

A pattern language for software quality assurance in small and medium enterprises – The foundation patterns

Armin Scherz, Wolfgang Zuser, Thomas Grechenig
Research Industrial Software Engineering
Vienna University of Technology
a.scherz@swt.tuwien.ac.at, wolfgang.zuser@rise.tuwien.ac.at,
grechenig@viveka.tuwien.ac.at

Introduction

Small and medium IT enterprises are forced to compete against large companies in many cases. It is an unequal competition. While SMEs are usually characterized by restrictions concerning budget and personnel, large companies have many possibilities concerning personnel selection and monetary flexibility.

A working software quality assurance is one of the key success factors for SMEs in this situation. The existing restrictions in SMEs do not allow a QA-department on its own. Usually they 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 IT projects in SMEs without specialized QA personnel. The patterns are divided into foundations patterns, which are abstract enough that they can also be used in other contexts, and action patterns, which are directly applicable to IT projects. The patterns can supply a minimal set of QA activities. Together with excellent software engineers they are sufficient to produce high quality software.

Roadmap

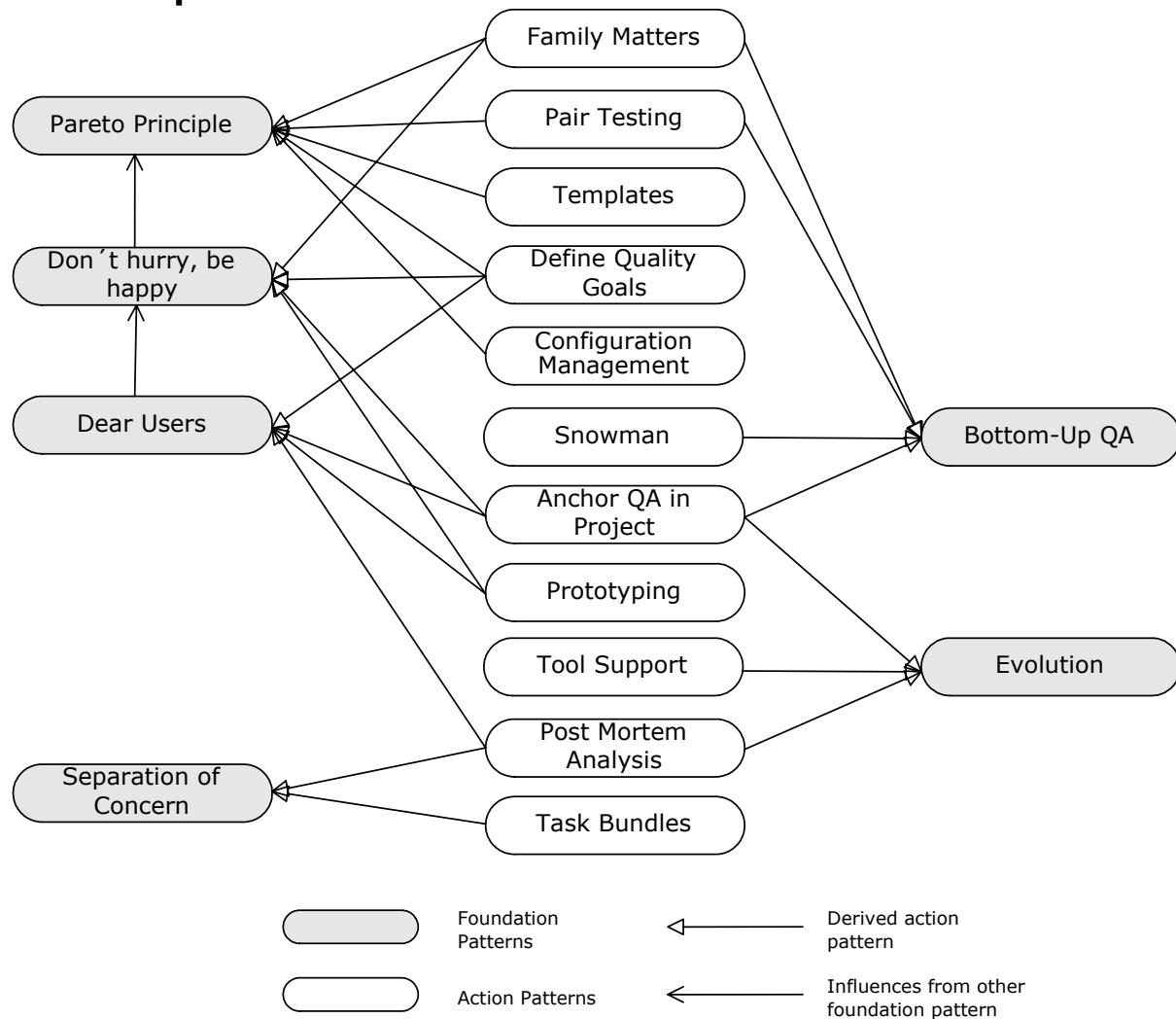


Fig. 1 Roadmap to a pattern language for software quality assurance in small and medium enterprises

