e

METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1023-4918**

Summer 2001 (Volume 9 - number 2)

Bring Back the User-Friendly Paradigm!

In the 1990, the diffusion of the GUI-based operating systems (MacOS and Windows) has put a new perspective on the user interface aspect of software development. The buzzwords were that interfaces have to be "user-friendly" and "intuitive". These notions are sometimes vague and have a marketing content. As a long term Windows user, I am still not at ease with the Mac interface, even if this product is regarded as being more "intuitive". Nonetheless, the concepts developed at Xerox-PARC are now widely accepted and shared. The GUI vocabulary (window, push button, drop down list, etc.) is used in all software development.

One of the benefits from this era was that, for a moment, GUI-based applications developed by external vendors and internal teams could have a common "look and feel", if the developers were respecting the visual standard of the operating system. The domination period of the client/server paradigm was however a relatively short one. The ascent of the Internet and the diffusion of Web-based applications completely changed the situation. Running in a browser, the application interface is independent of the client operating system. Based on HTML and available extensions (JavaScript, Flash, etc.), each application is able to regain the "freedom" to have its own graphical standard. The different - and sometimes delirious - looks that are visible on Internet symbolise how the "look" aspect overtook the user interface vision. However, is it in the interest of the user (often a customer) to suffer two minutes Flash introductions or to download heavy graphical files over a low-bandwidth modem connection? Is this the best way to use the server and network resources? The situation reminds me the first days of word processing software. Freed from the typewriters, people produced documents with ten different fonts in four sizes, just because it was easy...

Let us remember that the first objective of the interface is to ease the usage of the application. Just look at heavy traffic sites like Yahoo for an example on how simplicity meets usefulness!

[illegible]

Inside

Automation Strategy Out of the Garage

Mark Rossmiller, SWQA Software Design Engineer,
Vancouver Printer Division, Hewlett Packard

Abstract

As a software engineer, I have been involved in software quality and testing of various forms of commercially developed software for over seven years. Many of these software applications ranging from database tier servers, LAN|Web communication, HR applications, installers, and printer device drivers. In all of these software examples, I developed various forms and levels of automation to increase the efficiency of the testing effort with an equal variety of success. It has been my experience that most of the success implementing automated test tools has primarily not resided in the automation tools employed. Rather, it has existed more in the vision of the leadership to make a paradigm shift from old methods of testing software in order to actualize the value of automated test tools. Without an environment designed, accepted, and developed in all stages of the test planning process the success of automation can be plagued with delays, feature changes, and resistance to change. The following testing strategy was developed to assist my HP colleagues and partners to recognize the necessity to reduce the time and cost of testing methods as it currently exists. I have borrowed from the HP Rules of the Garage as a template in defining the paradigm shift needed to facilitating better, faster, and smarter use of software testing resources.

Believe you can change the world

Past success can be an invigorating reminder of how good a job we have done; that there is no need to alter the course of how we test our products, or whether we need to investigate alternatives to the way we have performed testing on past products. In this view, past success can be a detriment to being able to change testing habits. To believe that one can change the world requires that we are willing to make change ourselves. I think Carly Fiorino stated this best, "*We must be willing to eat our own dog food…*". This means you cannot simply advocate the benefits of automation. One must actually implement and use the tools within a process improvement cycle – **plan|do|check|act** (the Shewhart cycle) [Grady]. Automation development and testing must be understood as having no difference in application from that of the development practices involved in the software being tested.

Work quickly, keep the tools unlocked, and work whenever

Although software testing is often constrained to follow the product development release schedule, it nevertheless should make every effort to optimize planning and arrange resources on its own well defined set of checkpoints. Setup time must be reduced to a minimum or completed prior to drop of the next code release. It is essential to develop processes that enable testing as soon as release drops occur. This requires significant reduction in setup time associated with testing of the next software release cycle candidate. In most cases, there needs to be a consistent test environment established having static test machines in place at all times. In addition, a table of objectives and backup procedures should be prepared in advance to accommodate dynamic testing where a variety of SW|HW|OS configurations are necessary to complete the scope of planned testing. We cannot move quickly enough if we choose to wait until the next drop occurs.

Testing tools are only valuable if they get used. Particularly with complex automation, it is necessary for the tool users to become familiar with its use, provide for the expanded use of the tool, and most importantly, to provide usage feedback that further enhances the value of the tool in testing. It can then be determined if the support of the tool for required testing has merit or if a suitable alternative is available. Some of the risks in practice that can inherently devalue tool and automation investment are:

1. Lack of a sponsorship authority that champions and continuously promotes use of the tool(s).
2. Tools are implemented for testing that is ineffective, or not as originally intended.
3. Support and involvement of partners are not incorporated in the tool design.
4. Significant and sometimes un-communicated feature changes to the product being tested occur.
5. Unclear ownership, expected outcome, and training in the use of the tool.
6. Lack of motivation to use the tools; usage improvements thus cannot be determined rendering the tool obsolete or “locked”.

When testing resources become scarce, further improvements in efficiency can still be realized through automation that can be run unattended or after-hours. “Lights out” testing has been developed and available for several years at HP. VCD has several automation tools built with Visual Test ® on site and leveraged from other sites. These tools have great value, have a history of proven use, and should continue to evolve to meet changing product testing demand. *(Please find tool detail in Resources section of this document).*

Know when to work alone and when to work together.

“Lasting productivity improvement must come from within and cannot be imposed from without.” [Bumbarger].
If you want to work alone in an organization, propose radical change. Obviously, this is not the time to work alone since progressive change requires a receptive and collaborative effort. But the act of proposing change presents many perceptual, habitual, cultural, emotional, and political obstacles from those who would be expected participants and supporters. That is why these environmental factors need to be addressed or it is futile to introduce the tools of change. Real, lasting productivity improvement requires change. And change requires creativity and innovation. The irony in productivity improvement is that success is most unlikely to succeed when imposed from without – but often is forced to change when ineffective from within.

Share tools, ideas. Trust your colleagues.

Results of using automation in software driver testing showed a reduction in testing time by half. There is also the inherent benefit of consistency of execution – repeatability. Thousands of keystrokes or settings can be repeated exactly without the variation when done by hand. Automation reduces thrashing through defects that are not properly characterized and/or hard to duplicate. This results in thousands of reported defects over time remaining open and without a clear resolution. Automation further benefits the tester by providing time savings when multiple SW|FW|HW|OS configurations are required. A multiple of tests can therefore be run concurrently and multiply the effectiveness of the testers time.

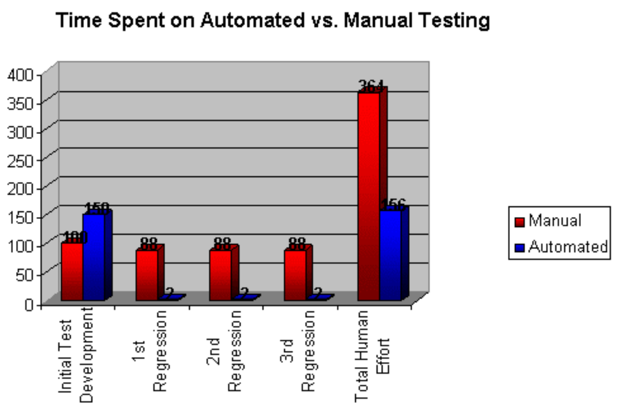


Figure 1. Illustration of statistical average.

