Software Engineering Practices: An Auditor's Perspective

Krzysztof SACHA[1]
Warsaw University of Technology, Poland

Abstract. This report details part of the results of five software audits that were done to evaluate various aspects of the quality in five very big software projects. One result of our work was a method for software quality evaluation, which is described in detail elsewhere. Another result was a review of the software engineering practices and methods that were used throughout those projects by the development companies. The paper presents a survey of these practices and tries to answer the question which software development paradigms, processes and methods are used in the software industry and which of them can contribute to the final success of the project more than the others.

Keywords. Software quality, quality evaluation, software engineering, software development

Introduction

Software systems are used in many application areas in which a malfunction of the system can be a source of serious losses or disturbances to the functioning of the society. Examples of such application areas are not only command and control systems, but also public administration, social insurance or post delivery services. The quality of services offered in these areas depends heavily on the quality and dependability of software systems that support the functioning of the appropriate public or private organizations (service providers).

Software development processes consist of a selection of methods and tools that vary from project to project. It is interesting to know which of the methods described in the literature are used in everyday practice and how do they work. The question is vital, as research shows that the success ratio of the software projects is low, when comparing to other branches of engineering. According to The Chaos study [1, 2] of the Standish Group, in 1994 only 16% of projects were completed on-time and on budget, 53% were challenged, i.e. completed but over-budget and over time estimates and 31% of projects were cancelled. Ten years later The Chaos study reported 29% of successful projects, 53% challenged, and 18% cancelled. Despite a significant improvement (Table 1), the success ratio of the software projects is still far from satisfaction. Similar data can also be found in American Programmer [3].

[1] Krzysztof Sacha: Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warszawa, Poland; E-mail: k.sacha@ia.pw.edu.pl

Table 1. Project resolution (source: The Chaos Study, The Standish Group)

Year of research[*]	**Successful**	**Challenged**	**Cancelled**
1994	16%	53%	31%
2000	28%	49%	23%
2002	34%	51%	15%
2004	29%	53%	18%

[*] The data were published one year later

The methodology of The Chaos studies was based on questionnaires and interviews responded to by IT executive managers of over 50,000 IT projects (during 12 years of research), with the most important part aimed at discovering the key factors of a project success or failure. The list of the most important factors that cause projects to succeed changed over the years, and in 2004 was the following [2]: User involvement, executive management support, clear business objectives, minimized scope, agile process, experienced project manager, formal methodology, and standard tools and infrastructure. Those results reflected a managerial point of view. More technically-oriented aspects of the software processes and the development methods were outside the scope of these surveys.

This paper relies on a different methodology. The results presented in this report are based on the observations that I did during a series of audits and quality evaluations of five big software development projects ($300 million the biggest) that were conducted for public administration in Poland in the last five years. During those evaluations, the evaluating team was positioned just between the customer and the development company, and dealt with the deliverables of the projects. Therefore, the research came closer to the technical level and was aimed at identification and evaluation of processes and methods that were used by software developers in their work.

The goal of this paper is to summarize our observations pertaining to the software processes and the development methods that are used by big development companies. Because the contract awards for building the systems considered in the paper were made through a competitive bid process with a participation of huge global companies, we believe that the results of our observations are representative to the contemporary IT market. According to our contracts we are not allowed to describe the details of particular systems and the development of these systems. Therefore the paper does not present a case study, but is a survey of practices that are used.

The main body of the paper is divided into five parts, the first of which provides the reader with an overview of the characteristics and the context of projects and systems that were subject to our evaluation. The results of projects considered in this paper are described, and related to major success/failure factors of The Chaos study in Section 2. Development processes and methods used throughout these projects are described in Section 3, and the evaluation of several process and product metrics is given in Section 4. Section 5 refers to a specific aspect of the development that is best visible to the customer of an IT contract, i.e. acceptance testing. Final remarks and statistics are gathered in Conclusions.

1. Evaluated Projects

The projects under evaluation were typical on the IT market: The developed systems were going to deliver common services, the development contracts were awarded through a competitive bid process and the development companies applied well known, yet different, development methods and tools.

All the development companies were big and had strong market position. One was a branch of a huge global company headquartered in US, while the other four were big national companies ($500 million annual revenue the biggest) with strong international cooperation. Therefore the observations described in this paper can be considered representative of the global software development market.

