A pattern language for software quality assurance with small resources – Action patterns

Wolfgang Zuser, Armin Scherz, Thomas Grechenig, Martin Tomitsch
Industrial Software
Vienna University of Technology
{firstname.lastname}@inso.tuwien.ac.at

Introduction

A working software quality assurance is one of the key success factors for every software project. The existing restrictions of many projects like limited resources often do not allow a QA-department or specialized QA workers on their own. Instead in such projects you have excellent software engineers doing the QA job together with their daily project work. They are concerned about the necessity of QA and they are able to do good QA within projects. This pattern language presents a set of patterns, which can be applied to any IT project which is not able (because of given limitations) to spend many resources on QA or which does not want to in order to save resources. The patterns are divided into strategy patterns, which can be applied to some general issues in QA. Furthermore there are action patterns, which are directly applicable in the daily routine of IT workers. The patterns can supply a minimal set of QA activities. Together with excellent software engineers they are sufficient to produce high quality software.
The audience of this pattern language are project managers and software engineers, who are concerned about the quality of their products and are willing to have quality assurance mechanisms applied to their project but do not have enough resources to have specialized QA people doing this job.

Roadmap

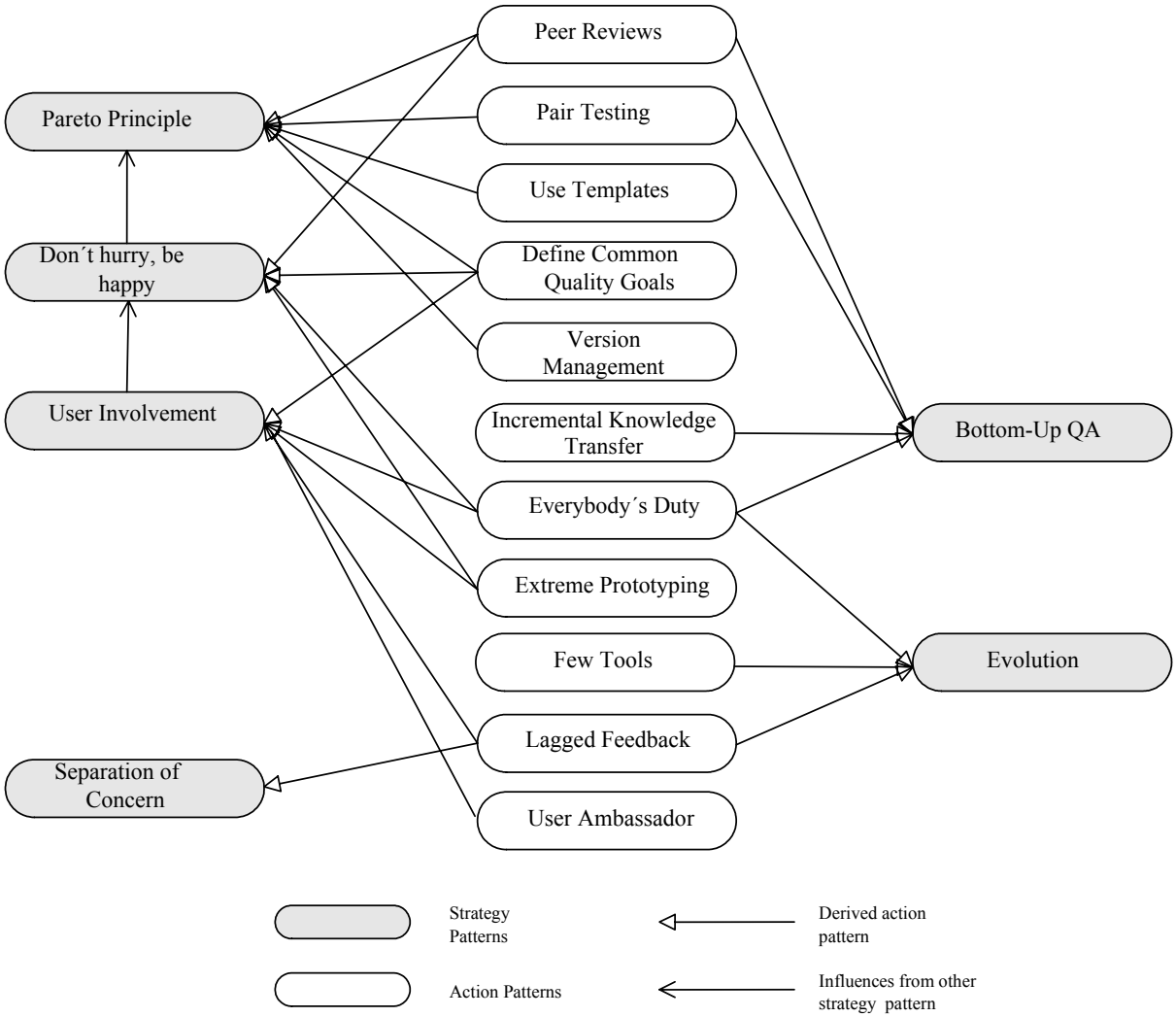


Fig. 1 Roadmap to a pattern language for software quality assurance with small resources

Strategy patterns (not part of the submission)

These patterns describe the strategic background for software quality assurance with small resources and represent the main ideas. The patterns are so abstract that they can also be used in other contexts.

