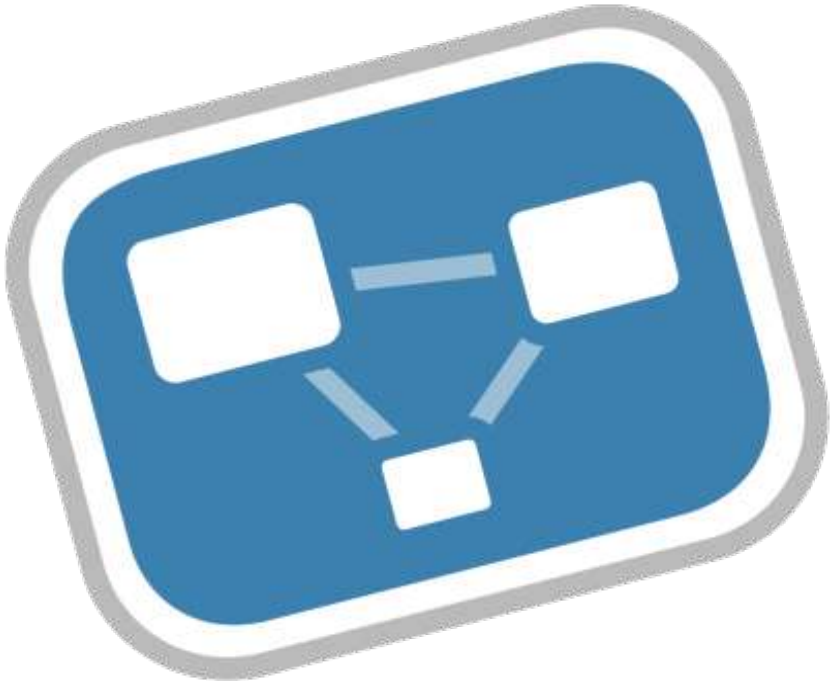
Set-based concurrent engineering in Open Source software development



A case study of work in freedesktop.org-related communities.

Sebastian Kügler, 9801278
Nijmegen School of Management
Katholieke Universiteit Nijmegen

I. Table of Contents

Foreword

The last decade, the Open Source movement is increasingly gaining momentum in the software market. Driven by high technical standards, and the idea that software should be free – not only free as in free-of-charge, but as in free speech and freedom[1] - more and more people, individuals as well as fortune 500 companies, adopt a new way of creating the software they need.

This thesis deals with the very special way, free software is developed in a community effort and tries to have a look at it in the light of a theory, that has been developed in a completely other context – car development.

I have chosen this subject, because I am on the one hand very interested in this way of developing software, a way where deadlines and commercial goals do not play an important role in the first place, but where a user centric approach is taken advantage of. On the other hand, there is not much research done in how far these collaboration effort fit today's theoretical approaches to organizational sciences, maybe I can add my two cents in this way.

I would also like to take this opportunity to acknowledge the help of a couple of people: Ton van der Smagt for providing me with productive critique on scientifical issues, Kim Kerckhoffs for her patience and methodological expertise and certainly not at least Ronald Bultje and Simon Edwards for taking the time to answer my questions. I want to express special thanks to my beloved mother for “just” being a good mother.

And then, for who may be interested in it, this thesis has been written exclusively using free software. I would like to thank the people who work on the following projects: The *Linux* operating system as a basis for the *Debian GNU/Linux* software distribution, the developers of *Openoffice.org* for making an excellent office suite available, the developers of *the GIMP for their* a compelling graphics suite, and not at least the *KDE and freedesktop.org* people for tying all the pieces together to a desktop, I am happy to be able to work with – all you hackers did a great job!

And now ...

"Talk is cheap. Show me the code."[2]

I hope you will enjoy reading my thesis, and that it will broaden your view on how productive collaboration can be accomplished.

Sebastian Kügler, July 2004

1 http://en.wikipedia.org/wiki/Free_as_in_beer

2 Linus Torvalds, 2000, http://lkml.org/lkml/2000/8/25/132

1. Introduction

1.1 Introduction

In this chapter, I will introduce the two players in this thesis. First, I will give a short impression of what set-based concurrent engineering deals with, and how it is related to the development work freedesktop.org. Then, I will, in short, give a description of what the Open Source model is about. Some words about what freedesktop.org is will finish this chapter.

1.2 What is set-based concurrent engineering, why SBCE?

Set-based concurrent engineering (in the course of this thesis also referred to as "SBCE") is a development technique invented by Toyota product developers, which focuses on collaboration between different development departments and aims at shorter development times with an increased quality level by improving collaboration and by paralleling parts of the development process.

SBCE is chosen as a theory to answer the research objective, because issues such as a paralleled approaching to development work and different collaboration play an important role in Open Source software development.

SBCE is likely to provide a good toolset to answer the research question, because on the one hand, SBCE is a technique which aims at improving team work in development tasks (concurrent engineering), and bundling workforce for projects which closely matches issues of Open Source software development. On the other hand SBCE aims at high quality solutions which is also of very high importance in Open Source software development. Both focus on efficient collaboration in groups of developers and in providing high quality solutions.

1.3 The Open Source Model

The *Open Source Model* stands for a way of developing Open Source software. The real inventor of the *Open Source model* might be Linus Torvalds, who has introduced the use of extremely short release cycles and close interaction with the user in the development process. The most popular paper about the Open Source model as used in Linux development, written by Eric S. Raymond, has been published under the name *"The cathedral and the bazaar"*. It describes the differences between the new, decentralized and user oriented development model, in contrast to centralized ways of development, as they are used in software development by companies applying closed source development techniques, such as Microsoft.

1.4 What is freedesktop.org?

KDE and GNOME are desktop environments for UNIX operating systems. A "desktop environment" is a program that enables the user to do many different things with his computer, but in a consistent way. Desktop environments can be seen as a "dogma" of how things should be done, and how they should look and feel. So, both, KDE and GNOME, provide features to work with the computer via a graphical user interface. Freedesktop.org is an umbrella organization set up by developers of the KDE and GNOME projects. Freedesktop.org is a community collaboration effort and facilitates software development of related projects. Paradigms handled by freedesktop.org are the separation of components and the application of modular development using standard tools. Freedesktop.org aims to establish strong standards to improve the interoperability

between different desktops, such as KDE and GNOME. One more political aim of the freedesktop.org project is to move control to the developers.

2. Research framework and conceptual design

2.1 Introduction

In this chapter, I will outline the steps I will take in order to realize the research objective. I will first give the research objective, along with the research object and the research questions. The following paragraph discusses the meaning of some of the most important terms used, followed by the nature of this research and a visualisation of the conceptual model it is based on. The chapter is finished mentioning the methods of data collection along with an overview over the sources for the research data.

2.2 Research objective

I have formulated my research objective as follows:

> *"The objective of this research is to make recommendations to improve the collaboration between freedesktop.org-related developers by testing it on the principles of set-based concurrent engineering in order to make their work towards the freedesktop.org goals more efficient."*

2.3 Research object

My study is carried out on the development community of freedesktop.org and on the principles of set-based concurrent engineering. Consequently, there are two research objects. I will analyse the development process of freedesktop.org in the light of the principles of set-based concurrent engineering and will relate the work of freedesktop.org developers to the theory of SBCE in order to derive proposals from this for improvements in the collaboration between freedesktop.org developers or, in other words, for a more efficient development process

I will take an example for an Open Source Software development community, and I will do research whether principles of SBCE are already applied, and where not, I will have a look at what the merits and disadvantages of SBCE could mean for this community.

I have decided to have a closer look at freedesktop.org. Freedesktop.org is a project which has the implementation of standards for Open Source desktop software as its goal. Technical, as well as political issues play an important role in this context. Also, freedesktop.org is supposed to deliver strong standards, which means that there is very much potential for conflict, since freedesktop.org's standards will have influence on a lot of different actors in the Open Source sector. Freedesktop.org focuses on the development of standards for desktop software, for example for UNIX-like operating systems.

2.4 Research questions

" Which SBCE principles can contribute to improving the collaboration between freedesktop.org developers? "

Or, to put it in another way:

"What can be improved in freedesktop.org's work in order to make it more efficient with respect to the project's goals?"

I will try to answer this question by having a look at different aspects of the problem, for example:

- timeliness of the development process (as well as maintaining flexibility of the product),
- collaboration,
- integration of subsystems,
- resolving of conflicts,
- structural issues, such as modularity.

2.5 Operationalisation

This paragraph contains a small abstract of central terms used in this paper. For a more detailed description of especially the SBCE principles, please refer to the theoretical framework.

2.5.1 Collaboration:

In this paper, I will use the term collaboration as *"working together with one or more others towards the same objective"*.[1]

2.5.2 Freedesktop.org-related developer:

Freedesktop.org-related developers are people, who regard themselves as working on one or more projects, that form a part of the freedesktop.org or works towards its goals. Work, or to put it in a less specific way, contribution to the goals of freedesktop.org can be done in the following forms[2]:

- Collection, promotion and integration of standards and documents related to X desktop interoperability (X being the standard protocol for graphical systems on Unix derived operating systems.)
- Work on the implementation of such standards
- Provision of resources and services to facilitate the above forms of contribution

To put it more general: freedesktop.org related developers are people who are working on software which contributes to the goals of freedesktop.org and who are actively being part of the community. Being part of the community can be operationalised as communicating with other developers and users, contributing to freedesktop.org software either in the form of code, bugfixes, ideas or end user support.

2.5.3 Principles of SBCE:

The principles of set-based concurrent engineering dealt with in this paper can be shortly described as follows:

Mapping the design space is about a way to create an idea of what the solution will 'look like'. A functional division of the solution and prototyping are parts of this principle.

Conceptual robustness has to assure that the subsystems and preliminary and partial solutions chosen for need little changes when other parts of the solution are altered. Modularity is a good example for this.

1 http://en.wikipedia.org/wiki/Collaboration
2 http://freedesktop.org/Main/MissionStatement

Integration by intersection deals with how the different subsystems will eventually be integrated, and how developers get an idea of constraints posed by other related subsystems.