The systems covered in this paper are the following:

- Integrated Information System for Social Insurance Institution that supports individual accounts of all employees and all employers in the country.
- Integrated Administration and Control System (IACS) that supports direct payments within the European Union common agriculture policy.
- Common Agriculture Policy System (CAPS) that supports intervention purchase, storage and sale within the European Union common agriculture policy.
- Computerized Postal System that controls the process of transferring and tracking of registered shipments across the country.
- District Level Elections Support System.

All of those systems cover the area of the entire country and influence the living conditions of millions of people. Therefore, they fall into the category of big or very big systems. A set of attributes to characterize the size of the information systems considered in this paper is given in Table 2.

Table 2. Size attributes of the information systems considered in the paper

Number of	**Social Insurance**	**IACS**	**CAPS**	**Postal System**	**Elections**
Accounts	17 000 000	2 300 000	800 000		
Documents per year	300 000 000	12 000 000	1 000 000	540 000 000*	
Users	23 000	8 500	500	2 500	6 000
Sites	300	330	17	17	5 500

* 1 500 000 registered shipments per day

The attributes given above are not quite comparable. Nevertheless, we believe that they characterize the size of an information system much better than traditional measures of lines-of-code or function points. In the first three cases the number of documents per year means the number of real documents that are sent by the customers and that must be scanned and processed by the system. In case of the postal system this number refers to the bulk of registered shipments that must be handled. The number of sites equals to the number of local branches of the organization, each of which has usually multiple users. However, in case of Election System those sites contained in most cases a single user only.

In the final result, two of five systems considered in this paper were built within the budget and schedule, in two cases the schedules were not met but the core elements of the systems were deployed with an acceptable delay. One project failed completely and did not provide the required services at the deadline. These statistics (40% of success, 40% of challenged and 20% of failed projects) look a bit better than the data of the Chaos study shown in the last two rows of Table 1.

The processes of building systems listed in Table 2 were subject to a number of evaluations made on behalf of the customers or of the state institutions of control. One of the evaluators was the Software Engineering Group at Warsaw University of Technology. The evaluation took place in the years of 2002-2004 and was based on an in depth analysis of the deliverables of the particular development activities. The primary goal of our evaluation was the assessment of the expected quality of software under development in one case, and the evaluation of the quality of phases of the software process in three cases; in one case we tried to find the reasons of a catastrophe.

The typical audit methodology, focused on the quality evaluation of the development process [4, 5], could offer only a limited set of means for the quality evaluation of the software product. Software quality evaluation methods described in the literature [6-10] represented the software development organization point of view. Neither of those methods fitted well into the environment of a software quality evaluation, which was done on behalf of external authorities by people from the outside of the development company. One difference was such that we had only limited access to the project data, and the quality evaluation had to be based on an evaluation of the deliverables of the software process that had been enumerated in the contract. Another difference was such that we had no historical data of the manufacturer related to a set of similar projects. Therefore we had to develop a new method, which was based on a modification to the GQM measurement model [6]. A detailed description of the methodology that was developed by us for the purpose of evaluation can be found in [11, 12].

2. Project Success/Failure Factors

It is not easy to isolate and evaluate the influence of the key success factors, identified in The Chaos [1] and cited in the Introduction, on the final result of a project. An attempt of such an evaluation is given below.

(1) User involvement and (2) executive management support were high in all but one project considered in this paper. The customer organizations that contracted the systems created special departments to help the development companies and to supervise the project. In one case user feedback was missing; this project failed.

(3) Business objectives were clear in four of five cases: Because of a change in legal regulations, the customer organizations could not function without a new support system any more. These four systems were built and put into operation. In one case the system was not indispensable for the customer organization; this project failed.

(4) Minimized scope means a decomposition of one huge project into a series of smaller projects, each of which can be completed within a shorter period of time and smaller budget. Such a type of project planning was applied in two cases, however, in a different way. In one case a huge centralized system was functionally decomposed into

a set of independent subsystems and modules coupled through a common database. The development of the system was then divided into a series of projects, each of which was restricted in scope to a subset of modules. This project has been delayed, but went ahead despite significant evolution of the requirements.

In the second case a project of an inherently distributed system was decomposed into two completely separate projects. One of them covered full functionality of a single business site, while the second project, started after full completion of the first one, covered the cooperation between the business sites. Both of these subprojects were finished within time and budget.