The "Strategy patterns" are:

- Pareto Principle: A pattern for a good distribution of your available resources.
  - Problem: ***How do you distribute the available resources along the project to get the best results?***
  - Solution: ***Apply the tasks with the highest rate of outcome vs. costs/time to achieve a great amount of the quality quick.***

- Bottom-Up QA: Start your QA activities at each single developer.
  - Problem: ***How can you have QA without an independent QA-department?***
  - Solution: ***Motivate every software engineer to use QA methods in the small to ensure that his/her individual work is of high quality.***

- Don't hurry, be happy; but do not fall asleep: Concentration on the early phases of software development should help you to achieve good quality software.

    - Problem: ***On which phases during software development should you concentrate QA activities for achieving high software quality?***
    - Solution: ***Especially concentrate your software quality assurance on the early phases of software development but do not forget to continue with your activities later on.***

- User Involvement: A pattern about tight user involvement for an optimum of information exchange.
    - Problem: ***How to continuously and efficiently get information about how the system should work?***
    - Solution: ***Involve the users as much as you can.***

- Evolution: Build you QA system step by step.
    - Problem: ***How do you find a suitable Software Quality Assurance system for your software company?***
    - Solution: ***Start with selected good fitting QA practices and improve and extend them from project to project.***

- Separation of Concerns: A pattern about how to distribute many concerns on few people.
    - Problem: ***How do you group many concerns for execution by few persons?***
    - Solution: ***Form bundles of concerns around critical concerns, which can be executed by single persons a one point of time.***

Action patterns

The "Action patterns" describe applicable methods for small and medium software projects. They are derived from the "Strategy patterns" and have a much more practical character. The set of "Action patterns" is a tool box for managers and engineers who want to do QA with small resources. They can select the patterns which are interesting for their own project and adopt them to their special needs.
The "Action patterns" are:

- Incremental Knowledge transfer: One approach for efficient knowledge acquisition in small teams.
- Few Tools: Manual operations are not effective.
- Use Templates: Helps saving effort and increases quality.
- Version Management: Even a few people need some help at integration of different modules.
- Everybody's Duty: QA should be as important as implementation or testing.
- User Ambassador: A dedicated user representative should be involved in the project team.
- Define Common Quality Goals: Just in case you want to achieve quality it is a good idea to have quality goals.
- Extreme Prototyping: A useful method for user involvement.
- Peer Reviews: This pattern describes one way of what and how to review in small software teams.
- Pair Testing: This pattern describes a test strategy small software projects.
- Lagged Feedback: How to preserve experience in small companies.

Known Uses

The patterns of this pattern language are the result of the experience of the authors with quality assurance activities in industrial projects in several different companies. Following selected recent projects in three Austrian companies will be used for known uses in the following patterns:

- KPF: The project „Crop compensation“(Kulturpflanzenausgleich - KPF) is a subproject of the INVEKOS projects (integrated administration and control system), which is a system for the electronic handling of the EU agricultural grant program for Austria (http://www.ama.at/portal.html).
- WWS: The WWS project has to deal with a small specialized ERP system for a large Austrian commercial enterprise.
- eBSS: This financial trading platform provided by the Austria's Export Credit Agency (OeKB) enables private customers to buy and sell short running obligations from the Austrian ministry of finance (http://www.bundesschatz.at).

Pattern Form

The patterns of this pattern language will have the following form:

Name	A short descriptive pattern name.
Context	Description of the context; derived from the example.
Problem	The underlying question.
Forces	What makes the problem a problem?
Solution	The basic idea of the solution.
Consequences	The resulting context after using the pattern.
Known Uses	Know applications of the pattern in practice.
Related Patterns	Internal links to other patterns of the language.

Action Patterns

1. Incremental Knowledge transfer

Context

The most important factor for high quality software is the knowledge of every single software engineer. Only skilled engineers can produce high quality products. In many projects it is not possible to get enough engineers with all the knowledge required.

Problem

How can you increase the worker's individual productivity and the quality of their products within a small company's budget?

Forces

- One of the most important resources in software development is know how. However know how transfer is expensive.
- It's expensive to have separate QA departments holding the knowledge about QA. However without a separate QA department you have to establish knowledge about QA in your software development teams.
- Small companies can not afford trainings for complete teams. However teams have to improve continuously.

Solution

Support the transfer of knowledge within the development teams.

Single developers will share their knowledge within the project team by showing new practices in their daily work and by explaining practices in special internal meetings or just in between the daily work. In small project teams there is much direct communication and therefore direct know how transfer.
Developers can gather new knowledge in trainings. Other team mates can benefit from others trainings as well if the new knowledge can be shared.
Instead of an external training, for single developers a personal internal training program can be granted. Again the improvements made with such a program should be shared within the whole team.
Effective ways of sharing knowledge are:

- Developers with valuable knowledge conduct small training units for others team members. Especially at project start and the time around major milestones there is some time for such activities.
- Groupware tools: The tools enable the asynchronous sharing of knowledge.
- Documents: A more durable way across projects of sharing knowledge is to write text documents.