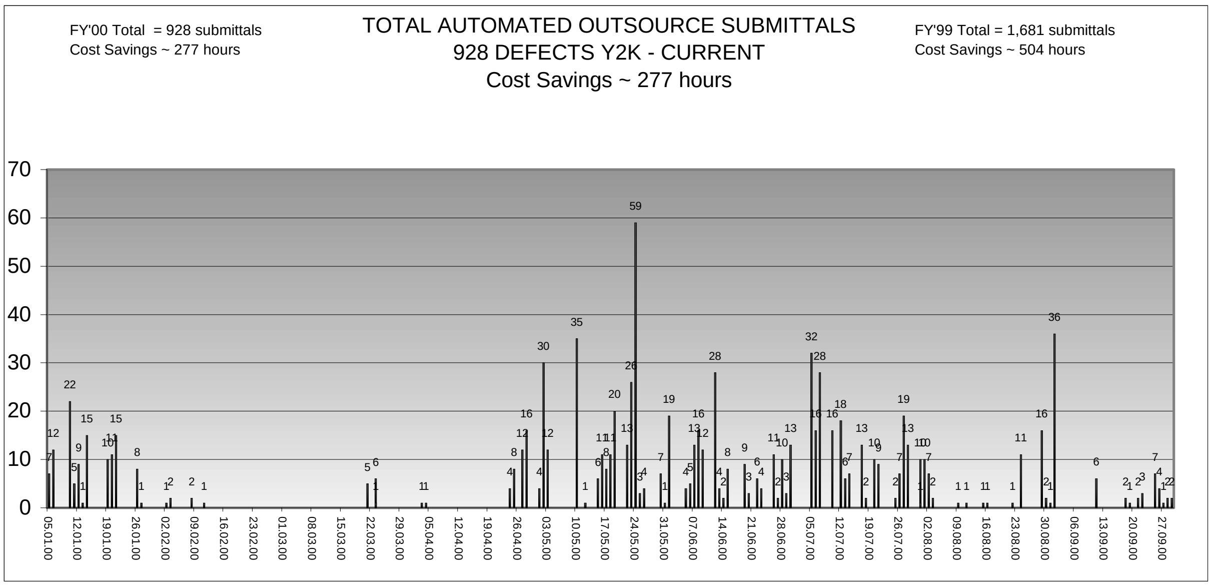


Figure 2. Efficiency improvements

No politics, no bureaucracy

Organizations by the very nature of their existence are political and as they grow - become bureaucratic. Our organization is no exception. But this does not mean that visionary thinking cannot be used to improve or eliminate as many obstacles within the organization to effect positive change.

The customer defines a job well done

From the standpoint of software quality, the customer as well as the next customer represents Argus project data, R&D partners, beta testing, outsource testing, and final media to name a few. Automation has indirectly provided improvements in servicing the next customer via automated defect handling tools to report defects more efficiently and earlier in the development cycle.

Automated defect submittals have, for the past 2 years, contributed the largest share of reported defects at a saving of more than 771 hours in submittal costs. We should continue to support these systems and provide the technical resources and tools to maintain the reduced cost of defect handling.

Radical ideas are not bad ideas.

I found the following “commandments” by Robert Schuller highly insightful in determining our organizational behavior tendency to resist radical ideas. If one can relate to experiencing one of these commandments then we have taken the first step in recognizing where to start improving how we approach testing:

1. Never reject a possibility because you see something wrong with it.
2. Never reject a possibility because you will not get the credit.
3. Never reject an idea because it is impossible.
4. Never reject a possibility because your mind is made up.
5. Never reject an idea because it is illegal.
6. Never reject an idea because you do not have the money, manpower, muscle, or months to achieve it.
7. Never reject an idea because it will create conflict.
8. Never reject an idea because it is not your way of doing things.
9. Never reject an idea because it might fail.

10. Never reject an idea because it is sure to succeed.

Demand change

Once again, the approach taken to automating software testing is greatly affected by the environment in which it is expected to perform. Currently, the software quality organization is entrenched in a technician rich paradigm where testing methods are geared toward manual, ad hoc validation of the software driver and associated printer products. This method is often far down-stream from where the eventually discovered defect – actually was introduced in the development design.

(1) The point is that optimization (automation) should be introduced at the most optimal time in the development cycle to prevent defects from being introduced.

(2) To put automation and tools where they are most effective requires the software quality organization to reclassify the roles and responsibilities of its quality engineering staff. Tools engineering should operate closest to our R&D partners in order to provide automation at the most effective time when design defects can be filtered out. This is before driver release candidates arrive in the labs for validation; rather than down stream where canned testing produces limited results.

(3) Software quality must aggressively evaluate the core competencies of its staff and actively anticipate the knowledge base necessary to meet current and future testing requirements for new and emerging technology. The organizations inability to fill requisitions for needed technical staff and FW engineers, for example, is an indicator that internally we have not adequately anticipated demand external to HP; or we have not grown our own competencies internally to meet future need.

(4) Software quality does not employ consistent testing methods that help facilitate automation, practical tools, or overall efficiency improvement. Other side effects to this inconsistency make valuation of cost or time associated with testing nearly impossible. Features may be added, changed, or even removed in development, which tends to ripple through the test plans and relevant test cases. The way in which we characterize and manage our defects is not consistent creating non-value added work activity increasing the cost of testing.

Invent different ways of working

Here is an analogy to explain the situation in a software quality organization. There are a number of people feverishly bailing water from a leaky boat. The captain brings on more people to help bail but none of them knows how to fix the leak in the boat. The weight of the additional people only makes the water rush into the boat faster and now everyone is feverishly bailing out the boat. Meanwhile, the boat has no direction and is going nowhere. The best solution to the captain's problem might be to bring people on board who can fix the leaks in the boat rather than bail water. Now the captain can stop the leaking, find a direction, and get underway making progress toward a goal.

Our organization must develop a new way of working, by not only investigating but also implementing new ways of testing our products. Automated testing methods are one of many ways to do that if properly sponsored and approached. Both management and testing personnel have become accustomed to doing things the same way, with relative success, and continually fall back on inefficient testing methods we are familiar with but cannot make real progress in improving. We have become complacent and indifferent to changing the methods of testing with which we are most familiar and comfortable. Therefore, when the work starts rushing in and people become stressed, we all start bailing faster and getting nowhere for the future.

Some common risks to the test automation effort include management and team member's support fading after not seeing immediate results, especially when resources are needed to test the current release. Demanding schedules will put pressure on the test team, project management and funding management to do what it takes to get the latest release out. The reality is that the next release usually has the same constraints and you will wish you had the automated testing in place.

Make a contribution every day

I recommend our organization take a long look at the core competencies of our staff and aggressively promote the learning and growth perspective of current software testing personnel. Development plans should have well defined learning expectations that both managers and personnel recognize as a necessary objective for the success of the individual in the software quality organization. These learning expectations should carry weight in deciding opportunities for advancement, ranking, compensation, and overall employee job satisfaction. We also must maintain a standard of expectations on new personnel that they have the skills to improve our organization now and in providing new ideas to increase testing efficiency for future technology of our products. Any concessions made toward qualifications must be made up through continuing education opportunities. Otherwise, it is easy to see why people become resist to using sophisticated tools and what seem to be complex processes they have done without just fine.