(5) Agile process. There is confusion in understanding agility in The Chaos study. None of the projects considered in this paper used an agile method, e.g. Scrum or XP [13]. Public systems are contracted through a competitive bid process, which requires a complete requirements specification available at the very beginning. Also the deadline, the price and the number of iterations are always written into the contract. None of these data is available at the beginning of an agile project. Therefore, I can hardly imagine the use of an agile method to contracting and building a public system.

However, if one identifies agility with iterativeness of the development process, then two of five projects were conducted this way. Both of these projects were finished within time and budget.

(6) Experienced project manager. In one case a system was built by a consortium of a few independent companies, with no hierarchical dependencies defined between them. The stakeholders discussed and agreed upon the schedule and the scope of tasks performed by the development teams. This way a sort of collaboration management was implemented, with no single project manager on top of the project structure. This project failed. In the other cases project management responsibilities were clearly defined and the results of those projects were much better.

(7) Strict methodology, though not very formal, was used in four of the projects. All of them were ultimately completed. In one case no strict methodology was used. This project failed.

(8) Standard software tools and infrastructure can resolve many technological problems and allow the development team to concentrate on business aspects of the application. In all but one project described in this paper, standard middleware was used as the main integrating keystone of the application. In one case a proprietary middleware package was used. This project failed, and an improper functioning, or improper usage, of this tool contributed to the defeat of the project.

There is one key factor missing in the list of 2004, which was present at the earlier editions of the Chaos study. This is the clearness and stability of the requirements specification. In two projects considered in this paper the requirements were stable; these projects were finished successfully. In two other cases the requirements varied. Both of those projects were delayed and over budget.

A detailed description of the final results of five projects considered in this paper (Table 2) can be summarized as follows.

Integrated Information System for Social Insurance Institution has not been completed within the schedule and budget. In fact, it is still being built, with the time overrun over 100%. However, the delay cannot be attributed to the methods that were applied during the project, but rather to the changes in the external environment of the project. It was clear from the very beginning that a powerful, quite new information system was indispensable to support the implementation of a very general reform of the rules of social insurance system in the country. The starting day of the reform was fixed by the government. A feasibility study of the system had been done, and the date at which the development had to start (in order to have the system ready at the date of reform) was known. However, the legislation process was late and when the development-start-date arrived, the necessary legal acts had still not been passed by the parliament. At that point in time the decision either to start the development or not, created the following risks:

- If they waited for the legislation, the time remaining for development would shorten dramatically and the deadline could not be met.
- If they started the development process immediately, the risk appeared of implementing requirements that differed from those ultimately present in the legislation act.

There was no good answer to resolve this dilemma. In our case the project was started on the basis of a draft version of the appropriate acts. Unfortunately, a parliamentary election arrived and the new government changed the acts and the requirements of the information system significantly. An annex to the contract was signed nearly one year after the development had been started and three years before the expected release of the system.

The final result of the project was not catastrophic, however. The core elements of the system were deployed and started working five months after the deadline. Due to the one year clearance of the social insurance payments, the delay appeared not essential to the success of the entire insurance reform. The legislation pertaining to social insurance is still evolving and the remaining modules of the system are still being built. The development used an Oracle-based structured method [14] and tools.

The story of IACS system was a bit similar. The system had to be built because of the accession of Poland to the European Union (EU). The date of the accession was agreed upon, but very detailed negotiations related to the Polish benefits of the common agriculture policy of EU lasted nearly to that date. As result, the development of IACS started on the basis of a draft version of the agreement. When the final agreement appeared different, an annex that changed the requirements was signed two and a half year after signing the original contract. The core elements of the modified system were released about half a year later, just in time to enable farmers to benefit from the common agriculture policy. However, a few auxiliary elements of the system were built much later. The system was built using RUP-based object-oriented methods [15].

Two systems: CAPS and Post Delivery Support System were developed in time and within the budget. Both of the two had well established requirements specifications that did not change during the development process. The development processes relied on Oracle-based methods and tools, however, use case specification was also created in one of these two projects.

The District Level Elections Support System crashed at the date of election. Investigation showed the lack of proper project management, the lack of sound methodology, and significant technological problems.

The lesson, which I learned from the above stories, is such that the key factors of success are user feedback, competent project management and a stable requirements specification. If the requirements evolve, then restricting the scope of the project and adding a dose of agility to the development process can create a good base to cope with the problem. The development methods are less important, provided that a certain level of technology competence is preserved. Such a conclusion matches quite well the conclusions of the Chaos studies [1] cited in the Introduction.