Feasibility before commitment can be seen as a result of the above principles. Ideally the solutions chosen for the subsystems will fit together when integrated, because the overlapping problem space has been well-defined, and the interfaces between subsystems, and constraints subsystems pose to each other have already been dealt with.

2.6 Nature of the research perspective

The objective of this research is to make recommendations to improve the collaboration between freedesktop.org-related developers by testing it on the principles of set-based concurrent engineering in order to make their work towards the freedesktop.org goals more efficient.
The theoretical literature about SBCE form the assessment criteria for the case study. The interview guides are derived from these criteria.

This research can thus be characterised as practice-oriented evaluation research.[3] Also, this research project is designed as a case study, since there is only one research unit, namely the freedesktop.org developers, which is observed in its natural environment, the community it is part of.
Two methods of data collection will be applied: first, the Open Source community is observed in a participative way. Different mailinglists have been followed in the course of more than a year. Also, other means of communication between developers have been used. IRC (Internet Relay Chat), websites and discussion groups (Usenet) are among the most important ones. Second, interviews with developers provide insight in more tacit information, such as how communication works, and in which way design principles are dealt with.

This research will mainly have a qualitative character, as the data used does not have very much quantitative value, and the type of this research does not fit into the framework of quantitative research. Qualitative research methods, such as in-depth interview and focus groups provide the information needed, which, compare to the SBCE principles can be the basis for answering the research question, and realizing the research objective. Also, this kind of information, in contrast to quantitative information illuminates meanings people assign to social phenomenons, that are essential for answering the research questions. Rich and detailed data, such as the data derived from interviews give a reasonable detailed impression of the development process, as well as the opinion of the developers.

In short, we can say that this research is a case study because it fits the following characteristics[4]:

- Small number of research units, actually just one, namely the freedesktop.org developing community
- Labour intensive data generation: gathering of data is done via interviews, studying of mailinglist archives and following development processes discussions for more than one year
- More depth than breadth: one object which will be dealt with, freedesktop.org
- A selective, strategic data sample: sample data is what can be found about the collaboration of freedesktop.org developers, the sample is based on two criteria: membership in the freedesktop.org developers community, opinions and observations on issues of SBCE
- Qualitative data and research methods, for example interviews

3 Verschuren, Doorewaard, chapter 2
4 Verschuren, Doorewaard, p. 165

- Open observation, on site: the developers are being interviewed and observed in their natural environment, for example interviews are held via the usual communication channels, the mailinglist and personal e-mail
- This research also carries many characteristics of a participative observation, since the data is being collected from within the community,
- Habits and communication channels of the community have been utilized,
- The researcher regards himself as an active part of the community and
- As part of the community, the researcher will be able to get and interpret information, that would otherwise be useless for a non-participative researcher (for example highly technical discussions, which also mirror specific patterns of behaviour, such as dealing with conflicts).
- A lot of data used in this research is not existing in an explicit form, since the research object is a very dynamic, self-organizing one.

Although there is always a certain tension between the two roles – researcher and participant – the observations are not too biased. On the one hand, the community itself has no interest in not showing certain behaviour, due to the open character. On the other hand, manipulation of data is clearly not desirable, since the researcher has a certain interest in the validity of the data and the research.
I will answer the research question by observing such a loose virtual community working together and analysing in what way it fits the principles of a theory on collaborative design, and in how far the theoretical framework can be applied to the case studied.

2.7 Visualisation of the research framework

The figure on the next page has to be interpreted as follows[5]:

(a) The first part is the formulation of the ingredients from which my research perspective will be developed, in this case study the principles of set-based concurrent engineering.

(b) The second part indicates the research object the research perspective will be applied to. Also, the ingredients from the research perspective are given in more detail in order to fit the aggregation levels of both, the research object and perspective.

(c) The third part indicates in which way the confrontation of research perspective and research object contributes to a solution for the research problem.

(d)The last part is the research objective, which is reached by following all earlier steps in the process.

As an important side note, this is not a literally serialized process, the research progress is not in the first place an indication of time, but an indication of progress towards the research objective. Therefore, all steps may be altered in the course of the research process in order to maintain a consequent process design. This technique is also know as iteration.[6]

5 Verschuren, Doorewaard, p. 56
6 Verschuren & Doorewaard, p. 21

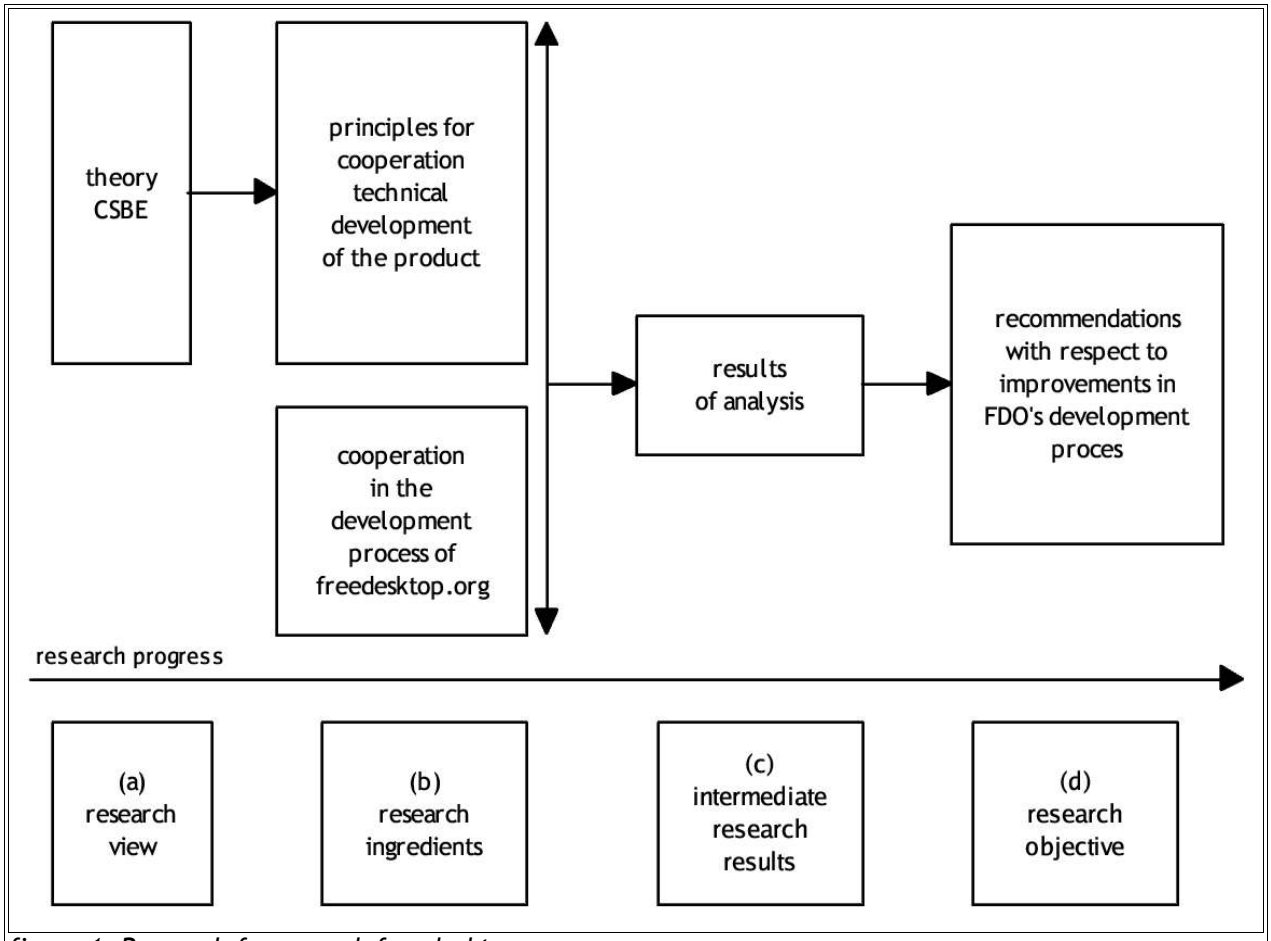


figure 1: Research framework freedesktop.org

2.8 Data collection

There are different papers about set-based concurrent engineering, that will be used for this research. Mark Klein, to give an example does research at the Massachusetts Institute of Technology and has written some interesting papers about SBCE.

Information about all projects is openly available on the Internet. In particular the following sources can be used to gain detailed information on the subject:

- mailinglists archives, such as http://marc.theaimsgroup.com/, especially different lists dealing with freedesktop.org, as well as general discussions about the subject
- other communication channels, such as message boards and user forums
- websites of related projects and organisations (e.g. KDE, GNOME, gstreamer)
- interviews with developers and related people (e.g. Ronald Bultjes, Simon Edwards), via e-mail or chat

More depth will be achieved by making use of different sources of data, triangulation of sources.[7]

7 Verschuren, Doorewaard; p. 164

3. Theoretical framework

3.1 Introduction

Set-based concurrent engineering is not some sort of cookbook for designing the engineering process, it is much more a set of principles which support engineering in teams or even larger groups.
Many organizations are looking for a recipe for design processes, an efficient and flexible design method that will lead to high quality products. Recent research indicates that the flexibility of the development system is a key point to success, particularly in rapidly changing, unpredictable environments. There are different approaches to concurrent engineering, one of them being *set-based concurrent engineering* (SBCE), a development technique invented by Toyota.
Current techniques of concurrent engineering have to suffer from constraints such as time-consumingness. Also, they are very often based on a serial and iterative workflow, which does not allow much flexibility.
SBCE is a design technique suitable for large groups of experts, each of them with different expertise. A central issue in concurrent engineering is how conflicts are solved among different experts.
In this chapter, I will give an overview of different issues related to set-based concurrent engineering, describing the approach in detail. For brevity, some parts of the theory around SBCE have been left out, in favour of a more in-depth view of others. First, I will give an overall idea what SBCE is all about, then I will elaborate on different aspects of SBCE, such as mapping of the design space, what role conceptual robustness plays, how collaboration is accomplished in the light of integration of the respective works, how feasible solutions for all related parties are designed, and last, but not at least resolution of conflicts. The chapter finishes with comparison between SBCE and point-based approaches, followed by an abstract of the issues dealt with in this theoretical chapter.