If it does not contribute, it doesn't leave the garage

Automating or simplifying unnecessary work only makes it more efficient--it does not make it necessary. Of one of our most recent printer driver products, only 23% of the defects reported resulted in development code changes. Of that 23% resulting in code changes, 21% of those were found by the documented testing cases or "canned tests" that are currently used to develop automated test scripts. Therefore, only 5% of documented or canned test cases result in code changes or product improvements.

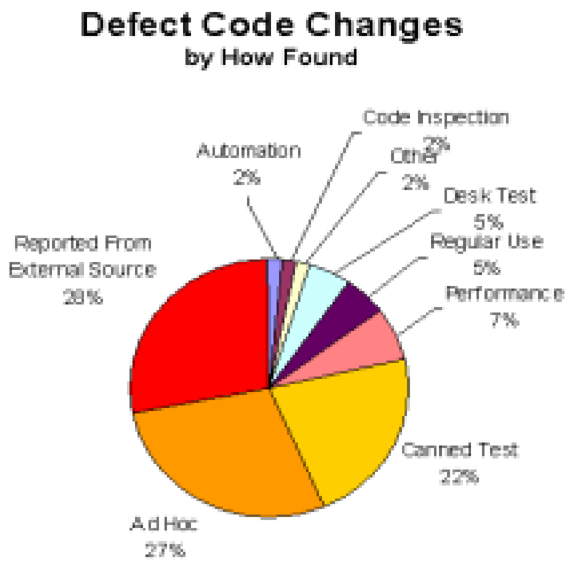


Figure 3. Defects resulting in code changes

Tools and automation should be deployed where testing methods have the highest repetition and/or result in the most return on investment – in the form of bona-fide defect resolution that result in product improvements. Moreover, areas of defect prevention in the design phase, code inspection, and validation of feature changes are the best area candidates for organizational and product improvements in software testing.

Believe that together we can do anything

I believe we can accomplish a great deal more than we currently do. How do we get participation? Here are some suggestions:

- Get management commitment to the automation and tooling process
- Ensure management availability throughout the process not just the beginning and end
- Nurture proper communication channels: be direct, open, and honest about progress
- Let people be involved in decisions that affect them
- Demonstrate mutual respect by listening to each other; listening includes empathizing

- Get those involved in the process to talk to each other
- Invest in training; we need knowledge workers
- Eliminate class distinctions and adversarial relationships; not SWQA + R&D = R&D

Attention to these requirements will create advantages for the whole organization by improving communication, spreading knowledge, increasing collaborative attitude, and improving motivation and morale.

Invent

You cannot discover new oceans unless you have the courage to lose sight of the shore.

From a financial and internal business perspective, software quality's best contribution to the bottom line is in reducing cost. This means focusing our resources and automation tool effort at the most optimal point in the value chain. Tools that will provide verification between R&D delivery of the software driver and preceding planned verification in software quality. Fortunately, efforts are underway to introduce automation where it is most effective. Baseline acceptance, code inspection, driver file and library verification tools are excellent opportunities to reinvent the way we test and deliver our software products.

The learning and growth perspective will require more knowledge workers. To accomplish this requires a regular assessment to increase our internal core competencies and to bring the technical expertise on board that offers efficiency opportunities to the organization.

References & Resources

1. Grady, Robert B., Successful Software Process Improvement. Hewlett Packard Company. Prentice Hall, New Jersey, 1997.

2. Fiorino, Carly. The Journey: Rules of the Garage. Hewlett Packard Company. Palo Alto, CA. 1999.

3. Degrace, Peter and Leslie Hulet Stahl. Wicked Problems, Righteous Solutions : A Catalogue of Modern Software Engineering Paradigms. Yourdon Press Computing Series, 1999.

4. Visual Test6 Bible by Thomas R. Arnold Ii, Thomas R., II Arnold.

5. Bumbarger, Wm. Bruce. OPERATION FUNCTION ANALYSIS. Presentation, WSU Engineering Management, 1999.

6. Test Automation© automated application compatibility, HP Deskjet feature testing; authors: Thong Nguyen, Mark Rossmiller, VCD, Hewlett Packard Company http://vcsweb2/ca/swqa/toolproc/testauto/testauto/ppframe.htm.

7. Host-Client Virtual Lab© automated networked resource and remote allocated testing; authors: Norm Peterson, David Wagner, AIO/HID/SWTC, Hewlett Packard Company http://vcsweb2/ca/swqa/toolproc/host-client/hostclient/ppframe.htm.

8. SWATH© automated application, network compatibility testing, HP Laserjet feature testing; author: Beverly Takeuchi, PLD, Hewlett Packard Company http://laserjet.boi.hp.com/2PDF/OneClickProjectPlan.doc.

9. VT in Depth. In Depth Productions, Inc. http://www.vtindepth.com.

10. NetHost© Network Distribution Test Script for host client test execution: author: Richard J. Wartner http://members.aol.com/rwvtest/index.htm.

HOORA, The Method for Requirements Analysis

Johan Galle, Johan.Galle@e2s.be, http://www.hoora.org
E2S, Technologiepark 5, B-9052 Zwijnaarde, Belgium

The HOORA (Hierarchical Object-Oriented Analysis) method is a pragmatic approach to modeling your systems. One of the unique features of HOORA is that it not only explains the UML (Unified Modeling Language) notation. It defines a sequence of steps (a process) that should be followed while modeling. It adds process knowledge and guidelines to the UML notation.

One of the innovative features introduced by HOORA, are so-called automatic diagrams. A static model diagram shows one particular class, and all the other classes that are statically related with it. A dynamic model diagram shows one class and all the other classes that dynamically interact with it. A package summary diagram shows one package and all the other packages that it depends on, statically and dynamically.

Such diagrams are recommended for complexity management and project follow-up. They are generated from the model information. These diagrams allow to combine the advantages of providing freedom, with the rigorousness required to tackle today’s applications.

A major add-on to the HOORA method is a tool going beyond UML. This tool is called HAT, HOORA Analysis Tool. HAT shows the feasibility and the need for a tool supporting the process, the modeling activity, and not just the (UML) notation.

In this paper, we explain the HOORA process, and these automatic diagrams. HAT screenshots are used.