3. Software Processes and Development Methods

There are two major approaches to software development and two groups of methods that are currently used by the development companies on the IT market: Structured approach and object-oriented approach. The methods of both groups can be used within the framework of various software processes. Two of these processes that dominate nowadays in the industrial practice are waterfall model [14], and incremental and iterative RUP software process [15].

Software systems that are considered in this paper were created using various combinations of a software process and a development method. We identified both of these two constituents of the development process using the following metrics [12]:

- A list of methods declared in the contract and in the analytical specification.
- A mapping from the steps of the software process into the set of methods.
- A list of artifacts and a mapping from the set of artifacts to the set of methods.
- Qualitative evaluation of the deliverables of the particular steps of the process.

The results of the evaluation are shown in Table 3. Two projects were conducted using structured methods and tools, one project relied on an object-oriented methodology and in one case a mixture of methods was used. This data is in quite a good correlation with data of The Chaos study, which reported that 70% of projects developed from scratch in 2000 used structured methods and languages, while the other 30% was based on object-oriented methods and models.

Table 3. Processes and methods used in the development of software

	Structured methods	**Object-oriented methods**	***ad hoc***
Waterfall process	2.5*	0.5*	
RUP process		1	
ad hoc			1

* One project started with an object-oriented use case model, but was continued using structured methods.

The use of the waterfall model did not necessarily mean that the entire system was developed, implemented and deployed within a single sequence of consecutive steps. On the contrary, the system under development was usually decomposed into a set of functionally independent subsystems that were built independently of others. The system could then be integrated by the manufacturer and deployed at the customer's site in one step (as an entity). However, it could also be constructed incrementally, with particular subsystems created and deployed within separate runs of the waterfall process.

The initial requirements statement, which began the development of systems considered in this paper, consisted mainly of legal acts passed by the national parliament or by the European Union, accompanied by several business demands and constraints. In all cases the requirements analysis began by doing the context analysis, which led to a definition of the context schema that documented the external systems, organizations and users, and the required inflows and outflows of the developed system. The expected size and frequency of those inflows and outflows were estimated. The values of those estimates corresponded to the number of 'documents per year' in Table 2.

The kind of methods that were used to perform the analytical activities, or steps, varied from project to project. We identified and evaluated those methods using a set of metrics, which can be exemplified by the following samples [12]:

- A list of methods declared for the project.
- A mapping from the steps of the software process into the set of methods.
- A mapping from the set of user documents and reports identified in the acts to the set of inflows and outflows.
- An evaluation of the analytical products.

The analysis of the required behaviour of systems under development was done in two of five cases using object-oriented use case method, while in another two cases a hierarchy of functions was built. Data structures were modelled at this stage of development using entity-relationship diagram notation (ERD) or class diagrams created from the conceptual perspective. In one case the requirements analysis was performed intuitively, without being specified in any formal document.

Detailed analysis and design relied in three cases on a principle of structured functional decomposition. The initial requirements statement was subject to a critical requirements analysis, which led to a multi-level hierarchy of functions. The flows of data between the functions at each level of the hierarchy were defined and documented by means of data flow diagrams. The structure of data that was stored and passed within the system was modelled using entity-relationship diagrams. The processing assigned to each particular function was documented by means of flowcharts and textual specifications, accompanied by paper-based prototypes of the user interface (screenshots). Program structure and data base structure were derived from the above models using Oracle-based methods and tools [14].

Object-oriented analysis and design was based on the RUP methodology. First, the use case method was applied within a two-step process. In the first step business actors and procedures were identified, and the scenarios together with the pre- and post-conditions of those procedures were defined and documented. In the second step the definitions of actors were refined, and the user functions that were to be implemented by the system were derived and specified. The specification of a user function included a set of alternative scenarios, a definition of exceptions and exceptional actions, and the conditions to start and stop each particular function. The structure of data that was identified within the application domain was modelled using class diagram notation, and the behaviour of the most important classes was described by means of state transition diagrams.

Then, the logical structure of the application was designed using patterns [16] and documenting the results by means of class and interaction diagrams. Finally, physical components were defined and implemented. According to our observations the dominating implementation languages were SQL, Java and C++.