Resulting context

Well educated engineers produce high quality software. New know how of single developers will spread all over the team after some time. This increases the system quality in single programmer project as well as in team projects. Good training increases productivity as well, which guarantees a return of training investment at some degree.

Related patterns

- Everybody's Duty: To be able to produce high quality software a proper knowledge of the state of the art and practice is vital.
- Tool Support: The transfer of existing knowledge can be support by groupware tools or tools helping to create a persistent form of knowledge (e.g. a word processor).

Known Uses

- **Software Engineering Education**: In a software engineering course we teach at Vienna University of Technology we use such a system in so called tutorials. One team member has to prepare on a certain topic and presents the most important facts in a formal meeting.
- **Groupware:** Many groupware tools support the idea of sharing knowledge by providing forums and document upload.
- The **Personal Software Process (PSP)** training program provides an improvement framework for individual software engineers. The training enables the engineer to estimate the required programming effort for a specific problem, to plan the necessary work and to evaluate the quality of the development process afterwards. A single PSP training program requires 120 to 150 hours per worker ([PSP]). The knowledge gain from such an individual program can be shared with the rest of the team.

2. Few Tools

Context

Currently tools for every project phase are available. These tools range from modeling and simulation tools to test data generation tools. For the ease and the quality of software production the choice of the right tool support is decisive.
To use tools software developers need the appropriate training. This causes additional costs which have to be taken into account beside the costs of buying tools when a new software tool is introduced.

Problem

How many tools are necessary to be productive enough as well as to be able to produce high quality software?

Forces

- The high number of different tools makes it difficult to decide about the tools one should use.
- Sometimes it is also not clear if the benefit of tool support is bigger than the costs of the tool and the necessary training for the developers.
- Expensive tools probably have low utilization.
- Only few tools can be bought because of a tight budget.
- Good developers do not need an excessive tool support.

- However without tool support productivity is not high enough.
- However tools help to handle project complexity.

Solution

Use few tool with high value for the project.

Invest in people not in tools only. Software Tools are an important way to increase both, software quality and the efficiency of the software production. The most important decision concerning tool support is the choice of the right programming language together with a sophisticated integrated development environment. The programming language should support the project goals. Primarily the programmers should feel comfortable with the selected language and should have experience using it.
The second most important tool categories are modeling tools and testing tools, which help programmers to avoid and find defects directly.
Most other tools do not have a great impact on the focused project size due overhead costs for initializing the tool and not enough project complexity, where the tool could help to reduce the complexity.
Therefore well trained developers will be able to handle the complexity of small and medium sized projects without excessive tool support.

Resulting context

The use of various tools is no substitute for well trained software engineers and a well designed and controlled software development process. Tools can only support people and such a process.

Related patterns

- Evolution: When developers are used to existing tools new tools can be introduced stepwise in order to increase productivity stepwise.
- Incremental knowledge transfer: This is the most effective way to distribute knowledge from single developers among the whole team.
- Everybody's Duty: Doing a good job and producing good software is not a question of tools only but as well a question of the motivation to do so. A fool with a tool is still a fool.
- Version Management: Can easily be done without tools at all (by defining a proper directory structure and using proper files names).

Known Uses

- **Spreadsheet based testing in eBSS:** Instead of introducing a commercial test automation tool the testing of batches was automated by defining the test data with spreadsheets and running test scripts using the data in command line mode.

3. Use Templates

Context

A lot of activities during software engineering reoccur in every project. Some of them are done for several times during one single project.

Problem

How can you support tasks which have to be done regularly?

Forces

- Project standardization aims at a lot of tasks during software development, which can be done in the same way in every project. However in many cases people who do the same things over and over again begin to loose concentration and do things wrong.
- There are a lot of tasks which have to be done in many projects similarly (this speeds up the execution of the tasks). However there are slight differences (the necessary adoptions slows down the tasks).
- The use of templates can reduce the time for doing recurring acitivities. However too many templates software development gets too formalized and bureaucratic.

Solution

Provide an elementary set of templates and checklists.