Foundation patterns

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

The “Foundation patterns” are:

- Pareto Principle: A pattern for a good distribution of your available resources.
- Bottom-Up QA: Start your QA activities at each single developer.
- Don't hurry, be happy: Concentration on the early phases of software development should help you to achieve good quality software.
- Users Ambassador: A pattern about tight user involvement for an optimum of information exchange.
- Evolution: Build you QA system step by step.
- Separation of Concern: A pattern about how to distribute many concerns on few people.

Action patterns (not part of the submission)

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

The “Action patterns” are:

- Snowman: One approach for knowledge transfer in small teams.
- Tool Support: Manual operations are not effective.
- Templates: Helps saving effort and increases quality.
- Configuration Management: Even a few people need some help at integration of different modules.
- Anchor QA in Project: QA should be as important as implementation or testing.
- Define Quality Goals: Just in case you want to achieve quality it is a good idea to have quality goals.
- Prototyping: A useful method for user involvement.
- Family Matters: 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.
- Task Bundles: How to organize tasks in projects with few people effectively.
- Post Mortem Analysis: How to preserve experience in small companies.

General Forces

Small and medium enterprises are restricted by following forces:

- **Limited resources** (especially people and tools) allow only a **limited number of projects** with limited size. This means **low turnover**. However, few people cause **little bureaucracy** and make the company and projects more efficient and **increases the net gain**.
- Using **external services** for training, software quality assurance or software process improvement is **very expensive**. Many small companies depend on such services due the lack of know how and resources within the company. However, to pass on these activities may have a **negative impact** on the **software quality** and on the **skills of the developers** (on the long turn).
- **(External) Specialists** are able to do an **excellent job** in some selected tasks and are highly performing. Some **other tasks can not be covered** (usually quality assurance, interface design ...) at such a high level or not at all. **Many normal developers** can handle **most of the tasks** (single tasks can be assigned to single persons) at a **satisfactory** level.
- **Few people** can work at **full capacity all the time**. **For a lot of people** there is **not enough work** for all the time.

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.
Example	A short story illustration the context of the pattern.
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	Appearances of the pattern outside of this pattern language.
Related Patterns	Internal links to other patterns of the language.
Further Reading	External links to books or papers.

Foundation Patterns

1. *Pareto Principle*

Example

A plan for software testing specifies which tests on which parts of the system are necessary and possible. According to this plan the available testing resources (workstations, testers, and time) have to be used in an optimum way to achieve the best error finding rate possible. Due to the high complexity and the size of today's software systems it is expensive to completely test the whole system.



Fig. 2 Vilfredo Pareto (1848-1923)

Context

Typical software projects have a limited amount of available resources (e.g. project budget), which must be shared among the different activities during software development.

Problem

How do you distribute the available resources to get the best results?

Forces

- The available **resources** (time, developers, money, development-hardware, know-how) are **limited**. Most **quality assurance** activities (e.g. tests, reviews...) need **almost unlimited resources to achieve 100% of the desired result**.
- Factors with high influence help to **save time and costs and contribute much to the result**. Factors with low influence **may cause additional costs without contributing much to the result**.
- Customer and management want to achieve **maximal output with minimal input**.

Solution

Use a few resources for applying the input factors with the greatest impact first to achieve a great amount of the quality.

Use the remaining input factors only, if there are resources left or additional resources are provided. The typical ratio is about 20% to 80% (see Fig. 3). 20% of input factors influence 80% of the quality of the output, while 80% of the factors only contribute to 20% of the results quality.