4. Evaluation of Methods

It was a superficial similarity between the final results of the analysis done by means of structured and object-oriented methods. In both cases a set of functions was defined, accompanied by a set of ERD or class diagrams. There was, however, one big difference between the two.

Hierarchy of functions and data flow diagrams resulted from a functional decomposition of the required processing. The top level functions within the hierarchy were nearly independent, as they referred to different business processes at the customer organization. The second-level functions had also very limited interplay, as they referred to different aspects and procedures within a given business process. The hierarchy of functions was then converted into the hierarchy of subsystems and modules, in which the top-level functions became subsystems that were developed independently, and the second level functions became modules of those subsystems. An advantage of such a development process was a good traceability of the design to the analysis. A disadvantage was such that the functions did not correspond directly to the business procedures at the customer organization and were not very useful in defining the acceptance testing scenarios.

The set of user functions, identified by means of the use case method, did not create any hierarchical structure. Such a flat and huge set of functions (about five hundreds in IACS) was nearly useless for the design purposes. Instead, a class model was developed with several thousands of classes. The classes were packed into packages that had very little to do with the initial user functions. As result, traceability from the design to the analysis was poor, and we found the verification of the design with respect to the analysis very difficult.

The advantages of using the use case method were: Completeness of the functional requirements and direct support of the acceptance testing process. The analytical artefacts that were created consisted of:

- Use case specifications and preliminary data model.
- A working prototype of the user interface.
- Preliminary test plan, closely related to the use case scenarios.

In this package the use case scenarios defined precisely the desired behaviour of the software, the prototype enabled the user to play with the (non-existing yet) software, and the test scenarios defined the verification method of the requirements. Usually, test scenarios corresponded directly to the respective use case scenarios. Preliminary data model (class diagrams) created a bridge towards future design activities.

Our evaluation of the quality of **functional** requirements specification in structured as well as object-oriented version was, in general, positive. Unfortunately, this positive evaluation did not spread out on the area of **non-functional** requirements. Performance requirements were, in general, not stated clearly during the analysis. Sometimes, an estimation of the number and the volume of input documents were given. Quantitative metrics, like response time or throughput measured in transactions per time unit [17], appeared in the specification of one system only. Instead, arbitrary (usually high) requirements for the performance of hardware were sometimes formulated.

Security requirements were described in an extensive, but qualitative and untestable way. A typical requirement was such that the system should use "*the most effective and up to date tools in order to guarantee perfect protection of data and other resources*". The position and length of the security requirements showed, however, that the customers were aware of the threat and were willing to spend money on the protection mechanisms.

The only kind of non-functional requirements that was stated in a clear and testable way was availability. The following metrics were used to define the required reliability and availability of the developed software systems:

- The percentage of time during a year that the system services must be available.
- The maximum time for recovery after a crash.
- The minimum period of time, within which a local server must provide the required functionality after a break to the communication links to a central server.
- The ability of process migration in case of hardware break down.
- The ability of re-running all the transactions that were lost as result of hardware break down.

The first three metrics were written into a service contract between the customer and the manufacturer of software. Test cases to verify the last two metrics were built into the test plans of the acceptance testing phase.

Comparing the quality of the four systems and their development processes, we did not observe definite superiority of one approach over the other. However, any kind of methodology worked better than *ad hoc* development that did not adhere to any method or standard and eventually led the project to a total collapse.

A surprising observation was instability of the development progress exposed by the RUP process. The development was driven by a use case model, which was created in the elaboration phase, and then used to plan the incremental development of the software in the construction phase. The construction phase started with the core requirements (a set of core use cases), and proceeded in such a way that the consecutive increments added functionality to the previously developed part of the software. It was said [15, 18] that this should lead to a stable architecture and modules.

In order to measure the stability of the development progress we counted the files of the source code that had been issued in the consecutive iterations, and compared the size of files that had the same name. If names and sizes of two files were the same, we assumed that the code in those files had not been changed. If the names were the same, but the sizes were different, we assumed that the code was retained with modification. Then, we calculated the following metrics:

- The percentage of files that were retained without modification in the next consecutive iteration.
- The percentage of files that were retained without modification in the final product.
- The percentage of files that were retained, but modified, in the final product.