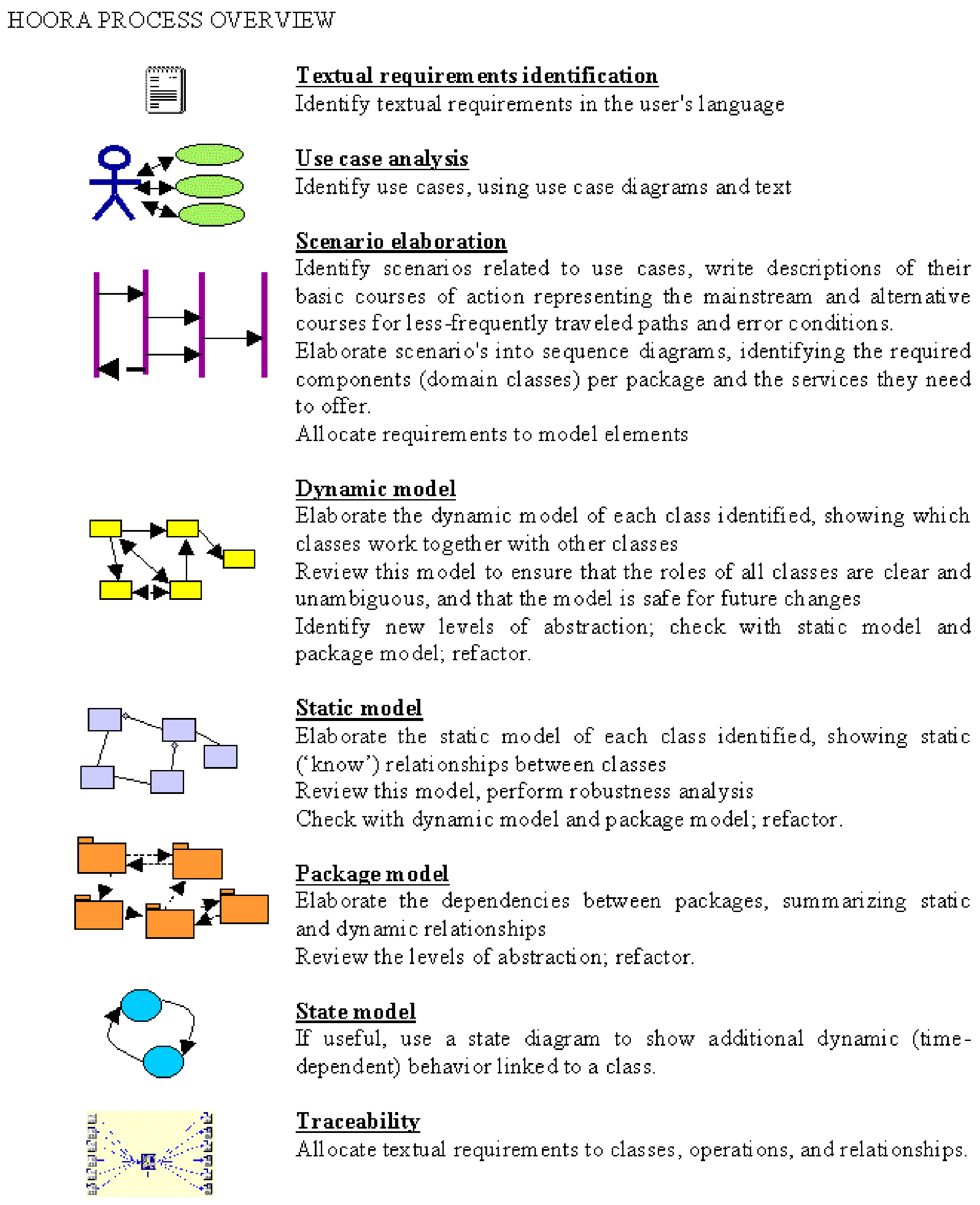
1. HOORA process overview

A complete overview of the HOORA process can be found in the HOORA Process Manual [1] on http://www.hoora.org. This paper only highlights a few issues.

The goal of the HOORA model, at each level of abstraction, is to derive a suitable collection of 'things', so that, when they collaborate (and when they know each other), they solve the set of problems applicable at the particular level of abstraction. Because of that, HOORA stresses the dynamics of the system. We do not just want to identify ‘candidate’ objects. We want to identify these objects that are well suited to structure the problem. And the main criterion we use for that, is to consider the dynamics.

How do we achieve this? We start with use cases to identify fairly big functional blocks. Then, we try to find the set of classes that we need to support every scenario identified within each use case. As a result, we obtain dynamic models showing which classes work together with other classes. The dynamic model shows the classes collaborating with each other (*'who do you talk with?'*). The static model is its counterpart, showing the static relationships (*‘who do you know?'*). As in the real world, object A can only talk with object B provided object A ‘knows’ object B, or object A ‘knows’ one or more intermediaries ‘knowing’ object B in their turn. So, static and dynamic relationships are interlinked.

Considering the dynamics is done by investigating scenarios and identifying classes that are suitable to capture that particular behavior. Because scenarios are examples, it is important to have a mechanism to summarize the decisions taken during scenario modeling. This is where the first automatic diagram type pops up, the dynamic model.



HOORA PROCESS OVERVIEW

Textual requirements identification
Identify textual requirements in the user's language

Use case analysis
Identify use cases, using use case diagrams and text

Scenario elaboration
Identify scenarios related to use cases, write descriptions of their basic courses of action representing the mainstream and alternative courses for less-frequently traveled paths and error conditions.
Elaborate scenario's into sequence diagrams, identifying the required components (domain classes) per package and the services they need to offer.
Allocate requirements to model elements

Dynamic model
Elaborate the dynamic model of each class identified, showing which classes work together with other classes
Review this model to ensure that the roles of all classes are clear and unambiguous, and that the model is safe for future changes
Identify new levels of abstraction; check with static model and package model; refactor.

Static model
Elaborate the static model of each class identified, showing static ('know') relationships between classes
Review this model, perform robustness analysis
Check with dynamic model and package model; refactor.

Package model
Elaborate the dependencies between packages, summarizing static and dynamic relationships
Review the levels of abstraction; refactor.

State model
If useful, use a state diagram to show additional dynamic (time-dependent) behavior linked to a class.

Traceability
Allocate textual requirements to classes, operations, and relationships.

Hoora process overview

2. Dynamic Model

2.1 HOORA responsibility-driven nature

There are primarily two schools of thought as to the best way to identify appropriate objects, data-driven versus responsibility-driven. We take the position that a responsibility-driven approach facilitates a more logical, more seamless, more object-oriented process [2].

While identifying classes, we emphasize dynamics. We only introduce classes because we need them at a given level of abstraction. At that level, we identify

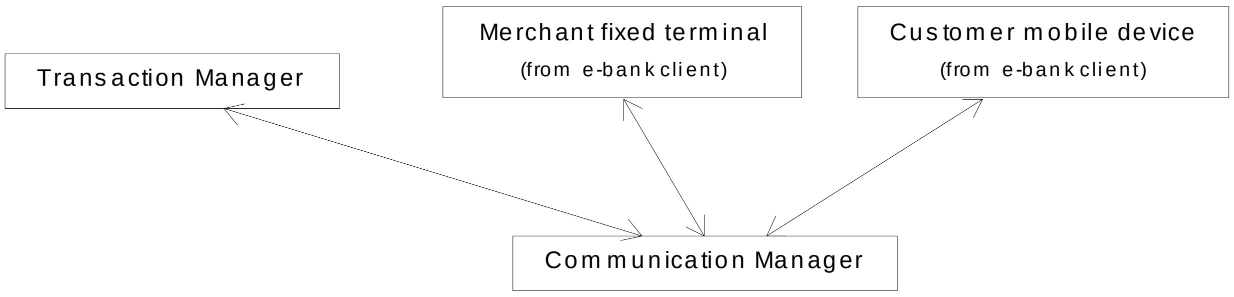


Figure 1. Dynamic Model Diagram

responsibilities and we allocate them to the newly introduced classes. At that stage, we may also identify operations for the classes. These operations implement (part of) the responsibilities [3].

The major reason for HOORA to be responsibility-driven, is that we feel that the core characteristic of object-orientation is NOT inheritance, relationships or real-world equivalence (although these characteristics are important). We feel that the core characteristic is that a model allows describing our system, and this model consists of the collaboration of a minimal set of objects. This defines ‘object’. An object is whatever part helps us describing the system as a collaboration of parts.