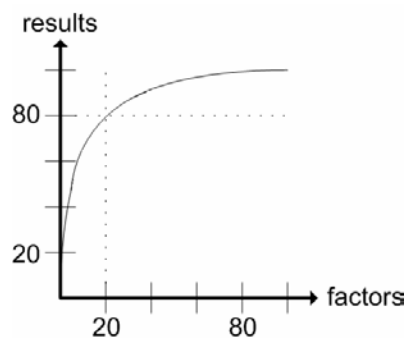


Fig. 3: Pareto Principle

Consequences

Applying the Pareto Principle will have following effects:

- Few resources (20%) are used for achieving most (80%) of the result. Depending on the project priorities the remaining resources selectively can be used for other tasks (more quality, more functionality ...).
- Finding the factors with high influence in your projects helps you to set the right priorities during the project not only for project planning but also for risk management and other important activities.
- Customer and management get 80% of the result with 20% of the resources. In a lot of cases 80% - quality is enough and you can save 80% of resources. But to achieve a perfect, 100%-result, you still have to do 100% of the work. In other words: The last 20% for a perfect result need 80% of the resources. That means that perfection is expensive. You have to be conscious of this when you decide if a product only needs 80% of quality or if you need a 100%-quality product.

This approach is only applicable for products, where it is reasonable to achieve not a 100% result. For system with a high demand on quality issues (e.g. safety critical systems), the pareto principle can help assigning the right priorities on tasks but can not be used for eliminating tasks.

Known Uses

- **Pareto Analysis** is a statistical method to identify the “vital few” 20% of input. For details of this technique see [KALI94].
- Steve McConnell uses the Pareto Principle to describe some characteristics of **error distribution**: 20% of the code contains 80% of the errors, 20% of the errors cause 80% of the costs.
- Barry Boehm uses the 80:20-rule for one his top 10 industrial software metrics in [BOEH87].
- In the **eBSS** project we decided to limit the tests to 20% of all classes which cover 80% of the critical functionality for the first test run. The result was representative enough for evaluating the overall systems quality.
- In the **WWS** project test priorities were defined for test planning and execution.

Related Patterns

- **Don't hurry, be happy**: Early phases have the highest influence on the final product. That's why one should concentrate on the early stages.
- **Templates**: To avoid too much bureaucracy one should focus on the most important templates, which cover 80% of the routine work with 20% of the input.
- **Configuration Management**: For small software projects it's enough to just do the central tasks of Configuration Management.
- **Define Quality Goals**: The different quality goals have different importance. Concentrate on the most important quality factors.
- **Pair Testing**: Find out which parts of the code are critical and concentrate on testing these parts.
- **Family Matters**: Reviewing is expensive. That's why only the most important documents can be reviewed.

Further Reading

[PARE97] Pareto V., *The New Theories of Economics*, Journal of Political Economy

Volume 5, pp. 485-502, 1897

[MCCO93] McConnell S.C., *Code Complete*, Microsoft Press, ISBN 1-55615-484-4, 1993

[JURA99] Juran J.M., Blanton G.A., *The Quality Control Handbook* 5th edn, New York, McGraw-Hill, ISBN: 007034003X, 1999.

[KALI94] Kaliszewski I., *Quantitative Pareto Analysis by Cone Separation Technique*, Kluwer Academic Publishers, ISBN: 0792394925, 1994.

[BOEH87] Boehm B., *Industrial Software Metrics Top 10 List*, IEEE Software, Volume 4, Number 5, pp.84-85, 1987

2. Bottom-Up QA

Example

In the ideal case there is a separate QA group within the company which performs quality tests and monitors the different QA activities. Independent quality management helps solving different problems (see: Forces). But small companies usually have no separate QA-group. This especially affects software testing and technical reviews: In small software companies software tests and technical reviews cannot be performed by a separate group. So the developers have to do the tests and reviews by themselves. This is a problematic situation because developers tend to overlook a lot of errors if they test their own code. This phenomenon is caused by subconscious attitudes towards the code (“I worked so hard on avoiding errors – so why should there be any errors left?”). This attitude leads to uncritical testing.



Fig. 4: Leafcutter ants at work without coordination from above.

Context

The system's quality is affected by the quality of every single component. The skills of different software engineers, who produce the different components, vary a lot ([DEMA87]). Testing and a lot of other QA-methods are best applicable if they are carried out by a separate QA-group (e.g. reviews, inspections, post mortem analysis). If a company lacks a separate QA-group it's not possible to assign all the QA-work to the usual developers without changing the QA-measures. An additional problem is that QA-activities which require a lot of coordinating work are not suitable for a project where developers have to do the quality management by their own.

