it will be just one particular case. Therefore the "risk" term is not sufficient to describe the "uncertainty" term and therefore should not replace it.

There are two major approaches to the uncertainties handling in software development projects nowadays. The first approach is to ignore possible problems and uncertainties until those occur. This approach is based on a belief that the

Kumlander D. (2008).
BRIDGING UNCERTAINTIES GAPS IN SOFTWARE DEVELOPMENT PROJECTS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 240-245
DOI: 10.5220/0001693702400245

problem can disappear by itself. Actually there are different uncertainties types and one massive class of uncertainties are low occurrence probability uncertainties (that the certainty is different to what is expected or known at the moment), which are triggered by strange or add-hoc thoughts that customers or designers could suddenly come up with. In this case the reactive approach is working well as people use to discard many requests they raised earlier. Unfortunately the reactive approach is not universal. Sometimes it is too late to react on a problem when it finally occurs. Uncertainties are potential problems by our definition as in the future the software could have to meet requirements that are not initially included into the design. Therefore, if a certain (well formulated) requirement appears, let say in the end project phase, then it could demand a full rebuild of the architecture and it is surely an expensive thing to do so late – expensive from time, resources, marketing and so forth points of view. All this means that the reactive approach is efficient to decrease the number of issues the software team has to deal with, but is expensive if any potential issue will turn into reality.

The second approach is a "pro-active" approach. Here any uncertainty is addressed right from the start (after an uncertainty is identified). There are different sub-approaches as for example risk management best practises and open issues monitoring methods (Sumner, 2000; Karolak, 1997; Kumar 2002).

Comparing the proactive approach to the reactive one, we could say that the proactive one surely demands much more resources and is not very efficient dealing with a large number of uncertainties. At the same time it is capable to decrease the overall risk a lot, so the project is efficiently controlled since all potential problems are addressed as much as possible minimising the potential loss in the project.

All this means that uncertainties should be divided by priorities and potential impacts. The uncertainties with low occurrence probability and low potential loss are not directly addressed by pro-active methods and the paper is going to follow this best practise. The remaining uncertainties should be addressed actively and the paper will propose a set of actions specific for software engineering to deal with those uncertainties.

3 BRIDGING UNCERTAINTIES GAPS

First of all, an efficient uncertainties gaps bridging requires close monitoring and dealing with all average and high impact uncertainties. Secondly those should be dealt actively in order to achieve the required efficiency. Finally uncertainties should be addressed using different methods rather than using a common approach. Uncertainties are like people – they have different properties, nature etc. General methods for addressing all kinds of risks that project management is used to use from projects to projects is shown itself either as not very efficient since it is usually quite passive or as not applicable at all for the uncertainties management in software projects. The common methods been applied usually give some "good" feeling (confidence) to project management that risks are under control hiding the real status. From our point of view, it is one major reason why so many software projects fail nowadays – those are monitored using general project management approaches (notice again that uncertainties is something else than risks, so risk handling methodologies are not applicable here) and as a result uncovered uncertainties turns into problems close to the project release date and as a result the project either requires much more resources than initially was planned or its functionality is cut down or the functionality is implemented "somehow" in order to release the project in the agreed time period.

Underneath, different types of uncertainties will be reviewed proposing corresponding actions for each one in order to minimise uncertainties impact to the software engineering project result. Under minimising the impact the following is meant. Let's review a moment in the future when an uncertainty will turn into a certainty, i.e. a certain requirement to the developed software (its functionality, used hardware, performance etc.) will become formalised. The local uncertainty managing task is to minimise the difference between software that will be developed at that moment and the new formalised requirement. The closer we are to the new target the better we managed the uncertainty. At the same time the spent efforts to the moment of eliminating the uncertainty should not exceed the benefit. Therefore the global (and true) uncertainties gaps impacts minimising task is to minimise the total efforts to be applied to achieve the final goal: the software product that meets all expectations after it is released.

Therefore the process of bridging gaps should always make some trade-offs between current extra activities (spent resources) to handle uncertainties and activities avoided in the future (how much we will have to spent after an uncertainty is cleared up). Notice that the potential impact could include extra hours we need to invest into engineering sometimes including over-hours in order to meet deadlines, a potential loss of contracts if the desired functionality cannot be developed in time, postponed releases and negative impact on the software vendor imago or penalties.