Among other reasons, this emphasis is due to the need to consider real-time requirements early in the project phase. So, we need a way of working which is very suitable for describing (and annotating) behavior.

HOORA uses scenarios as a key driver to the modeling process. The scenarios will ensure that we identify exactly this set of classes that we need in order to capture the system's behavior. Scenarios are there to ensure that we identify the objects that are required at the particular level of abstraction that we are working on. Other guidelines (e.g. 'underline the nouns') do not provide such a guarantee.

Of course, HOORA does not reject data-driven approaches. In practice, both approaches are used in parallel, and data objects are still important candidate objects. But they are not the only source. The problem with data-driven approaches, is that you need additional guidelines (i.e. consider the dynamics) to determine which classes / associations NOT to consider at a given level of abstraction.

2.2 Dynamic model diagram

The dynamic model diagram is centered around a particular class. For this class, we consider all sequence diagrams (representing scenario’s) where this class occurs. We then generate arrows capturing the dynamic control flow to and from this class.

Figure 1 shows all the classes that Communication Manager interacts (‘talks’) with. It shows that Communication Manager talks with Transaction Manager, Merchant Fixed Terminal, Customer Mobile Device and vice versa. This diagram is generated based on a set of sequence diagrams. Classes have been identified based on responsibilities. So, this diagram must be understandable from the point of view of these responsibilities. Because the responsibilities have been chosen in a particular way, some classes do not appear in the Communication manager’s dynamic model.

Solely by reasoning about these responsibilities, we may identify errors or incompleteness in this diagram. Such errors need to be corrected on the corresponding sequence diagrams. This is part of the

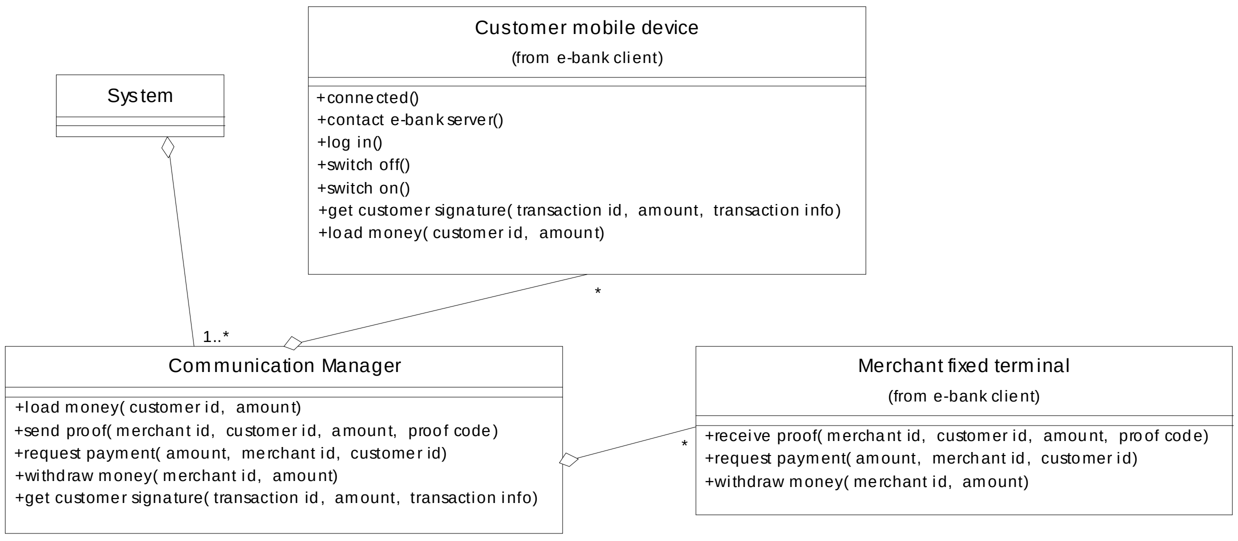

Customer mobile device
(from e-bank client)
+connected()
+contact e-bank server()
+log in()
+switch off()
+switch on()
+get customer signature(transaction id, amount, transaction info)
+load money(customer id, amount)
System
1..*
*
Communication Manager
+load money(customer id, amount)
+send proof(merchant id, customer id, amount, proof code)
+request payment(amount, merchant id, customer id)
+withdraw money(merchant id, amount)
+get customer signature(transaction id, amount, transaction info)
Merchant fixed terminal
(from e-bank client)
+receive proof(merchant id, customer id, amount, proof code)
+request payment(amount, merchant id, customer id)
+withdraw money(merchant id, amount)
*

Figure 2. Static Model Diagram

HOORA support for complexity management.

3. Static Model

3.1 Static model diagram

The static model receives less emphasis in the HOORA process. Still, the classes as identified (mainly by investigating the dynamics), have static relationships among each other. These relationships are represented on class diagrams. In the same way as the dynamic model, we generate a static model diagram, centered around one particular class, based on all class diagrams where this class occurs.

Figure 2 shows all the classes that Communication Manager is statically related with (‘knows’). Static relationships also contain multiplicity information (1..n, n..n relationships).

3.2 Comparing dynamic and static models

There is a validation check to be done when comparing the static and dynamic model.

We are now at a stage where we believe that, at some level of detail, we have entered all the model information required. The goal of this step is that we investigate what the links are between one class A and other classes B, both statically (A knows other classes B) and dynamically (A talks with other classes B). We will check whether these dynamic and static links are consistent with each other (it should be possible for A to select the appropriate B’s it wants to talk with, based on the B’s that it knows).

Consider the static and dynamic model of Communication Manager. We need to check that not only there are one direct or a combination of indirect static links for every dynamic link. We also need to check whether the Communication manager is able to select the particular instance it is said to talk with according to the underlying sequence diagrams.

The goal of this activity is to identify areas where the model is still incomplete. Static and dynamic model information is then added, until both diagrams are consistent with each other. It is a common practice to ignore some aspects during a particular phase in analysis. Still, it is important to have a model that is at least consistent in this respect. We are not allowed to ignore a particular aspect in one part of the model, but still to use it in another part.

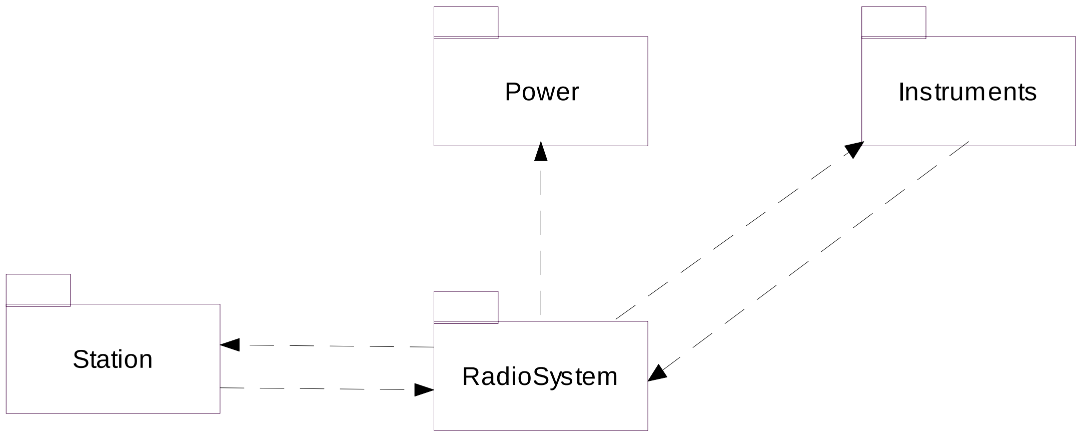


Figure 3. Package Summary Diagram

4. Package Model

Levels of abstraction are a major means to deal with complexity. In HOORA, we use packages to represent them. A package consists of model elements. A package can be decomposed further into packages. The static and dynamic models show classes being associated with each other. These classes can belong to different packages. The package summary diagram allows viewing the dependencies between packages (summarizing the static and dynamic relationships).