Problem

How can you have QA without an independent QA-department?

Forces

- Separate QA-departments (or a single person in charge of QA) are expensive and therefore **should work at full capacity all the time**. Small companies have **not enough projects** to fully load such departments (or even a single person).
- **External specialists** can be engaged for doing the QA work. External specialists are **very expensive**.
- QA activities **need a lot of work**. Small projects have a **limited number of workers**.
- Quality control should be performed by an **independent instance**. In small companies there often is **no such instance, if there is no QA department**.
- Quality assurance for every single system component causes **a lot of effort**. QA for only some components may cause **lower quality**.

Solution

Motivate every software engineer to use QA methods in the small to ensure that his/her individual work is of high quality.

This approach results in better-quality components which compose a high-quality system. A metaphor for this approach is the way ants are organized. They have no hierarchical structure which could coordinate the single workers. But the sum of the individual efforts is enough to solve the problems of the ant empire (see Fig. 4).

If every developer uses quality assurance methods during his/her own work, the personal results get better. So there is not so much QA work left when the components are integrated.

Consequences

Doing QA bottom-up will have following consequences:

- You will not need an expensive QA department which is not loaded with work at its full capacity.
- You will not need external specialists or only for few QA tasks which can not be done by your developers beside their project work.
- The lack of a QA department does not allow an independent external control. Some degree of external control can be substituted by some concluding control by developers not participating in the project.
- You will have at least some QA for each component in your system.

Known Uses

- **Personal Software Process (PSP)**: PSP is a training program which enables software engineers to improve and verify their personal capabilities ([HUMP95]).
- **Wiegiers' SQA Team Member**: Karl E. Wiegiers suggests assigning the QA coordination to one team member. To assure the independence of QA, each group member is asked to play the SQA role on someone else's project ([WIEG93]).
- **Unit tests in XP** enable collective code ownership: "Any developer can change any line of code to add functionality, fix bugs, or refactor." Any developer is therefore in charge of the systems quality. Since each unit test is written by the programmer of the corresponding code under test, quality assurance starts at each programmer (and does not require further QA at all – believing to XP).

Related Patterns

- **Snowman:** To improve the individual abilities a training program is needed.
- **Anchor QA in Project:** If all project members know about QA principles they understand the need for QA. This helps motivating the developers for QA.
- **Pair Testing:** The test plan is affected by the absence of an independent QA department.
- **Family Matters:** One basic idea of reviews is that external experts find more errors than team members, because they have a fresh few on the problems which are well known within the team. Also in small projects this effect should be exploited.

Further Reading

[DEMA87] DeMarco T., Lister T., *Peopleware: Productive Projects and Teams*. New York: Dorset House Publishing, ISBN: 0932633439, 1987.

[HUMP95] Humphrey W. S., *A Discipline for Software Engineering*, Addison Wesley, ISBN 0-201-54610-8, 1995.

[WIEG93] Wiegers K. E., *Implementing Software Engineering In a Small Software Group*, Computer Language vol. 10, no. 6, June 1993.

3. Don't hurry, be happy

Example

A common problem in software development is to determine whether a software project has reached a status which allows the project team to enter the next project phase or not. This status depends on the quality of the development products and documents. Usually the decision isn't easy and needs some project experience. If you proceed to the next stage too early, there are still a lot of errors left which cause more errors in the next stages. But if you proceed too slowly you lose time and the project gets more expensive.

Context

The quality of the final software system is dependent of the quality of all artifacts (code, documents) which are produced during the development process. That is why successful software quality assurance has to influence all phases of software development.

Problem

On which phases during software development should you concentrate QA activities for achieving high software quality?

Forces

- Understanding and documenting the true requirements **correctly** in the first phases of the project **needs some time**. Not understanding the true requirements and **saving some time in the beginning** increases the risk of realizing wrong or misunderstood requirements causing **time losses at the end** of the project.
- **Time pressure** leads to careless error detection activities. But **undiscovered errors get into the next generation** of documents and products. They do not disappear by some miracle. Additionally these **errors cause new, even more significant, errors**.

- Early phases are characterized by **documents and models**. **Developers do not like** documents and models. They want to build their system. **Users and customers do not like documents and models**, they want to see their system working. **QA activities need documents and models** as reference for validation and verification activities.

Solution

Concentrate your software quality assurance on the early phases of software development.

These phases have the biggest impact on the final quality of the software system because a concentration on early phases pays off in the following phases (Fig. 5, Fig. 6).