3.2 The SBCE approach

SBCE, in contrast to point-based approaches starts with a very broad design space, and tries to keep it that way as long as possible, while gradually eliminating infeasible or 'weaker' solutions. The design space is narrowed in the course of development and the best solution will be left in the end.[1] This could also be seen as a process of evolution of the best possible solution. The problem of time-consumingness is also an issue with SBCE. SBCE takes more time at the beginning of the process, because the broader the design space is (and thus the more possible solutions there are), the longer it will take to work out a solution, because the requirements to fit a broader set of solutions are much higher than the requirements which only fit to a very narrow (set of) solutions. However, this usually pays off in the course of the development process, as stepping back in the development process does not necessarily mean that all activities based on the preliminary disapproved solutions are rendered worthless. The chance that earlier activities are still of value is much higher when the solution space for which the work has been done is still applicable due to its broader nature. This manner of working might in short be described as “dismiss other solutions as late as possible”. This principle does not mean that decisions must be delayed as long as possible, but it means that the development process is not about finding a final solution as fast as possible in order to eliminate others.
The trade-off in this case is that on the one hand, concentrating on one solution has the merit that a narrower solution space might be less work-intensive to work out, but on the other hand that changes in the product require much more work. Design being a mostly iterative process,

1 http://web.mit.edu/wfinch/www/research/set_based_design/

application of this technique leads to – if correctly applied – generally more flexibility in the course of development, and thus does put much less constraints on the result, as a wider range of solutions is kept within the solution space throughout the whole process. To put it short: iteration (which is inherent to nearly all development processes) will be far less expensive.
This lead, in the case of Toyota's design process to an overall decrease of development time (and time-to-market) and increased overall efficiency in development activities.

There are some broad principles which Toyota applies to realize these results[2]:

3.2.1 Mapping the design space

First all functional departments related to the development process identify a solution space for themselves, independently from other departments. These functional departments base their decision on design constraints for their respective subsystems on experiences earlier made, on analysis of the problem, on testing, and nonetheless on information from outside. To explore which one is the best solution and to get a better idea of trade-offs inherent to different solutions, prototyping and simulating alternatives is one of the major tools to narrow down the solution space within their subsystem. An also very important issue is that communication between departments (or developers of subsystems) is based on "design spaces", not on single ideas of the result, so the discussion is kept really vague and abstract, at least in the beginning.

3.2.2 Striving for conceptual robustness

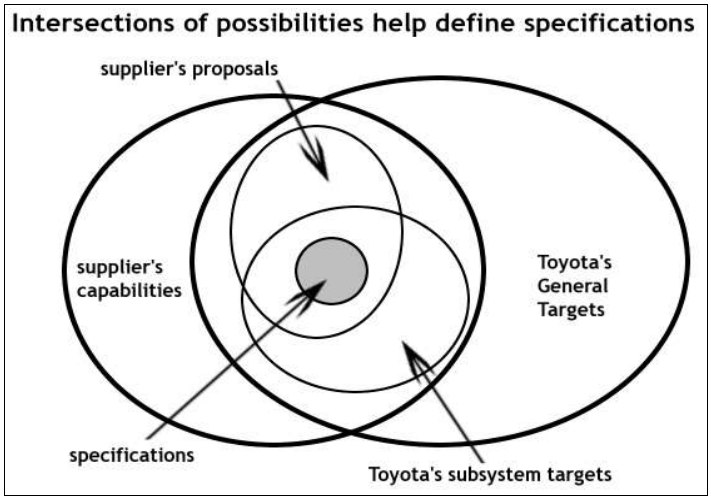
Conceptual robustness is a matter of a design remaining functional after variations in its environment. Vulnerability of a system to changes depends on a lot of different factors. One question that has to be asked is "*Will the product still fit the solution space over some time?*". So the development time, and thus the time-to-market plays a central role. Short development cycles and utmost flexibility help to realize this goal. Also, there is great need for standardization to decrease the necessity of *reinventing the wheel* for a couple of times, and make common appointments on concepts and techniques used in the development process. The following citation should give a good idea of what this conceptual robustness is about:

> *" [...] create designs that work regardless of what the rest of the team decides to do. If one function can create a design that works well with all the possibilities in another function set, it can proceed with further development without waiting for information from that function. Such conceptually robust (Chang, et al., 1994) strategies can collapse development time significantly while providing other benefits such as ease of module upgrades and serviceability."*[3]

3.2.3 Integrating by intersection

While the first principles concentrates on the respective subsystems, this principle poses the question how these subsystems will eventually work together. How are the parts integrated to meat each other at the point that will later be regarded as the "*best possible solution*"?

2 Sobek, Ward, 1996
3 Sobek, Ward, 1996; p. 7



As the respective designers of subsystems begin to understand their own constraints and get a more clear idea of their own part of the solution space, it's time to have a look beyond their department. The key to this integration lies in the overlap of feasible design spaces of the different subsystems. These intersections are more or less directly translatable into a solution that is acceptable for all. Toyota's effort at that point is to impose minimum constraints at that time, in order to ensure flexibility in the remaining part of the development process. Decisions are taken when the necessity arises for developers to know what to go on with, or when it gets clear that one part of the possible solution space can be ruled out without having impact on the remaining solution space. Furthermore a decision once taken has to be respected. Taking decisions at a later time in the course of development does not only mean that flexibility is maintained, but also puts more weight to the decision made, as more time has been invested to solve that particular problem. Conceptual robustness is of much importance as conceptual robust strategies can decrease the development time significantly. An examples for conceptual robustness is modularity, which has the benefit of easy upgrading and less complicated altering and upgrading of the respective modules.

3.2.4 Establishing feasibility before commitment

The whole design process requires that concepts are feasible before they are being committed. The way this is handled at Toyota's simply begins with a series of prototypes. Multiple concepts are considered in parallel, gradually eliminating infeasible ones. This avoids problems later in the design process, and saves lots of both, costs and time when problems arise. The fact *that* problems will arise at some point can be surely assumed due to the great number of uncertainties throughout the whole design process. Also, Toyota tries to prevent important changes by keeping as much as possible concepts within the solution space. Modular design is one means to accomplish greater flexibility with respect to product design.

Toyota tries to decrease uncertainties in the development process by checking sets of solutions for feasibility, and for different types and levels of uncertainty. Concepts, that still are under consideration are observed at mainly two criteria:

- Why is that problem still under consideration, why is it not proven (in)feasible yet?
- What role does this problem play in the whole product, what is the impact of later consideration on the whole design process?

Answering these two questions gives a decent idea of how important a solution to the particular problem is, and what has to be done to accomplish the task that this part of the solution deals with.

3.2.5 Conflict handling

Naturally, in an environment with different experts involved in decision making, conflicts will arise at some point. Set-based concurrent engineering handles a so-called “client assisted design advice system” in order to solve the conflicts and come to a decision that meats the

requirements of the customer. Principles handling in negotiations are equality between all related parties, avoiding of situations that cause great dissatisfaction to some party and equality of priorities of different points of view.[4]

While conflict decision usually depend on several interdependent issues, the solution space is very complex and too large to be explored exhaustively.[5] There are two types of subsets of problems in conflicts in set-based concurrent engineering:

- Competition vs. cooperation
- Domain level vs. control level conflicts.

In competitive conflict situation, each party has mainly its own benefit in mind, whereas cooperative conflict situation aim at achieving a globally optimal solution for the superordinate goal. The SBCE model is oriented towards cooperative conflict resolution, since that is the most appropriate for cooperative design where the shared goal of producing the best possible products exists.

Domain level conflicts concern the actual design of the product, whereas control level conflicts deal with the overall direction, design decisions have to take.

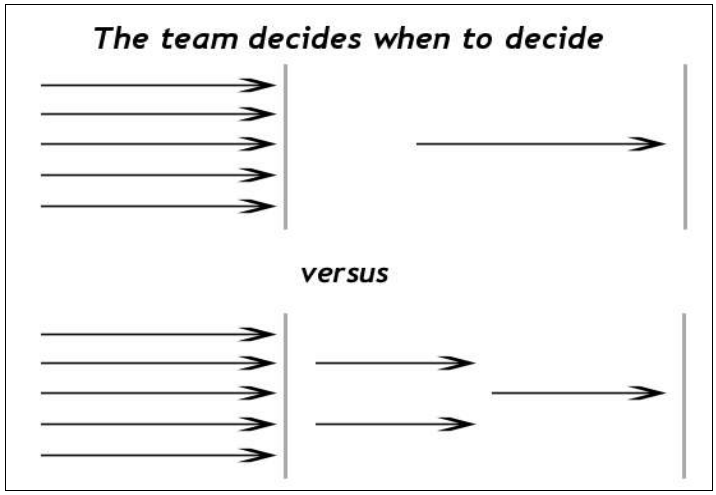
Analysis of human conflict situation has has helped to understand human conflict and describes a few prescriptions for how conflict resolution can be facilitated. Formalisms for conflict resolution include the following:[6]

- Development-time conflict resolution tries to rule out conflicts by exhaustive discussions when they are developed. It has the disadvantage, that it's very time-consuming in the first place and assumes that conflicts generally can be resolved by discussion.
- Knowledge-poor run-time conflict resolution waits until the conflict arises, and then tries to handle it and take measures to avoid the specific situation in which the problem causing the conflict will arise.

3.3 SBCE in contrast to point-based design approaches

Traditional engineering approaches the problem by first generating a set of possibilities, and then weighing these possible solutions against each other. Then the most compelling idea is chosen and developed. If this solution turns out not to be feasible, the process as a whole has to be started over.

SBCE's approach is to develop a set of possible solutions, and gradually reduce the number of them while working on the project. Weak concepts are eliminated in the course of the process as they turn out to not fit in the solution space any more. The goal is to keep one concept in the end, but not throw out concepts that seem to not fit the solution space too soon. This might be more work-extensive in the beginning but guarantees much more flexibility throughout the whole process.