The goal of such a diagram (Figure 3) is to check whether the dependencies between levels of abstraction make sense (taking into account the intended purpose of each package), and to suggest changes to it. Classes can be moved from one package to another, packages can be grouped or split, packages can be decomposed, … We aim to achieve minimal dependencies between packages. Cyclic dependencies imply the impossibility to separate packages.

The hierarchy rules in HOORA are stricter than in UML. HOORA defines two factors that determine the distance between two elements. We calculate the number of levels you need to go up the hierarchy and the number of levels you need to go down the hierarchy. The modeler, by investigating these numbers, has quantitative data showing the need to refactor.

5. Conclusion

This paper shows one way of using UML and HOORA. HOORA allows some variations, e.g. state modeling in early phase of use case modeling is possible as well, the moment when requirements traceability relations are decided upon,etc. Still, HOORA is a lot stricter than UML. A major advantage of the HOORA method is that we do not have a ‘blind’ phase of class identification. We only introduce the classes that we need, according to the collaborations required in the scenarios. This method is driven by the dynamics. The dynamic model dictates the static model. The motivation for introducing automatically generated diagrams showing the static model, dynamic model and package summary, is to contribute to complexity management (and thus the need to guarantee that all the links between model elements are shown).

6. References

1. HOORA Process Manual, http://www.hoora.org.
2. D. Demarco, “Analysis and design methodology”, Thoughtworks Inc., Summer 95.
3. R. Wirfs-Brock, “Adding to Your Conceptual Toolkit: What's Important About Responsibility-Driven Design?”, ROAD, July-August, 1994.

The Work Situation in Software Development

Michael Frese, University of Giessen, Otto Behagel Str. 10F, 6300 Giessen, Germany
Wolfgang Hesse, University of Marburg, Hans Meerwein Str., 3550 Marburg, Germany

If one wants to improve software quality, one ought to analyse not only the quality of the software products but also the production process itself. The work situation of software developers has been analysed in an empirical and interdisciplinary study; its major results are reported here.

Some of the important results are that communication is required to a high extent, that there are negative stress effects in terms of burnout and that user participation has negative effects on project and process quality. Moreover, software developers are not as interested in the functionality of new tools as their usability; new methods like object oriented design may lead to increased demands to communicate with each other.

The work situation of software developers as a factor of software quality

If one wants to improve software quality, one should not only look at the result of the production process but also at the production process itself. Traditionally the quality of the product has been the focus of attention. Recently, this view has been complemented by taking into consideration the process quality as well. Moreover, the production process of software is not only looked at as an aspect of total quality management and as a factor of the product quality but also as an issue in its own right.

This viewpoint implies that the participants of the process become the targets of research in the area of software engineering - the software developers, the project leaders, and partly those users who participate in the production process. The work situation during the production process is of major importance for process and product quality.

This was the starting point of the IPAS-project (IPAS is a German acronym and stands for "interdisciplinary project on the work situation software development). In all 29 software development projects from 19 German and Swiss firms were studied. The means size of the projects was 10 members; the total sample comprised 200 software developers. All of them filled out a questionnaire and were interviewed for several hours.

The software produced by these projects covered a broad application domain, for example administration, banks, insurances. Military or academic projects were not included because they do not produce for a free market

Our framework for research

The basic framework for our interdisciplinary project is presented in Table 1. The idea was to cross the different perspectives of the disciplines working together in this project (computer science, work psychology and industrial sociology). Our original framework was modified for this presentation because we excluded the sociological part from this article - a column on project management.

The rows of this framework consist of categories from work psychology that have been shown to be important to describe work situations, the columns are related to software-technical criteria that are important in software engineering. These two dimensions lead to 15 problem areas that will be used to describe the results of our study. This article presents an overview of the results.

Software technical categories / Psychological categories	Relationship with user	Development	Methods and tools
Coping with complexity	Familiarising oneself with user applications areas	Appraisal of one's job complexity	Use of tools and developers' requests
Cooperation and communication	Difficulties with user participation	Communication as major part of development	Influence of new methods on communication
Job discretion	Factors influencing job discretion: standards, guidelines, change requests, flexibility	Correlation between job discretion and burnout	Influence of job discretion on acceptance of tools
Qualification	Qualification of software developers in the application area	What characterise excellent software developers	Learning to use methods and tools
Stressors	Developer's stress through user participation	Stress and burnout: software development as challenge or stress factor	Effects of changing methods and tools

Framework for research

Results and discussion

1. Coping with complexity

The necessity to cope with the complexity of the software development process is an important problem for software developers and their managers. Most likely, complexity of this process will increase in the future. While new programs and tools render comfortable metaphors or graphical user interfaces, the user of such programs and tools for the design process is confronted with a rapidly growing number of products, difficult decisions, incompatible interfaces, etc.

There are ever shorter innovation and production cycles. Once one has to know one tool, there is already a new version or a new product on the market. This leads to new (learning) problems if these new tools are used. Thus, one result of our research is that usability and easy learnability are more important criteria for software designers than better functionality (cf. 1.3 below).

There are two strategies to cope with complexity: one can either reduce complexity or learn to live with it. In the software projects studied, we found both strategies. On one hand, there was problem oriented modelling, a coherent procedure with one method (e.g., with an object oriented approach) or the use of CASE tools - all of these are attempts to reduce complexity. On the other hand, psychological research showed that complexity reduced stress and increased motivation in the software developers (cf. l.2)

1.1 Qualification of software developers in the application area

One important aspect of complexity in the work of software developers is the necessity to get to know new application areas and to find solutions that are adapted to the specific user needs.

Current tendencies aim at a more highly integrated development process ranging from functional design to technical design and implementation. The pressure to produce software more efficiently and with fewer costs will most probably increase in the future. This implies that reusability will be a highly important topic. Reusability does not only refer to software modules but also to function design, user models, and even to analysis documents. This leads to new demands on software developers to deal with complexities not only in their own technical area but also in the areas of functional design, user modelling and analysis of particular application areas

1.2 Appraisal of job complexity

Since software development is a highly complex undertaking, one often draws the conclusion that its complexity should be reduced. This is probably true in certain cases. However, our results suggest that this should not be a general strategy. Work complexity is often something very positive for the software developers. For example, software designers with highly complex jobs show less burnout. They are also prouder of their work and their team is prouder than software designers with jobs of little complexity. Thus, a reduction of complexity may have counterproductive effects.

It is useful to differentiate between complexity of tools on one hand and complexity of tasks and activities on the other hand. Tool complexity is very often seen as something negative, task and job complexity are positive aspects of the work place. Tools should, therefore, not reduce the overall complexity of the job; they should only contribute to a reduction of routine activities or lead to a reduction of "detours". Examples for detours are activities that are not directly linked to the work tasks, e.g. mounting a new tool. This allows the software developers to concentrate the intellectual challenges in work.