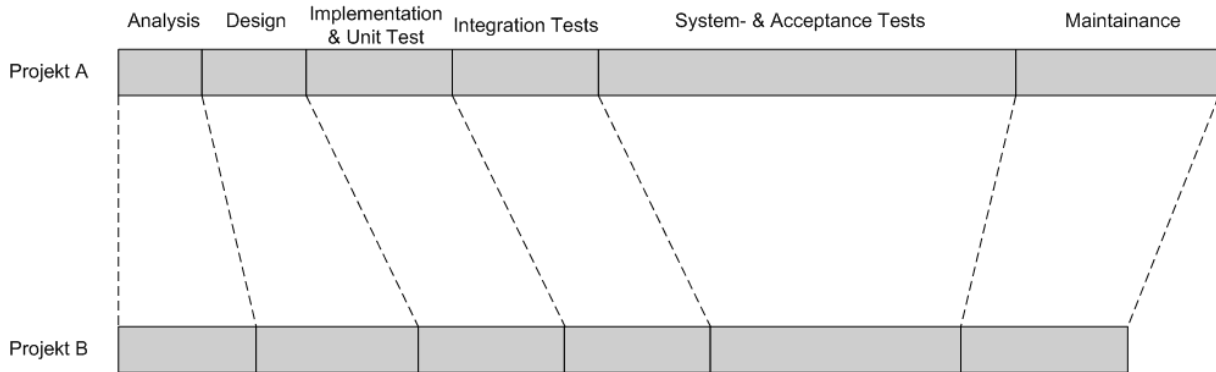


Fig. 5: Impact of concentration on early project phases ([SCHU92])

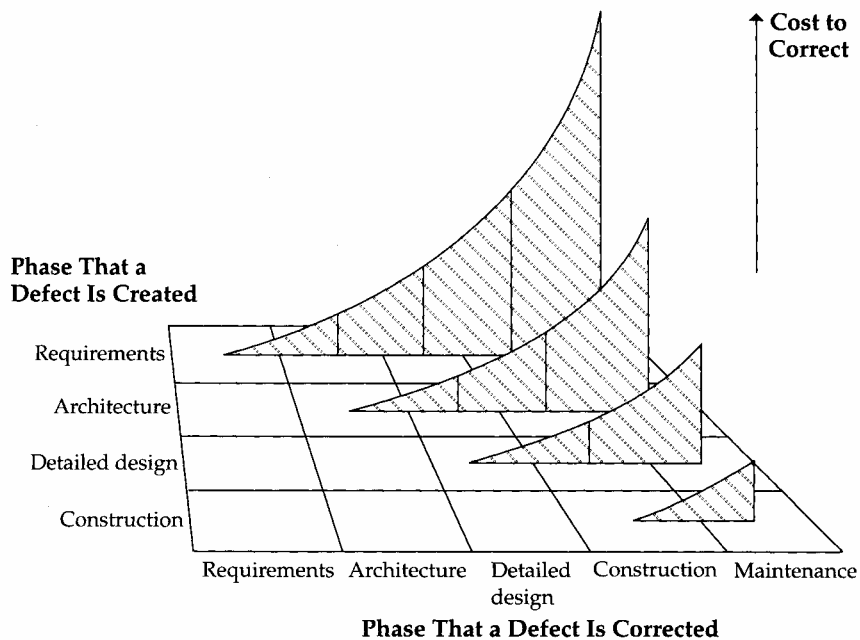


Fig. 6 Defect costs ([MCCO98])

Consequences

- A concentration on the first phases causes the project to proceed slower in the beginning. Usually the project benefits from this delay and is finished faster than without concentration on the first phases. Even though user change their mind about their requirements quite frequently (and therefore make some of your effort in the beginning at some later point of the project obsolete), these changes will not change the general vision of the system nor the most important requirements. Therefore

- most of the effort in the beginning will pay off directly through the whole project and
- some of the effort will pay off indirectly, because it is easier to discuss changes upon a well understood basis than upon no basis.
- Error propagation will be reduced to a low level.
- Concentration of QA activities does not necessarily mean only to produce documents and models. Prototypes can be used for eliciting requirements very effectively. Documenting the requirements (e.g. by the QA department itself) based on prototypes can be used as first step for validating the requirements.