4 Bahler, Dupont, Bowen, 1993
5 Klein, Faratin, Sayama, Bar-Yam, 2003
6 Klein, 1990

The cost of taking back a decision earlier made is much lower, so there is more room to improve the concept while developing it. Wrong decisions in later phases of the development process do not have that much impact on cost and are far less time-consuming than if these decisions would have been made in the beginning.[7]

3.4 Abstract SBCE

Set-based concurrent engineering is a development technique that gives an alternative approach to classic development techniques, which are mostly point-based. SBCE approaches the development process in a more abstract and flexible way than point-based approaches do. SBCE has great advantages when it comes to development in inhomogeneous and complex environments, where lot of different subsystems are touched and more developers or groups of developers are working on. SBCE is a set of principles, which, once applied to a development process, should shorten overall development time and offer more robust and less conservative results in the end. Principles of SBCE deal with subdividing the problem- and solution space, communication about interaction, integration of subsystems and strong concepts.

7 Sobek, Ward, 1996

4. Development work in freedesktop.org-related communities

4.1 Introduction

In this chapter, I will give a detailed description of how software development in Open Source communities works. First I will elaborate on the structure of collaboration efforts, followed by information on hierarchies. Then, I will discuss how complexity is reduced, and that role modularity plays in terms of conceptual robustness. Assuring quality will be the next subject. I will describe the most important communication channels and tools used, to give an idea how the above issues are implemented.

4.2 How is work and collaboration structured?

Early and frequent releases are a critical part of the Linux development model. This way, all developers and interested parties are able to keep up with recent development, even if they do not have access to the resources the 'main' developers use.
Also, the *'"release early, release often"-paradigm*[1] prevents lots of duplicate work and thereby increases efficiency. It actually never seems to be an issue in the development process, due to the developers propagating fixes quickly.

Despite of the efforts of preventing duplicate work, multiple implementations of the same thing are not regarded as a waste of time. In fact, having multiple implementations has a couple of advantages over being bound to just one implementation:[2]

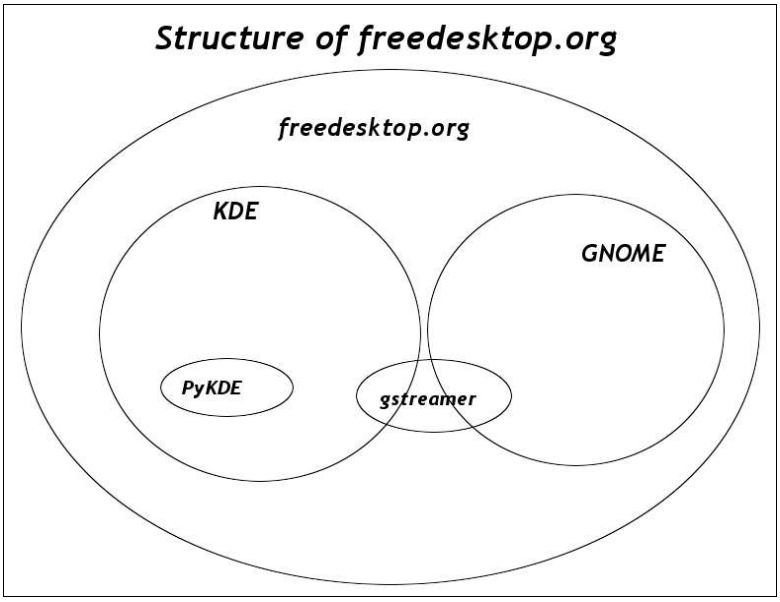
- Risk of experimentation is reduced. Experimental changes are far less likely to have a disastrous effect since there are still alternatives to fall back on.
- Diverse participation. Multiple projects allow the developers to choose between the projects they want to work on.
- Coping with conflict. In a centralized system, there is nearly no room for conflict. Handling multiple implementations makes it possible to overcome conflicts on a personal level, that would make it impossible for someone to work on a certain project. Sooner or later, one of the ideas would have to be given up in favour of another idea. In the Open Source model, these sorts of conflicts have often lead to a rearrangement of workforce, and more diverse choice for both, users and developers.
- It is very hard to know in advance, which solution will be the best in the future. Ideas that might at present sound very stupid might later be '*just what you need*'. The fact that multiple implementation is natural to Open Source development also keeps doors open in the course of further development. Not only does it stress the necessity of applying strong standards (which is regarded as a good thing), it does also put pressure on the various implementations to use well defined interfaces and offer compelling features. Not at least, it creates room for innovation.

Having multiple implementations, and being reluctant to be bound to one single peace, it is important to handle strong standards. The KDE and GNOME projects are good examples for this. KDE and GNOME are in fact concurrent products, that is, they do the same in a different way. GNOME is a more clean approach to a desktop environment, KDE offers lots of features. Freedesktop.org is an organisation that facilitates common standards both, KDE and GNOME use, making them interoperable.

1 http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html
2 http://cbbrowne.com/info/lsf.html

4.3 Hierarchies

Although the Open Source development model is a very decentralized one, there is still some hierarchy. Groups of developers work together in a more or less tight community. While the borders of these communities are usually very loose, that is the communities are open with respect to what they are doing, what issues are discussed and how. Open Source communities, especially, but not exclusively freedesktop.org-related ones, have a lot in common with ad-hoc networks. Links are established when needed and vanish at the time they are no longer in use.



Freedesktop.org is a sort of umbrella organization for all kinds of desktop-related projects. Projects, under freedesktop.org are KDE, the GNOME project, but also smaller projects like gstreamer, PyKDE and guidance. PyKDE is really bound to KDE, whereas gstreamer will be shared between both projects. Within these projects, there's also some hierarchy. The head of KDE, for example, might be the KDE core team. This core team provides developers with coordination for releases, but also makes policy decisions about things that can be added to KDE. The core team is also responsible for the managing of the technical infrastructure and libraries, that almost every KDE developer builds his or her applications on top of.

Every project has a so-called maintainer, who is the one who will take the final decision in the end. However, maintainers of projects are very dependent on their co-developers, so they won't easily take a decision which is not accepted by either other developers or upstream maintainers, since both could endanger efficient work on the project itself. Usually, conflicting issues are discussed within the closer project community, decisions are based on consensus. Where it comes to issues, that are also dependent from other projects, the appropriate mailinglist will be contacted in order to coordinate the efforts.

Maintainers of projects are mostly persons who are not only technically skilled with respect to the project, but also have a reputation of conflict resolving capability. These two sides accomplish respect by other developers on the one hand, and communicate a strongly technical point of view within the community on the other hand. Also, maintainers are the ones who contact upstream maintainers and handle propagation of (parts of) programs that will be integrated in other, coordinating projects. A maintainer usually plays different roles, on the one hand he facilitates development of the project by determining release schedules, taking decisions whether to integrate certain features or abandon ideas. In most cases the maintainer is the main developer, and as a result of that is very skilled with respect to the project. Also, the maintainer will provide feedback to co-developers as to why certain decisions are made in that way and not another and what is important in the context of the project and why. A participative style of leadership is necessary to accomplish the job in a good way. The thinking process of the group of developers resembles the one used in Delphi techniques.[3]

3 Zolingen & Klaassen, 2002

While the structure is kept pretty flat, it still has to suffer from problems natural to a hierarchical system. The case might also be that changes that would improve the system (at least according) to a group or certain developers may not make it into the mainstream because authoritative approval of some maintainer is not given. In the end, if all communication does not lead to an acceptable result for both parties, one can just use the source code and start its own version, independently. This is called a fork and is the common way to overcome conflicts that could not be solved. Allowing forking is granted by all Open Source licenses and is natural to Open Source software. Still, since the maintainers are regarded as highly competent people, it seems that this way of handling decision making is very useful with respect to code quality and structural issues of the respective project.

4.4 Reducing complexity

In traditional software development, the complexity and overhead of bigger groups of development grows exponentially to the number of developers. Reducing complexity is important to Open Source development in two ways. First, complexity of communication in very big and rather loose community, second with respect to the code and problems that have to be solved with it.
Raymond states, that this complexity is based on the assumption, that the complexity of the communication structure in a big network where every developer has to communicate with all other developers would overextend all possible communication channels.[4] Open Source projects generally work in another way. By dividing bigger projects into parallel subtasks, this necessity vanishes in favour of smaller development groups with very little mutual interaction. Changes in either code or interfaces shared between the parallel subdivisions is passed through via the coregroup. Effectively, only the core group is hit by the full overhead. A good modular design reduces the amount of communication needed to make the parts work together. The interfaces and standards used are discussed in a Delphi like approach and in that way take advantage of the Delphi effect.[5]

Complexity in technical sense is reduced by means of good engineering, with information hiding and encapsulation being the most important principles.

To reduce the complexity given by the nature of the problem to be solved, especially in agreeing on standards and implementation details, Open Source development takes advantage of something that might be closely related to the 'Delphi effect'. A Delphi approach is a famous approach to deal with complex, ambiguous and 'messy' problems, such as the development of software in our case. Simon Edwards explains this in short as follows:

> *"Encapsulation and information hiding mostly. Divide and conquer. Divide systems with encapsulation into small parts that can be separately understood and modified."*

Explaining information hiding[6] and encapsulation[7] in detail would be too complex in this context. I would advice the interested reader to read a good book on principles of object oriented programming (OOP). Both (closely related) techniques are used to make the definition of interfaces easier. The technique of object oriented programming is widely adopted, not only in the Open Source world.

4 http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s05.html
5 Zolingen & Klaassen, 2002
6 http://en.wikipedia.org/wiki/Information_hiding
7 http://bepp.8m.com/english/oop/encapsulation.htm

4.5 Conceptual robustness by modularity

In Open Source development, the key point to conceptual robustness is modularity. A functional division of the problem space into modules that provide a generic solution is a vital part of good engineering and as that being an important paradigm handled in daily work. Strong interfaces, a straight-forward and maintainable structure make it possible to work on different modules in parallel without having to take every single change in other modules in account. Encapsulation and information hiding keep working with modules relatively simple. Breakage of existing modules and interfaces is generally not allowed. Ronald Bultje's opinion on not coping with standards stresses that:

> *"Very important, since an incomplete modularity would lead to unusable libraries. A design is simply a solution for an issue, so the proposed design needs to solve the described issue correctly and completely. Good designs do this, and try to not redesign anything that already exists unless it's supposed to improve that."*

A modular design based on strong standards has two major advantages. First, it facilitates code reusing, second it makes the respective module less dependent on the main program and easier to understand, improve extend and possibly replace.