The first uncertainty to be discussed here is an unclear specification or a specification missing some important elements. The process of requirements gathering is not discussed here as there are well known approaches (Somerville and Jane, 2005; Somerville and Jane, 1997), but those do not ensure that the final specification can be produced immediately. Quite often some additions are expected in near future. The easiest way to address this problem is to implement as broad system scope (architectural, technological and functional) as possible to make the system quite universal. At the same time the system scope should stay as narrow as possible to minimise resources spent on the system implementation, to make the final system fast and user friendly. Therefore there is always a trade-off to be made between universality and applicability of the system, which requires:

- A skilled person to make the good trade-off instead of the bad one (the system should be efficient and broad instead of narrow and slow);
- Enough information including probabilities and varieties that uncertainties produce.

Concluding the previous list, we need first of all information on uncertainties flowing through the whole system as early awareness allows minimising negative impacts. Thereafter enough competencies is required to evaluate uncertainties – guessing the likely result after the uncertainty is turned into a specification or what the uncertainty could mean/require. Many software companies isolate the development team from talking to customers deriving obvious advantages from that and forgetting about the lost part. The less designers, developers etc. know about customers the more artificial system they produce in the end of ends. Therefore we do propose organising customers review meetings, which can be done by consultants or requirements collectors, in order to keep key project persons (those who are making decisions, design the system etc.) informed on use cases. This will make information on uncertainties meaningful.

Notice again that uncertainties are not always potential problems. Those allow to see the future system from more points of view and the broader scope doesn't always mean a slower or less user friendly system. Besides uncertainties could highlight other potential ways to use the system, so it will already meet some future (so far unformulated) requirements and markets.

The next uncertainty to be discussed is more general than the previous one. Consider a situation when the testing team is waiting for development to generate test cases and development is not able to come up with a technical solution. Consider also the previously described case of incomplete specifications, incomplete design etc. All this can be generalised by the following definition: there can be an uncertainty of functionality at any software development step produced by a previous step as its production is put on hold since the previous team is waiting for some information. In most cases the missing information is not more than 30% of the full package. The main issue here is that other teams cannot effectively do their work. So, they either stop their activities or do something basing on their "believes" facing a risk to re-work that part. Notice that typically, despites such slips in deliveries, the general project deadline is not shifted since it is agreed with customers or top management. As there nothing the dependent team can do about the problem, the solution for this case is based on addressing the "believes" word from the problem description above. If the dependent team has to act then they should do it basing on all currently available information. Therefore it is useful either to provide periodically information further down the project cycle or grant them access to the previous step team documentation base. The common outputs (documents) infrastructure that is visible for all teams' members can greatly assist solving the issue. Notice that the lack of information is not the only case. The other one is too much information – for example plans are changing each day, each small probability of changes in future is immediately submitted down to the development cycle producing quite a lot of chaos in plans and the implementation process. Therefore a balance is needed to achieve the global optimum minimising all kinds of extra work handling uncertainties.

The next uncertainties class a matching between the implemented software and stable requirements, considering a case when developers do interpret requirements incorrectly. It is an art of

communication and management to ensure implementation of exactly what was asked. The typically problem appearing here are communication gaps. Formally, a communication gap is a problem in the communication process that makes the transferred information to be either lost or deformed. So requirements, which are flowing through design and thereafter (in a form of design) through development, have a lot of chances to be misinterpreted leading to incorrect software release. There are a lot of reasons why it could happen. 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) (Ludlow and Panton 1995; Kumlander, 2006b; Hadelich at all., 2004), 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 (Cramton and Weber, 2003; Kumlander, 2006c). It is quite a typical situation nowadays as there are much more distributed organisations than it looks like at the first glance. Sometimes companies become distributed by their own wish since:

- The development process will be cheaper.
- There is a misfit of a skilled personnel location(s) and product markets. Unlike the previous case the cost of development is not necessarily decrease, but company gets much more skilled employees.

Sometimes companies become distributed because of external reasons:

- After buying other companies located in other geographical regions;
- Company branches have to work together although it wasn't planned so initially;
- Globalization of operations, i.e. a need to establish groups in other regions.

All this results in decoupling the development team into offices that are managed remotely. The problem is usually even deeper since information submitters rarely verify the information transfer process results and therefore the corruption stays invisible until the release is reviewed, i.e. until the very late phase.