1.3 Use of tools and developers' requests

Various parts of our research were concerned with tool use: which tools were used for doing which tasks, what did the software developers want from tools, effects of tools, etc.. The results show that there is a large discrepancy between theory and practice. While the CASE market offers tools with sophisticated graphical user interfaces, many software developers still have to deal with VSAM, CICS or line editors.

A comparison between what software developers want and what they have shows that typical tasks like coding, editing, and debugging are adequately supported by tools. On the other hand, the software developers would like to have better tools in the areas of documentation, graphical support (especially analysis and specification tools), test support, generators and project management.

In general, the developers report to be well supported by the tools available. However, the importance of usability was higher than the functionality of the tools. Forty per cent saw "easy to learn" and "easy to use" as the most important characteristics of a tool. Functional aspects like "without errors, "well adjusted to task", "easy to combine with other tools" or "performance" were only mentioned by 5% to 8% as important. The wants and requests of software developers are relatively modest. Only those designers who had got to know other tools in other projects or in their studies presented suggestions for improvement. The more they have used tools the more they thought that their tools are flexible enough to change them.

Tools may be important but they are clearly not more important than motivational factors and social support by co-workers and supervisors.

2. Communication and cooperation

2.1 Difficulties with user participation

Presently, there are many arguments for a higher degree of user participation in the design process, e.g., through prototyping Often, only the advantages of such an approach are presented and potential disadvantages are not discussed. We also expected that user participation would lead to positive consequences. However, the data prove the opposite.

We found that the efficiency of the software process, the changeability of the product, the quality of the product, time efficiency, etc. were all reduced in projects with high user participation. In other words, the users seem to disturb the development process. There are more change requests in the projects with user participation. The innovativeness of these projects is also seen to be lower by the software developers. These results are not the opinion of just a few software developers. They are supported by the majority of the software designers in the projects with high user participation.

Our results do not mean, however, that user participation is harmful per se. But it does show that there may be more dangers and difficulties in user participation than what is known at this point. User participation can lead to frictions, to additional tasks and demands. These problems have to be investigated and adequate strategies have to be developed to deal with them. At the very least, one has to take into account that these problems can appear with user participation and rapid prototyping. In order to cope with these matters a certain amount of time and budget should be allocated when user participation is planned.

In addition, software developers' orientations have to be changed. As long as user participation is not defined to be a central task for software developers, user participation will just lead to disagreeable add-on demands. Only after a new orientation is introduced can software designers look positively at change requests by the users.

2.2 Communication as a major part of software development

It is now generally known that the solitary programmer is a rather exceptional case. Nevertheless, many trainings and university curricula do not sufficiently take into consideration the fact that programmers have to work in teams (at least not in Germany). Interestingly, programming is only a very small part of software development (about 10% of the time) while all the social activities, e.g. exchange of information, consulting, team discussions, etc., make up a much larger percentage of the developers' work day (about 40% of the work tasks is in some way socially oriented).

Can we conclude that software development is a social profession? This is certainly not the case. But the social component is of utmost importance not only for team leaders but also for the team members. Most developers' tasks have a social component, e.g. specification, design, testing, system support, technical support, review sessions, coordination, adjustment of interfaces, etc.

All this requires much social competencies. The developers have to react flexibly when different communication requirements appear. These requirements may mean that one has to moderate a group, to structure ideas and to support other people or to criticise them. Conflicts have to be coped with and different communication strategies have to be followed in discussions, depending upon the other's knowledge and prerequisites. Not all of these competencies are acquired through normal academic training courses.

It is of particular importance not to present a general sort of social competence training but to offer specific courses integrating technical skills in the field of computer science with communication skills.

2.3 The influence of new methods on communication

New methods and tools may considerably affect cooperation and communication in a team. For example, Parnas principle of information hiding or the data abstraction method may lead to a bureaucratisation in software development and a reduction of communication. In contrast, a change to an object oriented method may lead to the opposite result.

The following remarks are not based on statistical analyses because there were oddly a few projects in our sample that used object oriented design methods. However, it was our impression, that there is particularly strong need for communication and cooperation in such projects. This is probably due to different software structures. While the data abstraction method exclusively connects different modules through import/export relationships, this is not the case in object oriented design where inheritance relationships play an equally important role.

The data abstraction method allows a design by developers who work relatively independently from each other. While division of labour is necessary in object oriented design as well, the various developers have to communicate a lot, e.g. on features inherited from each other. Usually, a purely formalised communication, e.g. the exchange of formal specifications, is not sufficient because intensive and frequent talks about the implications of various decisions in the design process are necessary in object oriented design.

3. Job discretion

Job discretion implies that a person has possibilities to change things at a work place. For example, if job procedures are not rigidly prescribed, if software developers have some influence on what kinds of tools are used, if they can decide how they are doing their job and if they can decide at what time they do certain tasks, their job discretion is high.

There are essentially two management approaches to job discretion. One implies that job discretion will only lead to a reduction of productivity and that management should organise work in such a way that there is little job discretion. This is the Tayloristic approach. The other approach is based on modern work psychology and it implies that the people at work will only be fully motivated to work if they can use their abilities and skills in their work. This implies that job discretion should be high.

3.1 Correlates of job discretion

Different software designers have different degrees of job discretion. One issue is the flexibility and the changeability of the programs. If there is a high degree of changeability of the program, one can, e.g., deal more quickly and adequately with change requests.

In this connection, it is of interest to know who is most often responsible for changes - is it the contractor of is it the software developers themselves. If it is the former, there is little scope of action (and hence little job discretion), if it is the software designers, change requests are themselves the result of a high degree of job discretion. Our results were surprising: apparently, a large part of the change requests stems from the software developers. About 40% of the change requests were due to internal sources.

Management or other developers in the team were responsible for 14% of the change requests. However, the most important cause of change requests was the development of new solutions by the individual software developer him- or herself (25%). Job discretion is used here to develop and improve one's work and one's solutions.

No wonder that job discretion is seen as a positive factor in work psychology that contributes to higher product and process quality.

3.2 Correlation between job discretion and burnout

Job discretion is not only an important factor influencing performance, it is also related to burnout. The higher the job discretion is, the lower is the burnout for software developers. If there is a high degree of rigid division of labour, this usually reduces job discretion. Moreover, there is a relationship between job discretion and work complexity. If there is little complexity, there is also little that can be decided in work.

Since burnout probably has an impact on productivity - be it through absenteeism or through little motivation for work - a high degree of job discretion is an important productive factor.

3.3 The influence of job discretion on acceptance of tools

One of the issues in our research project was to study the acceptance of tools by the software developers. In general, the tools were quite well received. The software developers think most highly of simulation tools, low level editors and project management tools. The tools that are perceived to be less good are project libraries. This may be due to the fact that these tools can only be used adequately with a high degree of experience.

There was also an interesting interaction of job discretion and the perceived usefulness of the tool. Perceived usefulness of the tool was a general perception by the software developers of how good the tool was adjusted to the tasks and how usable it was. In general, usefulness of the tool also implied that stress effects were minimised. This fact was mitigated by job discretion, however. Under conditions of high job discretion, usefulness of the tool was related to reduced stress effects. If the job discretion was low, there was little relationship to stress effects.

Thus, the usefulness of the tool seems to have a positive impact only if the general conditions in the job allow the person a lot of freedom. If this freedom is low, the usefulness of the tool does not help the software developers to cope with the stress effects.