4.6 Assuring quality

"Given enough eyeballs, all bugs are shallow"[8], also called "Linus's Law" is a popular paradigm in the Open Source movement. It simply means, that by means of peer review, as long as enough people review other's works, bugs will be found and evaporated in that way. Peer-review is the key to quality in Open Source development. In most cases, changes are either made public via mailinglists, where patches are submitted that can then be discussed and eventually get integrated in the main development source code tree. Also, some – for this project – more important developers, such as maintainers, have access to the source code tree via a version control system, which means that they can directly merge their changes. Simon Edwards, KDE developer and states *"Personally I watch what other people put into CVS/SVN, and if I see something that is a bit strange then I ask them why they did it that way, and also suggest that they do it some other way because of XYZ."*. This actually gives a good idea of how peer-review works in a lot of Open Source projects. Working on a project also means, that the particular developer will take a part of the responsibility of others by reviewing their changes.

8 http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html

But apart from peer review, there are other means of assuring quality. Testing is one of the more important. Short release cycles enable developers to get short term feedback of their work. The Open Source principle *"Release early, release often."*[9] facilitates that. Short term feedback has the major advantages that bugs will become visible soon after they have been integrated, so in most cases it is easy to fix them since only a small amount of work has to be redone. This "*Release early, release often*" principle also motivates developers and users to keep track of what is changing, it makes reviewing from outside the core developing community much more easy. A lot of projects even make it possible to access the recent source code on a daily basis. These daily releases are usually called snapshots. Other projects go even further by allowing anonymous read access to their CVS servers. That makes it possible to follow development in real-time. There is one more advantage to frequent releases: if users want to submit a bug, they are able to check if this bug has not already been resolved by downloading or having a look at the most recent (CVS or snapshot) version of the code. In that way, developers will get less worthless bug reports, since users themselves are enabled to figure out if this bug report still applies to the very latest version, or if they can solve the problem themselves by upgrading to the newest version available, or that they just have to wait for the next release to propagate a fix for their problem.

The *"deadliness of deadlines"* also plays an important role. Where developers are forced into a strict timeline, combined with an immutable feature list, the only mutable variable in the equation[10] is the quality level of the work. Therefore most Open Source projects don't use hard deadlines. This always leaves the possibility open to take the time to proper fix a certain problem. As a side note, bigger projects, like KDE use roadmaps, or so-called "release plans" which express the intention of the projects maintainers which steps to take at what time to get software released. They stress, however, that this is no hard date and state that the data are subject to revision. No promises are made with respect to release dates.[11] Obviously, a lot of users are interested in the release dates of a certain project. In most cases, when someone asks for the date of the next release, he or she will get the answer *"When it's ready."*.

So *Quality Assurement* in Open Source development communities is done by peer-review in the first place. The openness, transparency and involvement of users also plays an important role.

4.7 Communication channels and tools

In this part, I will describe some of the most frequently used communication channels, and give their purpose, in order to provide better understanding of how such a community works on an operational level.

4.7.1 Documentation

The probably most important piece of information a developer can get over a certain piece of software is its documentation. The documentation usually consists of a description what the respective software does and in most cases also how it accomplishes that, such as detailed of the approach to a problem chosen by the developer. Then, there are installation instructions, usually along with a note on licensing.

For developers, there are also a couple of guidelines. These guidelines should provide orientation on issues that appear on a regular basis and are usually intended to streamline the work of different parts of the community, or even between different developers, all having their

9 http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html
10 Features x Quality : Resources x Time
11 See, for example http://developer.kde.org/development-versions/kde-3.3-release-plan.html

own style of doing things. Popular examples are the *GNOME human interface guidelines*[12] and the *KDE User Interface Guidelines*[13], which provide the developers with a set of rules-of-thumb to create a unified and consistent user interface.

4.7.2 Source code comments

Developer who want to get to know a certain piece of software better, usually will have a look into the sourcecode. Developers will try to keep their source as clean and clear as possible in the way that another developer could quite easily read the source, understand what is actually does, and how, and then be able to fix bugs, develop more features or just use the code as a basis for own work. Peer-review also goes about checking the 'readability' of the source code written by others. Naturally, developers will not comment on every line of code they write, but it is also not unusual that there are actually more lines containing comments on the code such as explanation as to how and why a certain part of the program is written in this way and not another way that may be more clear and obvious at first sight. There are also tools in some programming languages that can build a documentation tree from the source code written.

4.7.3 Mailinglists, Usenet and e-mail

Most of the project related discussion is done via mailinglists. Mailinglists usually consist of one central e-mail address where questions, suggestions, announcements and patches can be sent to. An e-mail sent to a mailinglist will be forward sent to everyone who has subscribed him- or herself to this mailinglist. Mailinglists are quite flexible when it comes to representation of the discussion. Also, mailinglists do only require the receiver to be able to receive e-mail. In that way, the technological threshold for people to take part in the discussion (or simply follow the discussion) is very low. Also, there are searchable mailinglist archives, which offer the possibility to look up problems and issues that have already been subject to discussion, and in that manner reduce the volume of information the mailinglists (and its subscribers) have to handle. Mailinglists appear in the form of threaded discussions, that means that the discussion is structured in a tree like manner. People can add answers to questions, but can also comment on answers already given. A typical visualisation of a threaded discussion is shown in the illustration. Every line stands for a different message on the list. Lines that are indented at the same level are reactions to the same message. Further indentions annotates that the respective message is a reaction to a message one level higher. Users of mailinglists have a certain interest in not polluting their information channels. There are a couple of different mailinglists for every kind of topic. Examples for mailinglist topics are

Illustration 1 example of a threaded discussion on a mailinglist

- kde-devel@kde.org - for application developers (both applications in central KDE packages and contributed applications
- kde-usability@kde.org - for discussion and fixing of usability problems in KDE through the KDE Usability Project

12 http://developer.gnome.org/projects/gup/hig/
13 http://developer.kde.org/documentation/design/ui/

- fontconfig@freedesktop.org – for discussion dealing with fontconfig, the most popular font rendering libraries
- freedesktop-announce@freedesktop.org – for release announcements of new software
- Gnome-infrastructure@gnome.org – for public discussion about gnome.org services

to just name a few and give an idea about the various topics mailinglists in freedesktop.org-related projects deal with. This also applies to Usenet and its newsgroups since mailinglists and newsgroups are technically and socially closely interrelated.

4.7.4 IRC and Chat

IRC or to put it more general, chat sessions and channels are useful to provide direct feedback and discuss in a very close timeframe. The biggest IRC network for Open Source software development is freenode.net. The thematical division of this chat network can be compared to the one mailinglists have. Most bigger projects also have different IRC channels, where topics can be discussed, but where also users can 'come along' and get help with specific problems or just ask short questions. Developers can take part in different chat sessions at the same time. Usually, chat sessions are much more informal than mailinglists.

4.7.5 Version control systems

An import tool in (Open Source) software development are version control systems, which facilitate in integrating code of different developers into one program. Version control systems are usually networked applications, who can merge different parts of source code. To give a somewhat simplified example, two developers are working together on a project which consists of an arbitrary number of files. A typical workflow in a version control system would look like this:[14]

(1) Developer A updates his working copy by downloading the most recent version of the source code from the central repository.

(2) Developer A modifies some files.

(3) In the meantime, developer B is also working on the sourcecode, but in other parts (maybe, but not necessarily in another file).

(4) Developer A finishes his work.

(5) Developer A checks, if there have been made changes to the central repository that might conflict with his modifications. Usually the conflicts arising can be solved automatically. The commonly used version control systems are all able to handle conflicts such as a 'concurrent modification' (for example manipulation of File X by developer A somewhere at the top, and modification by developer B somewhere at the bottom.) This works in most of the cases automatically, saving the respective developers a fair amount of work when it comes to merging the changes. Where the conflict is not obvious, so it can't be resolved automatically, version control systems provide means to present changes made to the repository arranged in a timely manner. So problems can be traced back easily.

Commonly used version control systems are “CVS” (Concurrent Version System), “Bitkeeper” (a proprietary software which is used in today's Linux kernel development) and “subversion”, which is seen as a consecutive CVS program.

Merging of code from different developers might be the most important feature of recent version control systems. Frequently used version control systems also offer the possibility to

14 http://svnbook.red-bean.com/svnbook/ch03s05.html

keep track of all changes made. A developer can - at any time - fetch any version from this system. This makes it easier to track down issues by reviewing changes that have been made to the code. Also, the software provides a *changelog*, a list with all the changes, the name of the person who submitted the changes and a comment on them, which makes it easy to review new code and changes in existing parts. This is especially important in the light of peer-reviewing.

4.7.6 Websites

Websites of projects usually contain pointers to different sorts of information related to a project, common parts of websites facilitating Open Source developers contain pointers to mailinglists, lists with frequently asked questions ("FAQ"), installation instructions, documentation, names of developers and contact information. Also download locations, featurelists, screenshots and everything related to the project can be found there. There are also a number of sites which provide webservices for Open Source projects, SourceForge.net[15] and Freshmeat.net[16] being the most popular. SourceForge.net provides thousands of developers with version control systems, a webserver to host the project site on and archived mailinglists. Freshmeat.net also offers these kinds of services, but also has the character of a portal between developers and users. Freshmeat.net offers users a database to search in for an application, but has a much less sophisticated version control system.