This type of uncertainties can be solved varying the iterations' length. The modern software development, like for example the "extreme programming" and the "agility with SCRUM", do release software in a set of concurrent steps – the software functionality to be released is divided into parts and each part development goes through the full set of development work cycle steps and is called an iteration (Boehm , 1988). The central idea here is to produce a possibility to verify intermediate versions with customers or management and fix possible problems during the next iteration, i.e. as soon as possible. So the iterative software development is a possibility to remove uncertainties, and the shortened iterations cycle will sufficiently decrease the overall uncertainty for all project teams. Unfortunately we cannot just propose using the shortened cycle since it still should be as long as it needs to be efficient from the development point of view. Besides, too often releases (demos) makes customers unhappy since are demanding too much their time, which is of course valuable. Therefore the shortened development iterations are proposed to be internal and probably exclude some substeps of the full iteration. Under the internal cycle we mean an iteration that will end up with an internal demo to business analysts, management, testing team and so forth in order to verify the done part and ensure its correctness. Notice that this demo is done not only to persons that do verify the result, but also to others sufficiently increase their awareness and solving problems of uncertainties described above (see first two classes of uncertainties). The demo usually takes just an hour, so it will not demand too much time and resources from others. Under excluding some steps, we mean that it doesn't necessarily require the full testing or documentation in order to achieve the goal (verified iteration) with minimal resources. The internal demo can be done during weekly/daily meetings, if any is established as normally such project meetings involve all required attendees. Moreover it can be done in a web environment in case participants are located far away from each other.

The varying iterations length leaves enough space for the next extra rule: the more important part is developed (from its impact on the next phases) or the more misunderstandings we faced recently the shorter the cycle should be. It is vital to verify results acting in an unstable software development environment.

Notice that possible changes of expectations during software project implementation require both earlier mentioned approaches to ensure the match between expectations and software in future: we need to verify the current software (developed so far) and have a good architecture, which is broad enough to feet re-designs into. Moreover the expectations migration can be produced by

intermediate releases, but it will be a controlled migration which isn't raised suddenly. So the expectations' change is not an uncertainty impact any longer, but is a cooperative evolution.

A very important approach to be proposed next is an ambassador driven communication. The idea of ambassadors is similar to the political one – the ambassador is an official person accredited to solve a team problem either in another team or in another branch. A division on teams always means that certain communication barriers arise between teams within the same project. Sometimes you need somebody, who can speed up some processes by walking into a person office and asking to do something the team believes is important. People are normally quite slow to do something that is requested by emails or phones and do much quicker after they are asked (pushed) personally. The same can be stated for a special person within the customer company who can help to find requirements in a reasonable time frame. Another typical barrier that was mentioned earlier is a physical distance between teams. For example a manager locating in the head office looses a lot of information sources like informal one, a possibility to walk around and see what people are doing and having troubles with and so forth. All these troubles can be solved by ambassadors and ambassador's duties could take 10% or less work time of a person speeding up the process and decreasing the project uncertainty sufficiently.

One more interesting approach to decrease the overall uncertainty in the project is promoting discussion groups. The main idea here is to let people collaborate and post problems, highlight potential disadvantages of the current solution, discuss possible impacts of future requirements and propose solutions. The following uncertainties to be targeted here:

- Mis-implemented functionality. It is a way to verify and control how the information is understood by other people. The incorrect information interpretation will likely produce faulty questions or will lead to lack of understanding during conversations;
- Incomplete outputs that are not posted– early awareness of all team members on possible future problems and solutions is ensured during discussions;
- Uncertainties on methods to be used to implement the software.

Notice that too big groups are not advised to have as it will produce too much information moving around and sometimes certain anarchy. Therefore local and global groups can be created – local groups within each team and a global one that should include key persons and experts. This will produce certain hierarchy of discussions. So, all questions from less experienced members (which are probably their wrong vision or misunderstanding) can be addressed without requiring experts' attention and good ideas can be promoted to the highest level. The discussion groups do also promote the collaboration and cooperation (Rauterberg and Strohm, 1992; Forsgren, 2006) between different groups producing a synergy in result.

Another uncertainties type is technological uncertainties. Sometimes it is specified what should be done, but it is not clear whether or how it can be achieved. The technology lack can arise at any stage including development, testing (for example how new functionality can be tested in order to ensure required performance, extensibility etc), logistic (the system installation procedures and supply channels) and so forth. These uncertainties are typically covered by pilot projects and we don't see any better solution here. The only remark we can do – the pilot project has to be carefully organised, for example it requires information on desired and acceptable outputs, correctly defined and results should be verified.