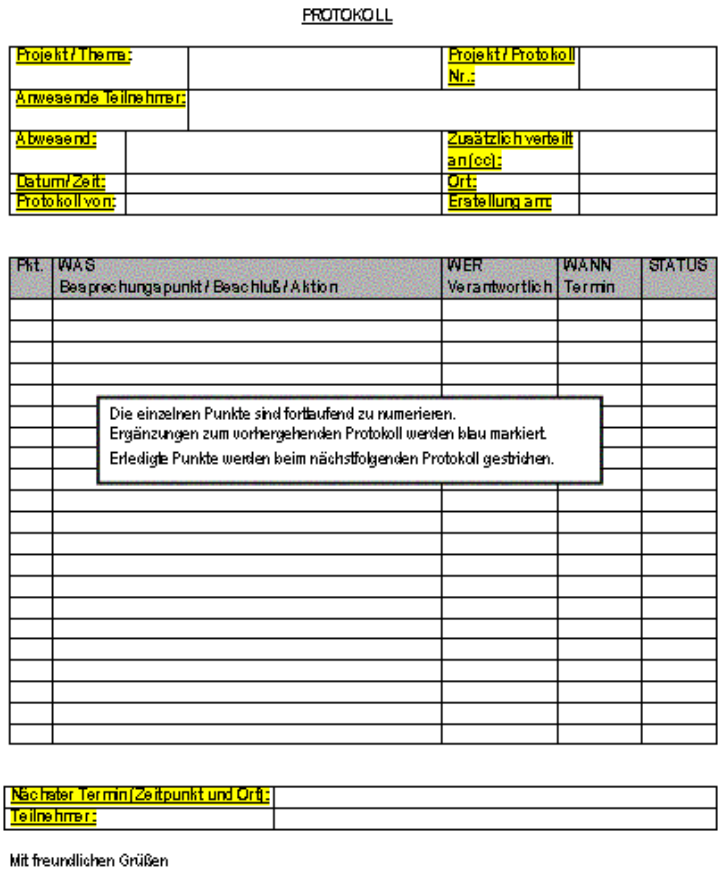
Templates and checklists are not only helpful during implementation; they speed up and increase the quality of all development phases.
Templates can be used as a "memory" for best practices which have been learned in past projects. They help a software engineer to do a task much faster and in better quality.
Checklists assure that all necessary information is gathered.
Templates can be extracted from existing and successful (in a sense of providing valuable information in an effective and understandable way) documents of a finished project.
Templates should only define basic items with a high probability to be valid in many projects.
Over time variations of templates can be created.

Resulting context

After a series of projects there will be a set of useful templates and checklists for several development tasks. If the templates really help the engineers they will use them voluntarily. There should be no pressure to use the templates.
The set of templates and checklists will change over time. They represent a part of the software engineering know-how of the company.

Known Uses

- **Templates in major software development processes:** Rational Unifed Process and Microsoft Solution Framework define a rich set of templates.
- **Templates for documents and project management:** In the KPF and WWS two templates were defined and used: a template for protocols and one for documents. Each template defined the front page, the main structure and style definitions.
- **Protocol templates at XEROX™:** An example for a protocol template used at XEROX is shown in Fig. 2.



PROTOKOLL

Projekt / Thema:		Projekt / Protokoll Nr.:	
Anwesende Teilnehmer:			
Abwesend:		Zusätzlich verteilt an (cc):	
Datum / Zeit:		Ort:	
Protokoll von:		Erstellung am:	

Pkt.	WAS Besprechungspunkt / Beschluß / Aktion	WER Verantwortlich	WANN Termin	STATUS

Nächster Termin (Zeitpunkt und Ort):	
Teilnehmer:	

Mit freundlichen Grüßen

Fig. 2: Examlpe template (in use by XEROX™)

4. Version Management

Context

Most software projects produce a vast number of different documents, like analysis documents, design documents, source code files, databases, test reports, communication protocols, review reports…). Some of these elements also occur in different versions and subversions. All software elements have relationships to other software elements. In addition the numerous relationships between the elements cause troubles. If one element is changed maybe a lot of interacting elements have to be changed, too. If a new system version doesn't work it might be necessary to step back to an earlier version. Therefore all changes have to be documented and controlled.

Problem

How to manage different versions of software elements (documentation, analysis and design documents, source code…) and their relations?

Forces

- In a small software project there are different software elements like in any other project as well. It is impossible for a human being to keep an overview over all the elements without a proper structure. However there are less software elements and fewer versions than in larger projects.
- No configuration management will make it difficult to keep track on valid and working configurations. Full size configuration management causes a lot of work to maintain configuration information.

Solution

Do version management only (not configuration management).

Large Scale Software Configuration Management consists of:

- Identification: All Software elements get a unique id.
- Change control: Providing formalism for changing an element.
- Auditing: Keeping the relationships between all elements under control.
- Status accounting: Keeping a history of all elements and all changes.

Small scale version management will be successful by applying the fist two steps identification and change control only.

Resulting context

Version Management is a part of software development which appears in every single project. With the help of standard software tools (e.g. CVS) this job can be done easily. In small project it's enough to concentrate on the management of versions on the most important products. The relations between different products (code, models …) can be additionally documented for overview reasons e.g. with spreadsheets.

Related patterns

- Pareto Principle: Following the pareto principle version management is very effective regarding the low effort for doing it.
- Few Tools: Version management can be done without tools.

Known Uses

- **Version information as part of file names:** In the eBSS projects each new version of a document was stored in a filename, where the version number was added at the end of the file name.
- **Releases in different directory subtrees:** In the WWS project different releases where stored and maintained in different subtrees of the project directory.

5. Everybody's Duty

Context

If there is no QA department, QA has to be done by somebody else. In case of small resources usually an independent QA department is not affordable.

Problem

Who is doing quality assurance in case there is no separate QA department?