The values of metrics, calculated for two components (subsystems) of the IACS system are shown in Figures 1 through 3. One of these components was responsible for collecting submissions and handling direct payments within the European Union common agriculture policy. The other component was responsible for the identification and registration of bovine animals. (There were also other components in IACS, such as: geographical information subsystem, accounting component, farm and animals data bases.) The characteristics of code of the two components are given in Table 4.

Table 4. Characteristics of the sample components of IACS

Characteristic	**Payments**	**Animals**
Number of classes	1 879	2 370
Number of lines of code	277 948	288 873
Number of lines of comment	82 343	60 410
A relation of comments to code	29.63%	20.91%
Percentage of classes with the relation of comments to code less than 10%	4.00%	33.50%
Percentage of classes with no comments	0.11%	12.57%
Percentage of classes with cyclomatic complexity [19] of a method ≥ 10	6.44%	6.82%
Percentage of classes with cyclomatic complexity [19] of a method ≥ 20	1.54%	1.54%

The stability metrics showed that the integration of code after a subsequent iteration required usually deep modification to that part of software that had been constructed and released earlier. The scope of changes to the code, which we measured between two consecutive increments, exceeded in average 40% of the total size of the existing code (Figure 1). Percentage of code that was retained without modification in the final product increased from iteration to iteration, but was well below 50% in the first half of the development process.

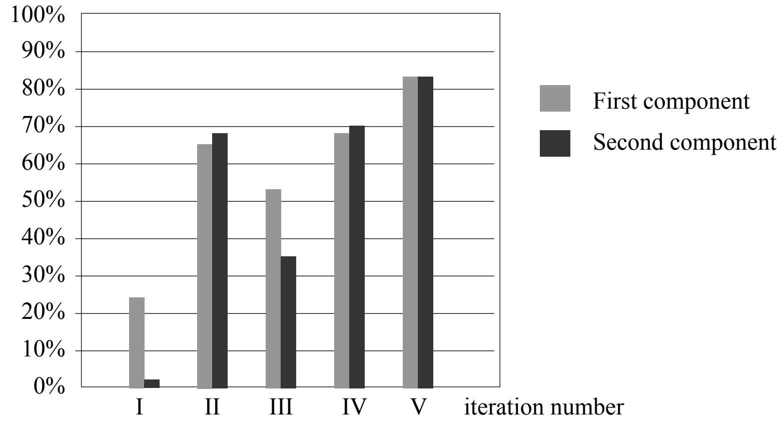

Figure 1. Percentage of code retained without modification in the next iteration

One explanation of this phenomenon is such that the method was not clearly understood by the developers and was misused. The other one is such that the partitioning of the developed software system driven by the use cases violated the rules of modularization: A particular increment of software did not constitute an internally consistent module with relatively weak interfaces to its environment. Instead, just the opposite was true, and the subsequent increments that were added within the loop of the construction phase, were strongly interrelated to the previously constructed part of software. The integration of such strongly related components imposed huge refactoring of the existing code.

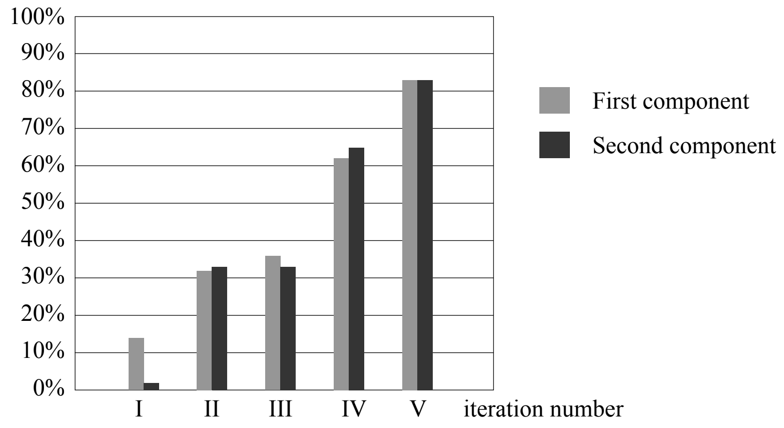

Figure 2. Percentage of code (issued in an iteration) retained in the final product