4. Qualification

4.1. Qualification of software developers in the application area

Software developers have to learn new things in every project. In most cases, this has to be done without formal training. Of particular importance is the knowledge acquired about the area of application. To gain this knowledge the software developer should have the following competencies:

- to learn a different conceptual system in which he or she has to be the translator,
- to be able to pose clear and sensitive questions,
- to separate plausible from implausible hypotheses and essential from unessential information,
- to discover structures which may be so routinised in the area of application that people will not be able to verbalise them any more,
- to distance oneself from one's own egocentrism and to put oneself into the shoes of somebody else; this implies that one has to be able to distance oneself from one's own discipline and to take the perspective of what the users have been saying,
- to develop criteria to find out whether one has been understood or whether or not one has understood what the users have been saying,
- to make abstract concepts concrete, i.e. to find examples for abstract examples in the area of application.

Moreover, the knowledge in the area of application has to be reconciled with the requirements of the project. This means, for example, that the software developer has to analyse requirements of the application area and whether they can be realised technically.

Thus, software developers need to show considerable qualifications. At the same time, they rarely receive formal training for many of these skills. This implies that they have to learn strategies to learn by themselves. These strategies may be one of the most important factors of software developers' success in their job - again it is something that has received little attention.

4.2 What characterises excellent software developers?

In order to find out which qualifications and competencies software developers need to do their job well, we have to identify excellent software developers in the projects studied.

This was done with a procedure used quite often in work psychology: peer nominations. All people were asked to identify the best software developer in their team. Those people identified by two of their colleagues were taken to be excellent software developers. Moreover, the software developers were asked what characteristics make them excellent. What characterises these excellent in comparison to average software developers?

The three most important characteristics were: technical competence, ability to work in team, work style. Technical competence implies a high degree of professional experiences, an ability to recognise and solve problems quickly and a high degree of knowledge about the particular project (e.g. knowledge to whom one can address which questions, which information can be received where, the structure of the system to be developed, knowledge of previous errors and wrong decisions and why are things done the way they are done).

The ability to work in teams implies that it is fun to work with this particular person, that this person can communicate effectively and that he or she is willing to work together with others. A good work style is characterised by using systematic procedures at work, by being independent in work and by being team oriented.

It is interesting to note that the excellent software developers did not have a higher job tenure but they had more intensive experiences. They had participated in more projects and they knew more programming languages. They are asked to participate in more meetings and they are asked more questions by the other software developers. Thus they have to communicate more (and are apparently better able to communicate) than the average software developers.

Again, we find the importance of communication skills. The excellent software developers are the natural leaders of their teams. Sometimes they are just the informal leaders, but they are also often made to become supervisors in due time.

4.3 Learning to use methods and tools

The qualification of software developers for the use of methods and tools is an important prerequisite to do the job well. Our studies showed that learning while doing the job prevails rather than formal and systematic training. In about 40% of the cases, the use of methods and tools was learned on the job; the software developers received some formal training in only about 16% of the cases.

However, there are clear differences between the academically trained computer scientists and the non-academic software developers.

The computer scientists had some knowledge on methods and tools in 41% of the cases; this was the case for only 26% of the software developers who were not academically trained.

This reinforces the importance of self-discovery learning in software development. Those who have learned how to learn will be at an advantage. To support this "learning to learn", one would have to teach learning techniques, self-organisation, etc..

5. Stressors

5.1 Developers' stress through user participation

There is a clear correlation between user participation in a project and the number and intensity of stressors impinging on the software developers.

In projects with user participation, there are more stress factors than in projects without user participation. At first sight, this result may be surprising. However, it makes sense that software developers, who have to work under high time pressure anyhow, will have more stress if their normal work is interrupted and additional user requirements have to be met on top of their normal work. Again, this shows that user participation is not as unproblematic as it seems.

5.2 Stress and burnout: software development as challenge or stress factor

From the standpoint of work psychology, the work places of software developers are in general quite good. The software developers have challenging tasks, usually of high significance and importance. These tasks are complex enough and it is possible to interact socially in the job. Nevertheless, there are stress factors which may contribute to psychological impairment even in this sector that may seem to be privileged: too much work, frequent interruption, unclear or late information, career pressure and pressure to outperform.

Many software developers and project leaders checked in the questionnaire that they have to deal with an enormous workload and that their work is interrupted again and again. They have to work overtime frequently and cannot compensate this with additional leisure time.

Most overtime has to be done by project leaders and user delegates in the software team. The latter have a particular high workload. At the same time, they have to work both for the project and in their normal duties. An added burden is the pressure from their constituency (the users to be) and the pressure to be part of the software team.

The burnout syndrome is of particular importance for software developers. Burnout implies that a person feels exhausted and is not able to identify with his or her work any more and that he or she has the feeling, that one's ideal, ideas and one's potential cannot be realised.

One has to assume that burnout may contribute to loss of motivation at work, to psychosomatic problems and to a higher degree of absenteeism. Thus, burnout probably does not only reduce motivation but also productivity. A finding supporting this claim is that excellent software developers show less burnout than the average ones.

Burnout has a clear-cut relationship to the stressors of software development: the higher the stress factors at work, the higher the burnout. Moreover, those people with higher job discretion and higher complexity of work also show less burnout. Similar effects appear with group climate: teams that are democratic, more open for critique and less authoritarian also show less burnout among their members.

5.3 Effects of changing methods and tools

The high degree of innovation in the area of methods and tools can be one stress factor as well. While tools are supposed to support the work of software developers, they often produce the opposite effect. This may be the major reason why software developer attaches the most importance to the criterion "learnability" when they judge tools (c.f. 1.3).

6. Conclusion

Our project has dealt with an area that has not been frequently or systematically researched. From a psychological perspective some surprising findings were the high degree of communication requirements in the job of software developers, the importance of burnout as the major effect of stress at work, the cohesive picture of excellent software developers and the negative impact of user participation on product and process quality. Some of these results have been reported by other as well. The most troubling finding is certainly that user participation may have a negative effect in the design process.

From the standpoint of computer science, important findings were that software developers are less interested in the functionality of new tools but rather in how easily they can be learned. Moreover, the well-known gap between software developers and users appeared again in our research.

New methods, e.g. object oriented design, may lead to a higher necessity to communicate well. The question of the complexity of the tools versus the complexity of the job is certainly something to be considered in the application and development of new tools.

Applying new software engineering techniques always implies that the work situation of software developers is changed for the better or for worse. Thus, the change of the jobs of the software developers is an important factor to be considered in the area of software engineering.

This article was originally published in the July 1995 paper edition of Methods & Tools

Classified Advertisement

Add Value by implementing Aardvark Bug Tracking Software
Consolidate all of your project information in one place and allow your development and testing process to thrive! Aardvark bug and defect Tracking software is the leading browser based bug tracking system, find out how you can concentrate on the tasks that add value at www.red-gate.com/bug_tracking.htm or contact info@red-gate.com for more information.

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organising software development related training? This space is waiting for you at the price of US $ 20 each line. Reach more than 15'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting those that download the issue without being registered! To advertise in this section or to place a page or Web site ad simply send an e-mail to franco@martinig.ch

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig; e-mail franco@martinig.ch ISSN 1023-4918
Free subscription: www.martinig.ch/mt/index.html