4.7.7 Wikis

Wikis are interactive websites which contain information on different issues of the aspects dealt within the projects, such as documentation, Howtos, amongst others. A wiki is web-based collaboration software with a very open character. The idea behind a wiki is is to enable users to share information and experience. Users of a wiki are enabled to modify the content of the wiki. So if a user misses some piece of information in the wiki, he or she can add the information to the wiki. In this manner, the content of a wiki is on the one hand very much dependent on the amount of active users, but can be a very good means to generate comprehensive documentation. Wikis spread the work that has to be done to administrate the content of a website, making it a dynamically changing source of information. An example for such a Wiki would be the website of the *Direct Rendering Infrastructure*[17], which provides hardware accelerated graphics for all sorts of applications, such as games. This website is based on a wiki, which enables the users to add useful information to the site, and make it a collaboration effort, rather than a site maintained by a single person, with all the constraints taken into account. Wikis are a phenomenon that emerged from the principles of Open Source software development and the idea of free and shared information.[18]

15 http://sourceforge.net
16 http://freshmeat.net
17 http://dri.sourceforge.net/cgi-bin/moin.cgi/
18 For an example with rapidly growing popularity, see the online encyclopedia WikiPedia, http://wikipedia.org

5. Analysis: Comparing the two approaches

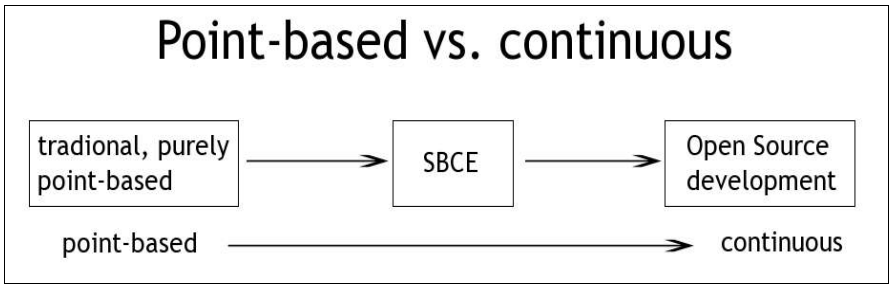
5.1 Introduction

In this chapter, I will first compare prerequisites and assumptions both, SBCE and Open Source development are based on. Then I will compare different keypoints of both approaches to equivalent or matching issues in the other approach. The issues dealt with are mapping of the design space, conceptual robustness, integrating of efforts, conflict resolution, concurrent design, collaboration and feasibility of the solution chosen.

5.2 Prerequisites and assumptions

The first striking difference in both approaches is surely that SBCE assumes a more or less closed system, on the one hand with respect to who's part of the development team, but also with respect to the borders of the development process. SBCE assumes that at a certain point, a design process is finished, usually with a product of acceptable quality and features.

In Open Source development, there is not really an end to a development process. Open Source development is much more a continuous process of improving the product, with no hard finish.



Another very important point is that SBCE deals with material products, while Open Source development deals with ideas and code, which can be copied without costs. This has major impact on the costs of integration, and probably on the suitability of SBCE for analysing Open Source development.

But there are also a lot of things these two have in common:

- Both are team-based approaches.
- Both assume that a robust, modular design is the key to a good product.
- Both work simultaneously with more than one developing team.
- Both approaches try to reduce the necessity of rolling back a decision earlier made.

The most striking thing in contrast to SBCE might be, that Open Source software development does not seem to know of a clear path of handling bad designs. While there are techniques within SBCE about what to do if a certain design or modules does not pass the feasibility check, there does not seem to exist clear ways of a) handling unexpected failure in modules and b) keeping the time that's lost due to unexpected problems at a low level. In both cases, the consequences of failure are kept small by design decisions, such as modularity issues.

5.2.1 Goals in both approaches

The goals of both approaches are quite different with respect to the results. While SBCE aims at short development times, time consumingness in Open Source development plays a far less important role. This gets reflected in the fact, that deadlines are not a common issue in most

Open Source projects. However, both approaches aim at good conceptual robustness, so there certainly is common ground.

5.3 Design spaces

Design spaces in Open Source development are generally dictated by the principles of good engineering. An analysis of the design space is somewhat difficult, since properties of the final product in Open Source development are developed gradually. In fact, building the solution to a small problem and then integrating this solution with other parts is the way to go in Open Source development. Mapping of the design space with respect to dividing the problem into parts of a solution is done much more on an ad-hoc basis, then it would be in SBCE environments. Open Source development handles a reactive approach to mapping of design spaces, communication channels are opened as needed, the same applies to sharing of other resources. The sources of information used in Open Source development, however, is roughly the same as in SBCE. Analysis of the problem, results from testing and information from outside (which is according to the open character not really outside) play the central role.

5.4 Conceptual robustness

Conceptual robustness is the keypoint to preventing unforeseeable problems with the product. In SBCE, the main advantage of conceptual robustness are in the first place that it reduces complexity, which decreases necessary communication between departments. This is accomplished by modularizing the design. Similar advantages apply to Open Source development. In the first place, complexity is reduced by splitting up the problem into different modules. This also makes it possible to work simultaneously on more than one module.

However, the interfaces between these modules have to be well-defined. Interfaces between these modules are defined within the developing group, usually via discussions on mailinglists. Module boundaries are often defined by a generic character of a certain task. That means, that if a certain task is likely to be necessary in more than one context, this task will be put in a generic module, and in this way made reusable for every context. This way, maintainability is increased. Improved reusability is also an important advantage.

As a conclusion, Open Source development and SBCE roughly handle the same approach to conceptual robustness. Both make use of modular designs as a means to a) make it possible to work simultaneously and b) provide means to improve maintainability and reusability of the parts. Strong interfaces are the key to conceptual robustness in both approaches.

5.5 Integration of the respective parts

In SBCE there are certain points defined, which should serve as integration points. A robust concept developed in the beginning prevents most infeasible designs. A higher integration point granularity increases flexibility of the design. SBCE assumes, that more integration points in comparisons with traditional point-based approaches improve both, product quality and the use development resources.

Open Source development handles a much more incremental approach to this. Tools like Version Control Systems accomplish access for virtually everyone to the most recent development version. Developers can decide by themselves when to integrate other's changes in their work. The workflows connected to Version Control Systems and the principles and organizational structure provide a robust “main version” that everyone can integrate his changes into. Since it is in the great majority of all cases “not done” to break existing work, integration of different parts will go seaminglessly.

5.6 Conflict resolution

Conflict resolution in Open Source communities seems much more easy than in theory. In the great majority of all cases there are pretty clear guidelines, either written and published as documentation of commonly used standards, dictated by principles of good engineering, common sense or just the solution that a respected co-developer suggests. Everyone accepts these guidelines, and not at least, people work with the same goals in mind. Open Source communities are – with respect to the issues dealt with in these communities – generally homogeneous ones. The approaches taken in the light of SBCE also aim at resolving conflicts in a cooperative manner.

Domain level conflicts do not arise very often, because in most cases, principles of good engineering provide the necessary paradigms. Since in Open Source development communities, technical elegance is one of the most important criteria, control level conflicts do not play an important role. Domain level conflicts are usually solved by consensus of the parties related. Control level conflict do not arise that often, at least not in the form of messy problems. Control level conflicts can in most cases be brought down to issues dealing with the principles of good engineering – with modularity and the like being the keypoints. In that way, there is not too much room for discussion. Then, these conflicts usually have to do with constraints other than technical elegance, for example costs and the risk of not finishing the product by a certain deadline. As already mentioned, costs and deadlines do not play an important role in Open Source development, so it comes down to that there is a superordinate goal, which is creating the best possible software within the principles of good engineering.

Klein also describes two ways of dealing with a conflict, development-time resolution and run-time resolution. Open Source developers generally try to resolve conflicts at development time. In technical context, there is often the choice between a solution that prevents problems in the first place by creating code, that is very strict with respect to what it actually does. Preventing situations that can have unclear or unforeseeable consequences is generally preferred above a solution that “works in a way”, but where too many variables are left open and as a result of that, problems have to be fixed afterwards.

Formalisms for conflict resolution used in SBCE environments are pretty clear in the context of Open Source development. In general, problematical situations should be prevented. Situations, where run-time conflict resolution would be needed are a sign of bad design or other structural problems. One could even state that solving conflicts at run-time, thus creating 'knowledge-poor' solutions, is in most cases not acceptable by criteria of good engineering.

5.7 Concurrent design

One of the principles of Open Source development are short release cycles, in order to prevent gaps between parts that have to fit together at some point. If two teams are working on adjacent parts of a product (or, in this case software), the more often these teams integrate their efforts with each other, the better these parts will fit in the end. This approach also offers the advantage, that a robust concept lets problems with the design come to light in a early stage

of the development process. This makes it easier to solve these issues, than if they would emerge at a point where a lot of work would have to be redone to provide a solution for the problems the integration brings. There is also the advantage, that *ugly workarounds* (ugly in the sense of conceptually not robust) can be prevented more easily. Also, it appears that these less feasible or conceptually less robust and flexible designs are sorted out at an early stage, because feedback from all parties related to the solution is dealt with throughout the whole development process.

While SBCE assumes that working out more approaches simultaneously will lead to short development times due to the fact that the time invested in 'bad' parts of the solution is reduced, there is usually no programmatical simultaneous development of different solutions in Open Source development. Different modules are developed in a simultaneous manner, although there may be independent approaches to it. Debugging is also easily parallelizable, according to Linus's Law. Although debugging requires feedback from some coordinating developer, it does not require significant communication effort between debuggers (which are generally either developers or more skilled users). Thus debugging does not suffer from an exponential grow of complexity a system where all peers communicating with each other would have to suffer from.

The *"release early, release often"* paradigm, however, is typical to Open Source. While it is for obvious reasons virtually impossible to handle this paradigm in a typical SBCE environment, for example car manufacturing, the merits of information technology make it possible to increase the "release granularity" significantly.

SBCE assumes, that integration of the parts depends on strong standards and not breaking decisions once made. This is also the case in Open Source development. Although the constraints given by the nature of the environment (physical products vs. immaterial products like software and information) offers possibilities, typical SBCE environment just cannot take advantage of.

5.8 Collaboration

Collaboration in Open Source development seems, at first sight much less structured, than in typical SBCE environments. This is explainable by the fact, that Open Source communities do have a much more voluntary character in the first place, and in the second place, they are much more flexible in the sense of ad-hoc networks. No structure is imposed by the hard goals, no one has absolute power.