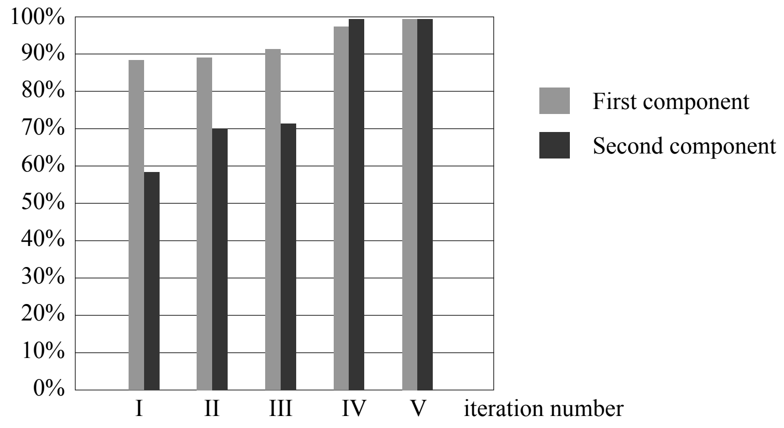

Figure 3. Percentage of files retained with modification in the final product

A major problem, which we observed in relation to such a development practice, pertained to the acceptance testing that was done after each iteration. The modifications to the code that had been released earlier made the development process unstable in that some errors that had appeared and had been fixed in an earlier version of the software, re-appeared again in a subsequent release of the same software component. Such a phenomenon was particularly disappointing for the customer, who wanted to use the existing part of the system that had been released in the latest iteration.

A minor problem was the scope of verification that was needed after each iteration of the software construction process. Because the modifications affected all kinds of development artefacts: Analytical, design and code, the same artefact, e.g. a sequence diagram, had several versions that had to be evaluated many times in a sequence.

5. Acceptance Testing

All the weak and strong points of the requirements specification, described in Section 4, were reflected in the artifacts prepared for acceptance testing. The well defined functional requirements were converted into well defined test scenarios, composed of test cases defined in terms of input data, output data and evaluation criteria. The set of test scenarios created a test plan with defined schedule and allocated resources (a testbed and a test team). Similarly, the lack of well defined non-functional requirements resulted in poor quality of non-functional testing.

A unit of acceptance testing was an "application", i.e. a functional module of a system. The scope of the application was defined by a set of functions (system use cases) that was subject to testing. We assessed the test plan of an application and the actual testing process using the metrics similar to the following:

- The coverage of functions by test cases.
- The coverage of functions by the sets of test data.
- The coverage of non-functional requirements by test scenarios.
- Qualitative evaluation of the actual test procedure.

The range of values of the first two metrics that we measured for a set of applications of a system is given in Table 5. The values below 1 were definitely too low and warned about those cases, in which only the main runs of functions were tested. The values above 1 could be confronted with the number of runs of the tested functions.

Planning of acceptance testing was the step of the development process in which the application of use case method appeared particularly beneficial. Use case scenarios could nearly directly be converted into test scenarios, and the coverage of the main and alternative use case scenarios by test scenarios was one of the best understood metrics that could characterize the quality of the testing process.

Table 5. Measured values of test coverage metrics

Metric	Range	Average value
Number of: Test cases / Functions	0.70 ... 7.00	2.34
Number of: Data sets / Functions	0.67 ... 15.00	3.80

The most important shortcomings that we found in test plans of the acceptance testing were the following:

- Low coverage of functional requirements by test scenarios – the set of tests offered by the development company covered sometimes the main scenarios of the use cases only, while neglected at least part of the alternative scenarios and exceptions (Table 5).
- Incomplete definition of the actual system state at the start of the testing process – this included the lack of component version numbers, lack of a specification of the initial contents of the data base and of a specification of deviations from the target hardware architecture.
- Imprecise definition of the expected test results – in some cases the expected results were defined by a statement "*Correct results of the computation.*"

The first of the above listed shortcomings (low coverage of tests) decreased the credibility of the acceptance testing process. The second one affected reproducibility of the test results and made the analysis of the current project status difficult when a total disaster happened. The last drawback disorganized the testing process and provoked discussions about whether or not the results that had been obtained were correct, which was not always obvious.

Despite the shortcomings described above, the evaluation of the quality of the functional part of the test plans was positive. The evaluation of non-functional part looked, however, much worse. The lack or precise definition of performance and other non-functional requirements led to the lack of systematic tests within the test plan. The evaluation criteria offered by the development company could not be traced back to the requirements, but rather reflected the "achievements" of the actual design and implementation. If the data was questioned by the customer, a negotiation process was started at the level of the Steering Committee of the project. In one case the performance tests had to be developed by our team.