Forces

- Well trained software engineers know about the importance of QA. However a lot of roles, which are involved in software development, have short-sighted interests which contradict the quality assurance work.

- A specialized role for QA could do an excellent QA job. However there may be not enough project members to have somebody doing this role only. There may be not enough QA work for a full time QA role.
- Distributing QA among all roles distributes this unpopular work. However some people like doing QA some people do not want doing QA at all.

Solution

Quality assurance should be an integral part of each role in software development.

Only if QA activities are embedded into the development process the quality targets can be met.
To achieve this, all roles in a project have to share common goals and work together. It needs a lot of information and motivation to get this to work.
The first step towards anchoring software quality assurance in a project is to inform all participants about the goals and methods of QA. This can be done by a kick-off meeting. During this meeting a QA plan (which plans all the QA activities in the project) is presented and all people have the opportunity to ask questions and propose better measures. As a result of the meeting the QA plan might be changed to fit the specific project. It is important that members of the customers are invited to the kick-off meeting.

Resulting context

An atmosphere that enables the project team to perform the required quality assurance measures should be created. This is extremely important for measures that slow down development in early phases and pay off much later (e.g. analysis and design reviews).

Related patterns

- Bottom up QA: There is nobody left if the developers are not willing to do QA.
- User involvement: The duty of contributing to QA also affects the customer and the users. E.g. they should support the goal by providing necessary informations as soon as the can.
- Don´t hurry, be happy, but do not fall asleep: The first phases have the most influence on the overall project quality and therefore the whole team should start to concentrate on QA in the early phases.
- Define Common Quality Goals: QA throughout the whole team only is possible of there is a common view of the quality goals.

Known Uses

- **Whole team review:** In the eBSS project most of the documents and parts of the source code were reviewed by the whole team.

6. Users Ambassador

Context

There a lot of different stakeholders in a project (contractor – the one who wants to build the software and get money for it, customer – the one who pays for the project, users – those how use the final software, developers – those how really build the software). The have different views of the software system and its costs.

The user's view on a software system is decisive for the way the system should work. Every user has a different view on the software system and different needs. These views and needs are very different from the view a developer has on the system.
The portions of needs which are fulfilled determine the whole project success and for the amount of money which will be paid for it.

Problem

How to create a common view of the system between users and developers?

Forces

- A common view is essential for a successful project. Creating a common view of the system between all users and developers takes a lot of time (reducing the available time for implementation and testing). Spending not enough time on building the common few may endanger the whole project.
- Developers want to minimize work (which means few and simple requirements without special cases). Users want to maximize functionality (many requirements with many special cases).
- Customers want to minimize costs. Users want to maximize functionality (which causes costs).
- Developers are available all the time (if they are fulltime assigned). Users have to do other stuff as well.
- Various review cycles are desirable for a good requirements analysis. Review cycles are expensive and take a long period of time. Small projects have a tight schedule.
- Few people will elaborate a common few quicker than many people. Few people will consider fewer concerns than many people.

Solution

Involve one representative user into the software development process from the very start till the end of a software project.

To achieve this involvement a dedicated contact person within the customer's organization should be named. For the project duration this person is part of both, the customer's organization and the software project team (see Fig. 3). The user representative is in charge for presenting a common view of all user needs. The user ambassador must be capable for representing all requirements satisfactory and also for handling basic technical issues.

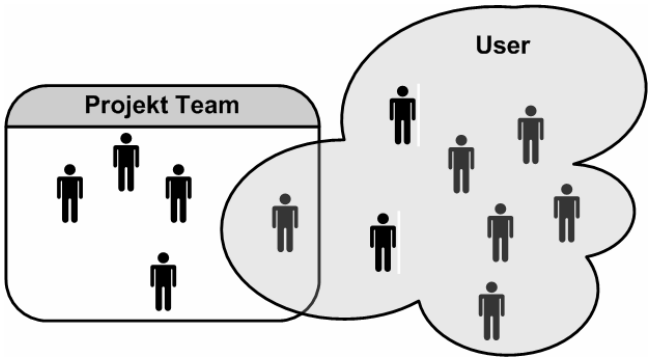


Fig. 3: User involvement in the software project

Consequences

The involvement of the user in the software development process at the very beginning has a lot of positive results:

- System requirements are analyzed well enough due the tight contact to a user representative with minimum of time effort.
- Users have to agree on a consensus about the systems functionality before the users ambassador will talk to the developers. Special cases and requirements for few people will usually be eliminated during this process. The work for the developers can be reduced.
- The external project costs for the customer will be reduced by reduced project organization costs (e.g. due low meeting costs). Anyhow the customer will have additional internal costs (due the internal meetings of all users with the ambassador. Still there are cost saving because internal costs should be lower than external costs.
- The availability of one single person for meetings with the project team is higher than the availability of many persons. Information can be exchanged intensively.
- Review cycles can me shortened by having only one reviewer at the customer side, which represents all customer interests.
- For projects sizes which can be handled by SMEs one person will be able to handle all project concerns quite will. In larger projects maybe one single user representative could not cover all necessary concerns.