This collaboration demands a good deal of discipline. The source of this discipline might have its roots in the motivational factors of most developers.[1] Collaboration in Open Source communities is much more a question of ad-hoc or on demand linking of the peers. In a typical Open Source environment, issues are not necessarily kept within the closer community rather than having a look at who would be the best person to answer a certain question, and then ask it. The barrier between the respective communities is kept low, social interaction is usually informal, but respectful although unnecessary politeness is not in its place. Developers argue, that time is a worthy resource, and that politeness is good, but should not create unnecessary overhead. Getting to the point is appreciated, wasting other developers' time with redundant information is not.

5.9 Establishing feasibility

SBCE assumes that decisions once made should have to be respected, breaking modules or existing interfaces is simply not done and wouldn't earn the developer much respect (apart from the fact, that addition or changes not respecting the existing structure do not have a good

1 See Appendix A for a more detailed explanation of motivational issues

chance to get integrated. The quality of the technical solution is extremely important. Compromises with respect to code quality are not usual. Keeping interfaces and decisions once made absolutely copes with the approach SBCE takes. While development in SBCE takes place in departments, that integrate their work at some point, this is not the case with Open Source development. The parts are put together in the earliest possible stage. New additions added will make problems make come to light immediately, or at least at the next release date (which should not be far away).

SBCE tries to maintain feasibility by keeping the intersections between development departments flexible, whereas Open Source development maintains feasibility.

Structural issues are another issue. Before a project is started, or major new features or parts are added, structural issues are discussed. These structural issues are discussed until a solution that is robust and flexible enough has arisen. Good engineering gives the basic ideas for the structure and layout. The parts will then be filled in and incrementally merged with the main project.

This stands in contrast to the SBCE approach, where it seems to be just the other way round. The differences between these approaches might be based in the fact that integrating parts of non-materials can only be integrated in a theoretical environment, which would not add significant advantage, since these issues already should have been dealt with in the initial design.

Software development naturally offers the advantage, that developers can try out things more easily. To give an example, if developers of a certain project would like to introduce a structural change in existing code, they could implement that, and release a test version. That would perfectly match the “release early, release often” paradigm. Users, who are interested in these newest (“bleeding edge”) code, could try it out. Developers will get feedback from these users, see if the changes are worthwhile and use them or not. The important thing about this situation is, that it would cost only little effort to change things back.

In typical SBCE environments, the developers would have to wait a rather long time, until they would get feedback. In the meantime, a lot of work will be done, which – in the worst case – would be useless.

Naturally, this worst case scenario is also possible in Open Source development, though much more unlikely. It would be safe to state, that if a developer knows about the possible consequences of a decision, the decision will be discussed by the community, but also, he would take measures, which make it easy to roll the changes back.

6. Conclusions

6.1 Introduction

In this chapter, I will outline the results of the analysis and put them in a more general context. I will give an answer to the research question and have a critical look at both approaches and the combination of them. The chapter finishes with a reflection about my research and some ideas I got while working on it.

6.2 General

While SBCE introduces shorter cycles of integration of parts, Open Source development is even more radical in contrast to traditional point based approaches. Short release cycles and short integration cycles make it easier to deal with problems in an early stage.

Simultaneous design of different solutions is usually not applied in Open Source development, but it does also not seem to be an issue since the approach taken in Open Source projects is to trust the design and lean on previously defined interfaces. An important issue in that respect is quality assurance, since the standards and previously taken decisions have to be of a certain quality level so the standards and already built interfaces new parts lean on are unlikely to change because they already offer the "best possible solution". Reuse is a good thing in the Open Source world, reinventing the wheel will in most cases offer little or no advantages, since the quality of standards, protocols and interfaces is in most cases very high. They have been developed, improved and adopted in the course of several years by a lot of skilled developers and reached a certain level of maturity.[1] It is rather unlikely, that a certain problem has not already been dealt with and that there is a real structural problem with these standards.

SBCE, in contrast to Open Source development can't apply short release cycles as in Open Source development. A product, for example a car in the case of Toyota, simply cannot be shipped when it still contains "bugs". Recalling a series of cars is very expensive and usually brings damage to the reputation of the manufacturer. In Software development, it is much more usual to ship a product still containing bugs, even in closed source commercial development (although the reasons might be different to the ones in Open Source development, namely tension between marketing and software development departments, amongst others).[2]

The shorter the development cycle, the less probable unforeseeable problems are. Frequent integration and testing of parts may in the first place seem time-consuming, but apparently it does lead to a robust design because feedback from all sides can be integrated in the solution at an early stage of the design process. Taking away the border between user / customer and developer and closing the feedback cycle works pretty well.

6.3 What can SBCE contribute to freedesktop.org's goals?

Actually, it seems that the strong parts of SBCE, the parts that make the difference with point based approaches already apply to Open Source development. Building on strong standards, and maintaining conceptual robustness stands central to Open Source development.

On the other hand, Open Source development does not make use of concurrent designing. Usually, parts are designed only one time. The necessity to step back does not seem to exist on a regular basis. Set-based concurrent engineering cannot contribute very much to

1 http://www.cyber.com.au/users/conz/shoulders.html
2 http://blogs.msdn.com/David_Gristwood/archive/2004/06/24/164849.aspx

freedesktop.org's goals, except stating, that under the circumstances of freedesktop.org's environment, typical constraints from SBCE environment do not play a role. For example release schedules are much more flexible, hard deadlines do not exist and the goals of all related parties usually are roughly the same.

A short answer to the research question is not easy to give. The main contribution that SBCE could provide for freedesktop.org's work is that they – in the light of SBCE – are actually doing a very good job. However, the different constraints of both approaches make it hard to recommend specific things, other than that they should make use of the non-existing constraints.

Posing the question the other way round reveals that if SBCE should be applied to Open Source development, the theory does not offer detailed solutions for specific problems. The general direction of SBCE, assuming that working together based on strong standards helps to make the work more efficient is evident.

6.4 What aspects important to freedesktop.org are not covered by SBCE?

Set-based concurrent engineering does not deal with technical issues in detail. In Open Source development, technical issues play an important role, and they are even applied to non-technical issues, such as the way communication takes place. Also, freedesktop.org, being an organization that deals with information technology has itself the capacity to improve the ways it is working. In the light of SBCE freedesktop.org does a pretty good job. A better question would probably be “What issues should be covered by SBCE, in order to be able to apply it to (Open Source) software development?”. I would advice that as a starting point for further research.

6.5 Reflections

Judging afterwards, set-based concurrent engineering does not fit freedesktop.org's work in all detail. That said, now I know why. Constraints such as hard deadlines, necessary compromises to maintain them and a mainly commercial approach might be the reason for that. In that light and regarding the quality of Open Source products, it seems that commercial issues and quality will not always play together nicely. On the other hand, regarding a lot of big vendors, like IBM, HP and others selling increasingly more Open Source products, it seems that it pays off. Commercial goals and Open Source software do work together very well. “Merely” the marketing model has to be changed. Selling software becomes less important than offering a good product, including service. This approach does also fit the requirements of the customer in a better way. The customer gains independence by not having to trust one vendor for fixing his software. The customer gains the choice to choose whoever he wants to deliver the service, a lock-in by a certain software vendor does not exist. Open Source software service providers are forced to deliver good quality products, and if they do not, the customer just chooses a different service partner, without having to overhaul the complete technical infrastructure.

The Open Source development model offers an interesting approach to engineering. A system set up by technical people for technical people, based on consensus, principles of sharing and open channels of information and communication and abandoning traditional hierarchical structures does actually work out quite good.

Me myself, as having become part of the Open Source community, appreciate the way people are dealing with problems, setting quality in the first place. And then, it's a non-democratic system, without a dominating hierarchy which in my opinion does a good job. The existence and

increasing success and adoption of Open Source products proves that there is more than commercial closed source software, and that the idealistic hacker world propagated by Richard Stallman has gained momentum.

7. Bibliography

7.1 Books and articles

- D.K. Sobek, A.C. Ward (1996), Principles from Toyota's set-based concurrent engineering process, *Proceedings of the 1996 ASME Design Engineering Technical Conferences*, University of Michigan, Michigan
- D. K. Sobek, II, Allen C. Ward, Jeffrey K. Liker (1999), Toyota's principles of set-based concurrent engineering, *Sloan Management Review, Vol. 40, winter 1999, pp. 67-83*
- Mark Klein, (1995), iDCSS: Integrating Workflow, Conflict and Rationale-Based Concurrent Engineering Coordination Technologies. *Journal of Concurrent Engineering Research and Applications*. Volume 3, Number 1, January 1995.
- Mark Klein, (1991), Supporting Conflict Resolution in Cooperative Design Systems. *IEEE Transactions on Systems, Man and Cybernetics*. Special Issue on Distributed Artificial Intelligence. Volume 21, Number 6, December 1991.
- Mark Klein, (1990), Conflict Resolution in Cooperative Design, *The International Journal For Artificial Intelligence in Engineering*. Volume 4, Number 4, Pages 168-180, 1990.
- Bahler, D., Dupont, C., Bowen, J. (1993), *Mediating Conflict in Concurrent Engineering with a Protocol Based on Utility*, North Carolina State University
- Mark Klein, (2003), Negotiation Algorithms for Collaborative Design Settings, In the proceedings of *The 10th ISPE International Conference on Concurrent Engineering Research and Applications (CERA-03)*, Madeira Island, Portugal
- Zolingen, S.J. van, C.A. Klaassen (2002), Characteristics of a Delphi study, in *Technological Forecasting and social change*
- Castells, M., (2001), *The Internet Galaxy: Reflection on the Internet, Business, and Society*, Oxford University Press, New York
- Torvalds, L., Diamond, D., (2003), *Just for fun*, *The Story of an Accidental Revolutionary*, HarperCollins Publishers, New York
- Collins-Sussman, B., Fitzpatrick, B. W., Pilato, C.M. (2002), *Version Control with Subversion,* http://svnbook.red-bean.com/
- Vennix, J., Methodologie 1 (2001), *Onderzoeks- en interventiemethodologie: een beknopte inleiding*, Nijmegen School of Management, Nijmegen
- Verschuren, P. and H. Doorewaard (2001), *Designing a Research project*, Lemma, Utrecht

7.2 Websites

- Website of the freedesktop.org project, http://freedesktop.org
- *Official website of the KDE project*, http://kde.org
- *Official website of the GNOME project*, http://gnome.org
- *WikiPedia, the free encyclopaedia*, http://en.wikipedia.org
- *First monday, Journal of the internet, Open Source Intelligence*, http://www.firstmonday.dk/issues/issue7_6/stalder/#s1

- *Christopher Browne's Web Pages, Linux and Decentralized Development*, http://cbbrowne.com/info/lsf.html
- *Agorics Inc. - Computation and Economic Order*, http://www.agorics.com/Library/agoricpapers/aos/aos.3.html
- Ian I. Mitroff & Murray Turoff, *The Delphi Method: Techniques and Applications*, http://www.is.njit.edu/pubs/delphibook/ch2b.html
- Eric Steven Raymond, *The Cathedral and the Bazaar*, http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar
- *Shoulders of Giants*, http://www.cyber.com.au/users/conz/shoulders.html
- David Gristwood, *21 Rules of Thumb – How Microsoft develops its Software*, http://blogs.msdn.com/David_Gristwood/archive/2004/06/24/164849.aspx

Appendix A: Free software, Open Source software: What is it all about?