Finally uncertainties of missed information needs to be addressed actively by corresponding iterations and functions' implementation project planning. The information about uncertainties should not be just kept somewhere. Instead, it should be directly reflected in the project plan. It is obvious that features we don't know anything about should be planned to later phases. Unfortunately it is less obvious for many managers that features which are specified only partly should also be planned to later stages. Sometimes the incorrect planning leads to a situation when the next step team starts to press out outputs from previous teams although they are not able to provide any, and lead to internal conflicts. It is important to not mix this situation with technological uncertainties or uncertainties on wishes. The last one can be solved using pilot projects to demonstrate the proposed software to customers in order to formalise desired functionality, interface design etc. – in the same way as technological uncertainties are solved by attempts to achieve the desired technological goals. Those have to be planned as soon as possible to decrease the overall project uncertainty.

4 CONCLUSIONS

The uncertainties is a well known factor affecting the final result of nearly any software project especially in the modern quickly changing world. The article aim is to propose some new methods to minimise the negative impacts that uncertainties have on software development process. The following uncertainties types arising in software development projects were reviewed in the article:

- Unclear / incomplete specification (requirement);
- Unstable customers/management's opinion/vision on how the final system should work / look like etc.
- Inability to predict the software project final output (how the system looks like, works etc.) due possible requirements transformation during implementation since requirements are incorrectly interpreted, details are lost etc.
- Unclear effect of the current requirements on later stages: technology (can the required be achieved), amount of required work (testing, development, education) etc.

Although uncertainty risk management is not something new and there are several methods targeted to solve those problems (Sumner, 2000; Karolak, 1997; Kumar 2002), the number of failed projects because of extra costs or mismatches of customers' expectations and released software produced by uncertainties is quite high. The paper has proposed the following methods in addition to existing to coupe with uncertainties in order to bridge the earlier mentioned gap between reality and requirements:

- Promote information on uncertainties to flow freely through the whole system and customers review meetings in order to give enough knowledge to key persons to deal with provided uncertainties information;
- Shortened iterations cycle (varying length cycles) with an internal software demonstration meeting to verify it and make others aware of what is done, what are current problems and what is in development;
- Ambassador driven communication;
- Discussion groups;
- Pilot projects (is not a new method);
- Careful project planning to start pilot projects as soon as possible and uncertain functionalities (that cannot be finalised [developed, specified etc] right now) development as late as necessary.

REFERENCES

Bennatan, E. N., Emam, K.E., 2005. Software project success and failure, *Cutter Consortium*, *http://www.cutter.com/press/050824.html*

Khan, A. A., 2004. Tale of two methodologies for web development: heavyweight vs agile, *Postgraduate Minor Research Project*, 619-690.

Ludlow, R., Panton, F., 1995. The Essence of Effective Communication, *Prentice Hall*.

Kumlander, D., 2006a. Software design by uncertain requirements, *Proceedings of the IASTED International Conference on Software Engineering*, 224-2296.

Somerville, I., Jane, R., 2005. An empirical study of industrial requirements engineering process assessment and improvement, *ACM Transactions on Software Engineering and Methodology*, 14(1), pp. 85-117.

Kumlander, D., 2006b. Bridging gaps between requirements, expectations and delivered software in information systems development, *WSEAS Transactions on Computers*, 5(12), 2933-2939.

Boehm, B.W., 1988. A spiral model of software development and enhancement, *Computer*, 21(5), 61-72.

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

Somerville, I., Sawyer, P., 1997. *Requirements Engineering – A good Practice Guide,* Wiley.

Kumlander, D., 2006c. 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, 181-192.

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

Cramton, C.D., Weber, S.S., 2003. Relationships among geographic dispersion, team processes, and effectiveness in software development work teams, *Journal of Business Research*, 58(6), 758-765.

Hadelich, K., Branigan, H., Pickering, M., Crocker, M., 2004. Alignment in Dialogue: Effects of Visual versus Verbal-feedback, *Proceedings of the 8th Workshop on the Semantics and Pragmatics of Dialogue*, *Catalog'04*, Barcelona, Spain, 35-40.

Sumner, M., 2000. Risk factors in enterprise-wide/ERP projects, *Journal of Information Technology*, 15(4), 317–327.

Karolak, D., 1997. Software Engineering Risk Management, *IEEE Computer Society*.

Kumar, R., 2002. Managing risks in IT projects: an options perspective, *Information and Management*, 40, 63–74.