Related Patterns

- Pareto Principle: For user involvement is expensive, it should be limited to the most important activities.
- Don't hurry, be happy: User involvement is most effective in the beginning of a project.
- Everybody´s Duty: Also the user has to understand the need for QA.
- Define Common Quality Goals: The customers view is decisive for the quality goals.
- Extreme Prototyping: Is a tool for user involvement, because the prototypes can be presented to the user.
- Lagged Feedback: Only the user can say if his needs are fulfilled by the system.
- SurrogateCustomer: In case the customer is not available enough, the "OrgPatterns" suggest to simulate the customer within the team (http://www.easycomp.org/cgi-bin/OrgPatterns?SurrogateCustomer).

Known Uses

- **Wieger's Project Champion**: Wieger suggests naming one "project champion" who is part of both, the customer's organization and of the project team.
- **Extreme Programming**: One of its rules is "The Customer is Always Available". Anyhow it does not distinguish between the customer and the users.
- In the **eBSS** project one representative of the customer worked very tightly (at least twice a week, during requirement engineering daily via phone, mail or the change management system) with one of the developers on the one hand and the quality assurance for test planning on the other hand.

7. Define Common Quality Goals

Context

Successful quality assurance is always target-oriented.QA measures are always influenced by specific quality goals. In large organizations the QA department cares about how to reach the goals and who should be in charge of it. In smaller organizations this duty has to be distributed within the team.

Problem

How can you reach desired quality goals?

Forces

- Trying to reach single quality goals by single developers will cause high quality according to the distinct goal. However realization of multiple single quality goals may cause quality goal conflicts.
- Trying to reach many quality goals at a moderate level by all developers simultaneously will increase the systems quality linear. However the overall system quality may remain at a moderate level. .

Solution

Define a shared view of the quality goals at the beginning of the project.

To avoid troubles which might be caused by goal-conflicts every goal should have a unique priority.
It's important that the quality goals are defined in a way that enables tests if the goals are reached or not by each team member. A final assessment if the quality attributes have been reached or not should be done in a project post mortem (see Fig. 4).

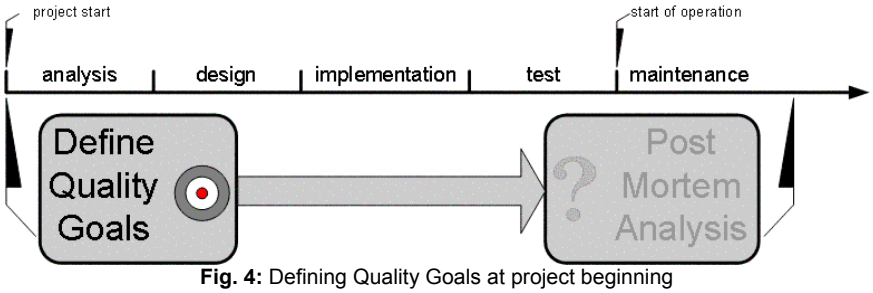


Fig. 4: Defining Quality Goals at project beginning

Resulting context

The QA plan has to be adapted on the quality goals. A clear connection between a quality goal and a QA measure may help a lot of people to understand why a specific QA activity is important.
Clearly defined quality goals may also help the software company when the customers are not happy with the delivered system.

Related patterns

- User involvement: The definition of the quality goals depends mainly on the users view.

- Don´t hurry, be happy, but don´t fall asleep: It is not possible to review quality goals in early phases without a prior definition of the goals.
- Lagged Feedback: A good way of assessing, if the quality goals in a project have been reached.

Known Uses

- **Vision documents:** Major software development processes like the Microsoft Solution Framework (MSF) suggest such documents to define major project goals including quality goals.

8. Extreme Prototyping

Context

For software engineering is a young discipline, most new software systems include solutions, which have never been done before. Therefore many users can not have an idea how their solution may look like. Therefore they are not able to express their expectations properly.

Problem

How can we get enough valid information about the software system which has to be developed?

Forces

- The users have all information about the system. However most users have no idea how a realization of their ideas could look like.
- Building prototypes takes time and money. Building no prototypes makes it hard to get valid information.

Solution

Build few software prototypes concentrating on the most important functions and quality attributes of the system.

A software prototype is a changeable and scalable software system which is much simpler than the desired system. The prototype has not to implement all features; it should focus on the most interesting parts of a problem (with respect to the pareto principle).
Prototypes can be used:

- during analysis to get more information about the problem (Fig. 5),
- to test if a chosen technology is able to fulfill desired quality goals and
- for system implementation if prototypes are extended step by step.