8.1 The hacker culture

The hacker culture emerged from the 'real programmers'[1], who were, in the years after World War II and about 1970 the operator and developers of early computing, also referred to as batch (scientific) computing. However, the roots of today's hacker culture can be dated back to 1961, when the MIT acquired their first computer, that formed the basis for early research on artificial intelligence. As a side note, Richard Stallman worked for several years at the MIT Artificial Intelligence lab, at the beginning of his career and end of his studies. The hacker culture is seen as the cradle of Open Source software. In contrast to the image of the hacker that is used in current media, a hacker is a passionate computer programmer, rather than a person illegally penetrating into systems in order to crack codes and committing other 'cyber-crimes'.[2]

8.2 What is Open Source or Free Software?

Illustration 2 Open Source (tm)

In general, *Open Source Software* are programs or parts of programs, that are licensed under certain terms. The *Open Source Definition* is primarily based on the *Debian Free Software Guidelines*, originally written by *Bruce Perens*. The *Open Source Definition* emerged from the need to have the *status quo* in the *Free Software Movement* explicitly articulated.

Software that is *Open Source* has to be released under a license, which complies with the following criteria:[3]

1. The license must not restrict any party from selling or giving the software away.
2. The software must include source code, in order to make modification easy.
3. The license must allow modifications and derived work.
4. Although modifications are explicitly encouraged, modifications have to be made in a way, that makes sufficiently clear who wrote a certain piece of code.
5. The license may not contain restrictions for certain persons or groups.
6. The license must not restrict anyone from using the program in a specific context, for example in a commercial way.
7. The license applied to the software must apply to all to whom the software is redistributed.
8. The license must not be specific to a particular software distribution. The software may be taken out of its original appliance and used for something different, with the same licensing terms.

Illustration 3: Bruce Perens

1 http://www.catb.org/~esr/writings/cathedral-bazaar/hacker-history/
2 Castells, 2003, p. 41-42
3 *The Open Source Definition*, http://opensource.org/docs/definition.php

9. The license must not restrict other software. For instance, the use of a program can't be restricted to a combination with certain other programs.
10. The license must not predicate the use of any individual technology, of style of interface.

The rationale behind this license is that the software should be *free*. *Free*, in the *Free* or *Open Source Software* context does not mean "*free of charge*", but freedom to use, freedom to modify and freedom to (re)distribute the software. The difference between *Free Software and s*oftware that's just *free-of-charge* is in the *Open Source Movement* often referred to *"free as in beer"* in contrast to *"free as in freedom"*, with the latter describing the idea behind *Free Software*. This came forth from the opinion of the 'hacker culture' that there should be no restrictions on how people use a program, what they do with certain programs, and not at least that people who use the software are allowed - and given the means - to fix bugs or other problems they have with the software.[4] Free software is about sharing, according to Richard M. Stallman[5], father of the GNU Operating System and the *Free Software Foundation*[6].

Illustration 4 Richard M. Stallman

The term *Open Source Software* was more or less a compromise to reflect some of the properties of free software in a better way. The reason to name it Open Source Software is, that the word *free* is very often associated with *free of charge*. The Open Source Software foundation wanted to stress the possibility to change things and to actually work on the software in an active way, and deal with it in a more technical way.

In general *Open Source Software* means the same as *Free Software*, although the term *Open Source* does stress more technical properties, while *Free Software* is more used in ethical and non-technical discussions. In short: the term *Open Source* stresses the principles of open standards, shared source code and collaborative development.

The Open Source model has emerged from the early UNIX development model, but has undergone lots of changes. In this paper, I will use the term Open Source development model for the development model that has become common for the Linux community.

8.3 Why are people working on Open Source software?

In this paragraph, I provide insight in why Open Source developers are working on Open Source software, rather than keeping the code for themselves.

There are 2 main reasons for working on Open Source software:

1. Open Source software has already solved lots of technical problems. If people need or want to develop a program with a certain functionality, they can simply fall back on Open Source software. However, in some cases, there is no Open Source software which does exactly what the 'customers' want it to do. In this case, they can just build the functionality on top of existing applications or libraries. Duplicate work can be very often prevented. Using Open Source software for a project can simply save the developer lots of time.
2. Personal affection. There are a lot of hobbyists who are working on Open Source projects in their free time. The motivation in these cases can be explained with the fact that taking part in a community can be a motivation driver. Then, the fact that writing programs as a hobby is a creative activity, especially, where it is likely that the code written will eventually be used,

4 See also: "Free as in Freedom", Richard M. Stallman, http://www.faifzilla.org/ch01.html
5 http://www.findlink.dk/stallman/stallman.htm#The_philosophy_of_Stallman
6 The GNU project, http://www.gnu.org/

and not drown on a pile of 'once written, once used' increases motivation as this acts like a reward in the community. The Open Source model keeps its users and developers constantly stimulated and rewarded. Stimulation is accomplished by the prospect of having 'an ego-satisfying piece of action'[7], reward is provided by constant improvement of the work being done and the obvious experience of the developers, that their work is actually being used, and thus appreciated. The community plays an important role here, since it provides both, positive and critical feedback. Generally, we can state, that the feedback loops are very tight, and the barrier to give feedback or propose or contribute improvements is kept on a low level.

Linus Torvalds has a good point on motivation to work on Open Source software, too:

> *"It's been well established that folks do their best work, when they are driven by passion. When they are having fun. [...] The Open Source model gives people the opportunity to live their passion, to have fun. And to work with the world's best programmers, not the few who happen to be employed at their company. Open Source developers strive to earn the esteem of their peers. That's got to be highly motivating."*[8]

7 http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html

8 Torvalds, 2003

Appendix B: Interviews

9.1 Interview partners

A lot of the data used in this thesis comes from interviews held with developers from different freedesktop.org communities. The interview partners are Ronald Bultje and Simon Edwards. The roles these people play within freedesktop.org will be described in short here.

9.1.1 Ronald Bultje

Ronald Bultje is a multimedia application developer, who is working on the freedesktop.org project gstreamer[1]. gstreamer is an Open Source multimedia framework for UNIX (and Linux) operating systems. Ronald focuses on desktop multimedia integration and deploying gstreamer for the GNOME desktop environment. He also does work on different plugins, which enable gstreamer to use formats like the popular MPEG video technology.

Ronald does this work for personal reasons, he wants to be able to use free software for multimedia tasks on his computers. These tasks include, but are not limited to audio/video capture, playback and editing.

9.1.2 Simon Edwards

Simon is a developer closely related to the KDE project. He has developed a firewall application that runs under the KDE desktop environment, called guarddog[2]. Recently, Simon is working on *guidance*, which is a set of applications and utilities for managing hardware and other operating system specific tasks, such as user management. Guidance aims to fill a gap between KDE as a platform independent system and various Linux distributions.

Simon also works for the KDE project as maintainer for PyKDE, a set of programming libraries that provide KDE with support for the Python programming language. In this way, he takes part in different, all somehow freedesktop.org-related communities.

Simon also works for freedesktop.org out of personal motivation. He wants to see what he can do with the tools provided by recent development in the hardware and software domain. He is also eager to improve end-users productivity by providing them with better tools. Simon wants to create better user interfaces. The bottom line is that he loves building software.

9.2 Interviewguides

The questions that were dealt with in the respective interviews were all derived from this interviewguide. However, some questions have been discussed in more detail than it is shown below. In this way, the interviewguides are not a complete reflection of the interviews and of the information gathered, but should give a rough idea of what sources the data used in this thesis is based on.

Personal

1.What is and has been your contribution to freedesktop.org's goals?

2.What is your motivation to work on freedesktop.org's goals?

3.Where in freedesktop.org's structure would you consider yourself to be situated?

1 http://gstreamer.freedesktop.org/
2 http://www.simonzone.com/software/guarddog/

Structure of freedesktop.org

4.How would you, in short, describe the organizational structure of freedesktop.org's development community with respect to

1. hierarchies?
2. information channels?

5.How are decisions about commonly used interfaces between parts of applications (and also general issues) made?

6.Who's the final decision maker in conflict situations?

7.What is the course of handling conflicts? (Assuming, that a certain conflict is not easily resolvable, what steps are followed, in what order and why?)

8.What are the most important communication channels used in your development work?

9.What are the most important collaboration tools used in your work?

Design decisions

10. Which criteria make a design “good”, which characteristics make it “bad” of less acceptable?

11. What role does modularization play in your development work?

12. How are module boundaries defined, what are they based on?

13. Under what circumstances would it be acceptable to deny the use of a certain (set of) interfaces?

14. How are “black box solutions” handled, that provide a certain feature, but internally don't meet the requirements, like proper coding, documentation and such?

15. In what way do agreements on standards, interfaces and design issues contribute to a certain design? What would be the consequence, if these agreements are not followed?

16. How is complexity reduced?

17. What's done to assure a certain level of quality, how are these quality criteria defined?