The smaller a software project the smaller is the benefit of a concentration on the beginning. That is why one should be careful not to proceed too slowly in small software projects. Paradigms like XP (eXtreme Programming) are aware of this problem. That's why one XP paradigm is to start with implementation as soon as possible. At first sight this approach seems to conflict with the "Don't hurry, be happy" pattern. In fact the "Don't hurry, be happy"-attitude is kind of a counterpart to an early implementation start. The ideal schedule in small software projects is always a mixture of careful analysis to avoid development into the wrong direction on the one hand and courageous and effective progress (XP) on the other hand.

Known Uses

- IBM: „*An unpublished **IBM rule of thumb** for the relative costs to identify software defects: during design, 0.5; prior coding, 1; during coding, 1.5; prior to test, 10; during test, 60; in field use, 100*“ ([HUMP95], p. 275).
- **Focus on Requirements** ([WIEG93]). Carl E. Wiegers' advice is to achieve high quality through excellent requirements.
- **Steve McConnell** uses the pattern discussing the bad effects of defects not detected in early phases of software development [MCCO98].
- **Barry Boehm's** cost of Change Curve applies this pattern [BOEH76].

Related Patterns

- **Users Ambassador:** Requirements analysis is only possible if users are involved.
- **Pareto Principle:** Early phases have the highest influence on the final product. That's why, according "Pareto's principle" one should concentrate on the early stages.
- **Anchor QA in Project:** That's one of the challenges in early project stages.
- **Define Quality Goals:** The central task of QM during Analysis.
- **Prototyping:** Provides early results for the customer and the developers. Prototypes are also a compromise between fast and slow advance in the project.
- **Family Matters:** Reviews are a central tool to control quality in the beginning of a project.

Further Reading

[HUMP95] Humphrey W. S., *A Discipline for Software Engineering*, Addison Wesley, ISBN 0-201-54610-8, 1995.

[WIEG93] Wiegers K. E., *Implementing Software Engineering In a Small Software Group*, Computer Language vol. 10, no. 6, June 1993.

[SCHU92] Schulmeyer G. G., McManus J. I., *Handbook of Software Quality Assurance (Vnr Computer Library)*, 2nd edition, Van Nostrand Reinhold; ISBN: 0442007965, July 1992.

[MCCO98] McConnell St., *Software Project: Survival Guide*, Microsoft Press, 1998

[BOEH76] Software Engineering, IEEE Transactions on Computers, Dec. 1976

4. Users Ambassador

Example

There are a lot of different possible designs for a user interfaces. Different people like different designs for a lot of reasons. Some of these reasons are personal preferences, de facto standards in the application area or certain work-flows within the customer's organization. For the system developers it is very hard to find the right user interface design for the users' needs.

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 do you understand effectively what the user expects from the software system?

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. 7). 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.

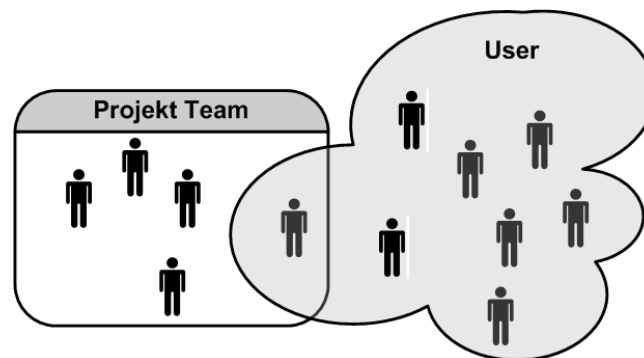


Fig. 7: User involvement in the software project

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.

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 ([WIEG93]).
- **Extreme Programming:** One of its rules is "The Customer is Always Available". Anyhow it does not distinguish between the customer and the users. ([XP]).
- 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.

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.
- **Anchor QA in Project:** Also the user has to understand the need for QA.
- **Define Quality Goals:** The customers view is decisive for the quality goals.
- **Prototyping:** Is a tool for user involvement, because the prototypes can be presented to the user.
- **Post Mortem Analysis:** Only the user can say if his needs are fulfilled by the system.

Further Reading

[WIEG93] Wiegers K. E., *Implementing Software Engineering In a Small Software Group*, Computer Language vol. 10, no. 6, June 1993.

[JURA99] Juran J.M., Blanton G.A., *The Quality Control Handbook* 5th edn, New York, McGraw-Hill, ISBN: 007034003X, 1999.