The way in which the acceptance tests were executed showed that the customers made a significant effort in order to make this process credible. In all but one case the testing process was planned in the project schedule, the test procedure was defined and the necessary resources were allocated.

The mechanics of testing was not uniform. In one case the testing process was centralized and all tests were executed in sequence on a single workstation with the results being displayed on a screen by a computer projector, and evaluated by the commission. In other cases the tests were performed by testers sitting on a set of individual workstations. All the events that occurred during the testing process were formally recorded in a log and the errors revealed were classified according to their importance, and submitted for fixing according to a predefined procedure.

6. Conclusions

The observations related to software development that are presented in this paper are based on the analysis of project documentation and evaluation of metrics exemplified in the previous sections of the paper. Not all of those metrics are quantitative, i.e.

evaluate to a numerical value [10]. However, many of them are formal, i.e. take the form of a mapping between the sets of artefacts or documents.

The results of our work confirmed the results of the survey [1]. Even though the number of projects in our research was small, we could observe a good correlation between the key success factors identified in [1] and the results of the projects evaluated within the scope of our research.

Another result of our study was an observation of a gap between the scope of university courses in software engineering and the reality of the software industry. The majority of software engineering courses are concentrated on object-oriented methods, while structured methods are usually considered obsolete. The reality is different, and structured methods still occupy at least half of the software development market.

Some numerical data that were collected during our study in order to characterize the current practices in software engineering are shown in Table 6.

Table 6. The observed characteristics of the software projects

Characteristic	**Values**	
Software process	waterfall	– 3
	iterative	– 1
Development methodology	structured	– 2.5[*]
	object-oriented	– 1.5[*]
Use of CASE tools	upper case	– 4
	lower case	– 5
Data base architecture	centralized	– 5
Quality of functional testing	adequate	– 3
	low coverage	– 1
	lack of tests	– 1
Quality of non-functional testing	adequate	– 1
	low coverage	– 3
	lack of tests	– 1

[*] One project started with an object-oriented use case model, but was continued using structured methods.

References

[1] The Chaos Report, Standish Group International, Inc, West Yarmouth, MA (1995, 2001, 2003) www.standishgroup.com.

[2] Hartmann, D.: Interview: Jim Johnson of the Standish Group, http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS

[3] American Programmer, 5 (1996).

[4] CISA Review Manual. Information Systems Audit and Control Association, (2002).

[5] ISO 9001: Quality management systems – Requirements. ISO (2001).

[6] Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering, Wiley-Interscience, New York (1994).

[7] Erikkson, I., McFadden, F.: Quality Function Deployment: A Tool to Improve Software Quality. In: Information & Software Technology, 9 (1993), 491-498.

[8] Haag, S., Raja, M.K., Schkade, L.L.: Quality Function Deployment Usage in Software Development. In: Communications of the ACM, 1 (1996), 41-49.

[9] Fenton, N: Software Metrics: A Rigorous Approach, Chapman and Hall (1993)

[10] Lethbridge, T.C., Sim, S.E., Singer, J.: Studying Software Engineers: Data Collection Techniques for Software Field Studies, Empirical Software Engineering, 10 (2005), 311-341.

[11] Sacha, K.: Evaluation of Software Quality. In: K. Zielinski, T. Szmuc (eds.) Software Engineering: Evolution and Emerging Technologies, IOS Press, Amsterdam (2005), 381-388.

[12] Sacha, K., Evaluation of Expected Software Quality: A Customer's Viewpoint, in. L. Baresi, R. Heckel (eds) Fundamental Approaches to Software Engineering, LNCS 3922, Springer-Verlag, Berlin Heidelberg (2006), 170-183.

[13] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, Addison-Wesley (2000).

[14] Rodgers, U.: Oracle: A Database Developer's Guide, Prentice-Hall (1998).

[15] Kruchten, P.: Rational Unified Process: An Introduction, Addison-Wesley Longman (2003).

[16] Gamma, E., Helm, R., Johnson, R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995).

[17] ISO/IEC TR 9126-2: Software engineering – Product quality – Part 2: External metrics. ISO/IEC (2001).

[18] Fowler, M., Scott, K.: UML Distilled, Addison-Wesley Professional (1997).

[19] McCabe, T.J., A Complexity Measure, IEEE Trans. on Software Engineering, SE-2, 4 (1976), 308-320.