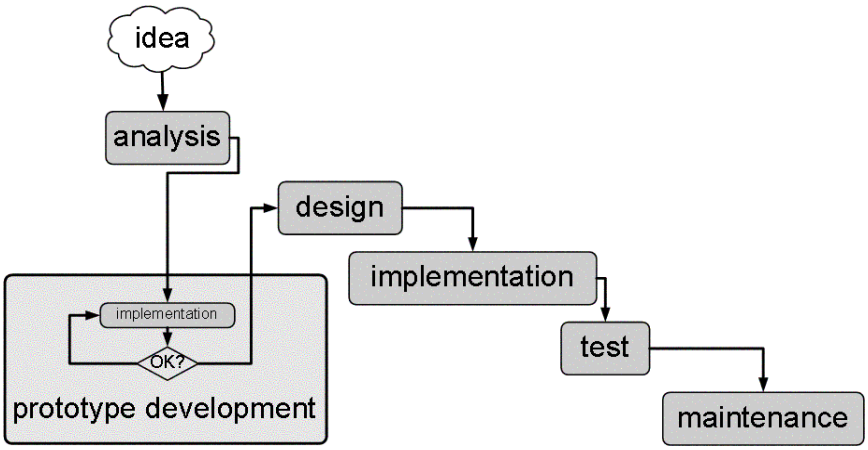


Fig. 5: Software development process with prototyping.

Resulting context

Prototyping helps to identify critical development areas early in the software development process.
The prototype system increases the motivation of the developers because they see a working part of the system quite early. Also the customer might be happy to see a prototype demonstration in early phases of the project.
If a customer sees a working prototype this might lead to some problems. Maybe the customer forces the project team to cancel regular development and to extend the working prototype until the system has enough functionality. This approach is very dangerous because it is a step back to unplanned and chaotic development of software.
Another problem is that the system specification might grow very fast when a user sees a prototype and realizes the potential of a software solution. This can lead to problems with the project's schedule and budget.

Related patterns

- User involvement: Reviewing the prototypes requires user feedback.
- Don´t hurry, be happy, but don´t fall asleep: Prototypes are a good mean for reviewing concepts and ideas in the early phases.
- BuildPrototypes: The creation of prototypes is proposed in the [OrgPatterns] as well as in many pattern languages else.

Known Uses

- **eCard:** Implementing a nation wide system for health insurance approval two prototypes were created. The first prototype was evaluated with usability tests. The second prototype was evaluated with a usability workshop.

9. Peer Reviews

Context

During a review a group of people scans a document for errors and problems. The idea of reviews bases on the effect of teamwork. Reviews can be used for all kinds of documents (analysis and design documents as well as source code) in all phases of the development process. The success of reviews bases on the fact, that uninvolved people have a different

view on a solution and see new errors and problems. Often in a small software company all employees are involved in one project.

Problem

How can reviews be organized in a small software company effectively?

Forces

- Reviews are expensive. If a review meeting takes 2 hours and 5 people are involved and every reviewer needs 2 hours of individual preparation, the review costs at least 20 man-hours. However besides testing reviews are the most important quality assurance techniques in small software projects.
- Extern experts can improve the review process with another point of view. However these experts are usually very expensive.
- Participants from the management may have useful contributions from a more strategic view than the project team. Anyhow in many cases the participation of a management representative is negative because the author of the reviewed document might get under pressure.

Solution

Conduct well organized peer reviews within the team.

A proposal for a review process in small projects is shown in Fig. 6. The most important factor for the success of a review is the review-team. The members should trust each other. In general the authors of a reviewed document must be protected by an experienced moderator (usually the project leader) against personal or unobjective attacks. The moderator can also provide information from the management and experts, which do not participate in the review directly.
The author must remain responsible for the content of the document.
Review meetings should last for not more than 2 hours. After this time span the concentration of the review team decreases rapidly.

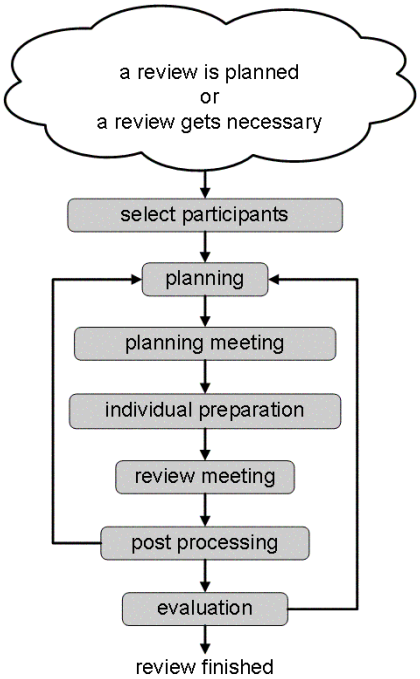


Fig. 6: Review process in a small project

Resulting context

An unexpected result of reviews is that sometimes the mere activity of presenting a document to other people helps the author to discover errors.
Periodical reviews have some positive side effects:

- During review meetings all developers learn intern standards and conventions.
- Different developers learn from each other

Related patterns

- Pareto Principle: The development team will be able to find most of the problems without extensive training in conducting reviews.
- User involvement: Especially requirements have to be reviewed by the users.
- Don´t hurry, be happy, but don´t fall asleep: Finding problems early helps to save time and costs.
- Use templates: Templates can be used for review reports or for a checklist how to do a review.
- Self Motivated Peer Reviews: The pattern language [HopeBeliefeWizardry] states that programmers are motivated enough to conduct peer code reviews by themselves without loss of project time. Of course the author means that ironically.