[XP] <http://www.extremeprogramming.org/rules/customer.html>

5. Evolution

Example

Imagine that you want to introduce Software Quality Assurance in a small Software development company. How should the QA plan be designed to pay attention to the special needs in the company? It's very unlikely that a QA plan which works in another (larger) company is applicable in your company without major changes. So you have to develop your own QA approach.

Context

Software quality assurance is not an activity that is performed independent from software development itself. In fact QA has a lot of influence on the software development process. That's why the chosen QA plan has to fit in the whole software development process used in a certain company. Otherwise QA activities become cumbersome and will not have the desired effect on the systems quality.

Problem

How do you find a suitable Software Quality Assurance system for your software company?

Forces

- **Determining the right QA system** will take some time causing **some costs**. **No or a inappropriate QA system** can cause **even higher costs** due the lack of quality in your projects.
- An existing **not perfectly fitting** QA system (from another company, from literature or derived from a standard) can be used **straight away**. A QA system from scratch **perfectly fitting** to your software engineering process will take a **bunch of time** (missing some projects meanwhile).

- In complex fields like software development **experience** is very important. But **Software Engineering is a young** discipline. That's why there is not much reusable knowledge available and you have to find your own solutions in a lot of areas.

Solution

Start with selected good fitting QA practices and improve and extend them from project to project.

Take an existing QA approach, downscale it to your usual project size and select good fitting practices according to the existing software engineering process and work habits. Use an evolutionary process to continuously improve and extend the QA system from project to project.

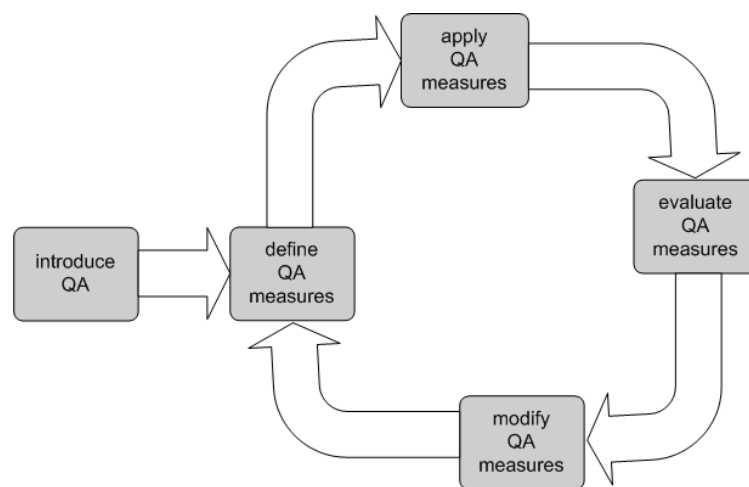


Fig. 8: Evolutionary Improvement of QA

Fig. 8 illustrates that process. The first step is to introduce QA in the software company. After that you have to define an initial set of quality assurance measures. These activities are performed during a project. After the project you will have learned a lot about the chosen measures. So you can modify them and get a new definition of QA measures. After some iteration the Quality assurance plan will be appropriate for the specific needs within the software company.

Consequences

Using an evolutionary approach towards the QA system will have the following consequences:

- You avoid the costs of low quality even in the first iteration as you have a QA system right from the start. You also avoid the high costs of discussing the “optimal QA system” which you will never find.
- You do not miss projects and have a QA system in place right away.
- Experience with any QA system and evolutionary improvement is better than endlessly discussing about something you never implement and never test. Therefore it's better to take the second best solution and improve it instead of never coming to an end of discussions.

Known Uses

- **IEEE 730-2002:** By using a model like the one just explained you can develop your own QA plan beginning with the standard IEEE 730-2002. Standard IEEE 730-2002 takes this

possibility into account by providing a history-field for former quality assurance plan versions ([IEEE97]).

- **Deming circle:** The Deming circle is an evolutionary approach which uses “plan-do-act-check” iterations. ([DEMI86]).
- The **evolutionary introduction of quality assurance activities** in the OeKB was an implicit requirement for QA in the eBSS project.

Related Patterns

- **Tool Support:** The set of tools which can be used in a project can also be changed in an evolutionary way.
- **Templates:** The set of supporting templates behaves in the same way as the set of tools.
- **Anchor QA in Project:** During a QM Kick-off the changes in the QA plan can be discussed.
- **Post Mortem Analysis:** The foundation for evolutionary improvement is the knowledge about mistakes and successes.

Further Reading

[DEMI86] Deming W.E., *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, ISBN: 0262541157, 1986.