Known Uses

- **Peer reviews forever:** Since the Austrian software industry is heavily dominated by small and medium enterprises doing small and medium sized projects peer reviews are the dominating review form.

10. Pair Testing

Context

During a test the software is executed with the purpose of finding errors in the software. Testing is the most common quality assurance activity.

Problem

How to guarantee independent tests in small teams?

Forces

- In (very) small software companies usually there is no specialized person for QA activities like testing.
- In (very) small teams there is a limited possibility to distribute different tasks (e.g. programming and testing) to distinct persons.
- However the independence between programmers and testers is a main success criterion for software tests.

Solution

If possible try to engage a tester from another team (Fig. 7), if not guarantee that code is tested by somebody else than the author of the code.

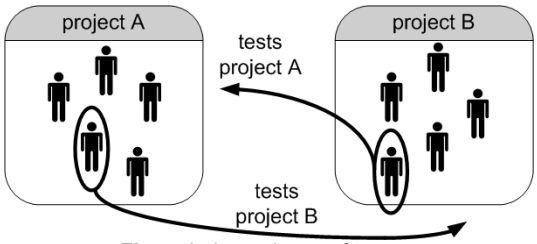


Fig. 7: Independence of testers

Resulting context

A minimum level of independence of testers can be guaranteed. Testing is one important task of QA. Anyhow software correctness can't be assured with tests alone. Tests should be combined with other QA approaches (e.g. reviews, inspections, prototyping).

Related patterns

- Bottom up QA: The testing still is conducted by developers instead of specialized testers.

Known Uses

- **Two teams in eCard:** In the eCard project mentioned above there are two teams: one teams of creating and evaluating the clients. The second team is in charge of implementing a (transaction) proxy and the complete backend servers. The code of the second team naturally is tested by running the clients on the defined interface. Furthermore the programmers of the first team are running white box tests on the proxy and the backend based on dynamic diagrams provided by the second team.

11. Lagged Feedback

Context

Small companies have few customers. That's why it is important to keep the customer relationships vital. Small companies should exactly know how good their final software meets the users and customers requirements. Small companies also have to know, how good their software development process met the customers expectations.

Problem

When should you collect feedback from the customer and users?

Forces

- Immediately after the delivery of the system it is impossible to say if quality assurance was successful. Most problems show up after some time of using the system. However since the project teams breaks apart in most cases, the team members can not wait for feedback for a long time.
- Assessments of the software development process have to be done immediately after delivery. Otherwise people are not reachable any more or too many details are lost. However the assessment can not be completed without knowing about the acceptance of the software product at the customer's site.
- Increasing time between project end and an assessment fosters too optimistic memories. However no for users to experience the software inhibits qualified feedback.

Solution

Collect feedback after a short period of time after product delivery.

The evaluation process of the received feedback from the customer and the data from the project itself (project reports …) against the quality goals of the project usually is done in a so called post mortem analysis.
The feedback from the customer should be collected after some weeks of use (note in Fig. 8 that there is some time between start of operations and the post mortem analysis). Then the customer has experienced the good and the bad sides of the system.

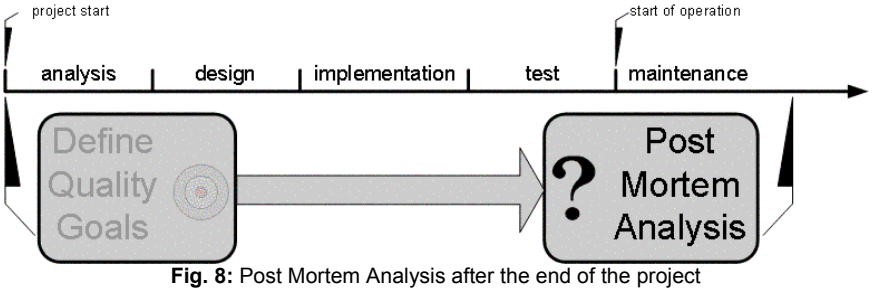


Fig. 8: Post Mortem Analysis after the end of the project

Resulting context

Post Mortem Analysis helps the software company to keep contact with the customers.
Lagged feedback improves the quality of the feedback of the customer and therefore the results of the post mortem analysis.
The lessons learned out of past projects can help to make project and QA plans better.

Related patterns

- User involvement: This is the last chance for the customer to tell about his impressions about the project (team).
- Evolution: Feedback is the necessary input for any improvements.
- Use Templates: The existing templates should be improved based on the feedback. New templates can be created, if activities or products are identified, which regularly caused problems.

Known Uses

- **XEROX™** does x.30 und x.90 interviews 30 and 90 days after the installation of a new system.

References

[HopeBeliefeWizardry] Markus Völter: *Hope, Belief and Wizardry - Three different perspectives on project management*, Europlop 2002.

[OrgPatterns] Jim Coplien, Neil Harrison: *Organizational Patterns*; http://www.easycomp.org/cgi-bin/OrgPatterns?BuildPrototypes.

[PSP] Humphrey W. S., *A Discipline for Software Engineering,* Addison Wesley, 1995.