[IEEE97] *IEEE Software Engineering Standards Collection, 1997 Edition*. Los Alamitos, Calif.: IEEE Computer Society Press, ISBN: 0738115630, 1997.

6. Separation of Concerns

Example

After the project software engineers want to learn how to make things better in the next project (with the same customer). At which point of time should the data be collected not causing too much work for the few developers in the project?

The process data of a project should be collected immediately after the project ends. Together with this some rudimentary metrics about the projects can be collected and evaluated as well (task bundle 1). Customer feedback should be collected later. The project post mortem can be done after finishing the customer feedback considering the process data collected already earlier (task bundle 2).

Context

Elaborated process models consist of many diverse concerns (e.g. tasks, roles, organizational units ...) which affect many diverse roles within the project. In small projects the concerns have to be bundled to few persons. This is especially true if some roles do not exist in small projects (e.g. quality assurance).

Problem

How do you group many concerns for execution by few persons?

Forces

- Many concerns have to be assigned to few persons. That's why **every person has to be in charge for many concerns**. But for one single person it is **hard to think about many different concerns in parallel**.

- Too **many separated concerns** cause much **overhead** (due communication, organizational load ...) and need a lot of resources. Too **few separated concerns** result in huge bundles, which **overload a single people**.

Solution

Form bundles of concerns around critical concerns, which can be executed by single persons a one point of time.

Find the critical concerns (cannot be moved anyway, other concern depend on them) first and put them as center of bundles. Add to the center concerns, which are not critical and can be moved. Try to put concerns around the center, which are similar to the center concern. Fig. 9 shows an example for a possible clustering of concerns into bundles.

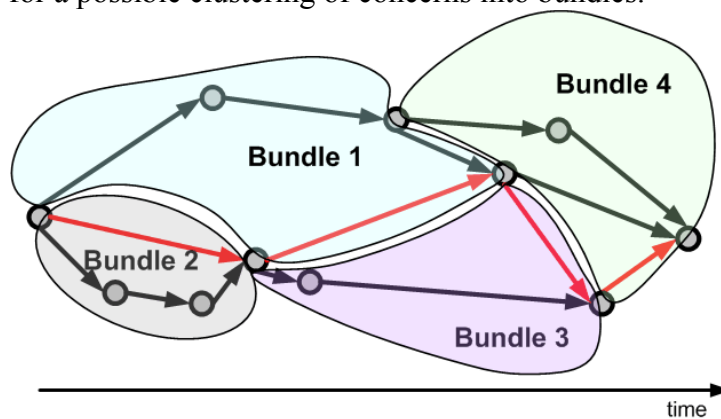


Fig. 9: Example for Clustering of Concerns into Bundles

In the graph every edge represents a task and every node represents a milestone (e.g.: project start, project end, finishing of a document...). If an edge starts at a certain node, the task can only start after the milestone. If an edge ends in a certain node, the milestone can only be reached as a result of the task. The red path is the critical path (a delay on this path also delays the entire project), consisting of the critical tasks. Each of these critical tasks is the center task of one task bundle. The other tasks are distributed among the different bundles according to their causal relationship with the center task and to the size of the bundles.

Consequences

- The concern bundles should minimize parallel concerns in one place (due the textual correspondence of all concerns in a bundle).
- Overhead can be reduced by the concern bundles. Anyhow the expected work load of each bundle must consider the existing projects work load of the developer who will be in charge for execution of the concern bundle.

Known Uses

- Ancient (?) regimes used the pattern for administration of their empires.
- “The technique of mastering complexity has been known since ancient times: ‘divide et impera’ (divide and rule).” (*E. Dijkstra*)

Related Patterns

- **Bottom-Up QA**: The design of the bundles influences how independent the work of the engineers is from the work of others.
- **Don't hurry, be happy**: The important tasks in the beginning of a project can be bundled together.

- **Dear Customer:** Tasks which require interaction with the customer can be bundled together.
- **Configuration Management:** can be one separate bundle.
- **Define Quality Goals** and **Post Mortem Analysis** should be in one bundle because the defined goals are tested in the end.
- **Task Bundles:** A derived action pattern of this pattern where the concern of interest is tasks in a project.
- **Divide and Conquer:** part of Organizational Pattern by Jim Coplien [COPL95].

Further Reading

[COPL95] <http://www1.bell-labs.com/user/cope/Patterns/Process/section33.html>