Master's Thesis

Advances in Domain Centered Software Design and Development on the Example of a Budgeting Software

Martin Brugger, Bakk.rer.soc.oec.

Institute for Information Systems and Computer Media (IICM),
Graz University of Technology

Supervisor: Assoc. Prof. Dr. Andreas Holzinger, PhD, MSc, MPh, BEng, CEng, DipED, MBCS

Graz, May 2011

This page intentionally left blank

Masterarbeit

(Diese Arbeit ist in englischer Sprache verfasst)

Fortschritte in der Entwicklung domänenspezifischer Software am Beispiel einer Kalkulations Software

Martin Brugger, Bakk.rer.soc.oec.

Institut für Informationssysteme und Computer Medien (IICM),
Technische Universität Graz

Betreuer: Univ.-Doz. Ing. Mag. Mag. Dr. Andreas Holzinger

Graz, Mai 2011

This page intentionally left blank

Abstract

Development of commercial software requires a high quality technical implementation, as well as the consideration of legal and economical conditions. Therefore, cross-disciplinary skills need to be acquired, to create a holistic approach to software development.

With the vast amount of information freely available on the internet, tremendous potentials, but also legal risks emerge by using open source software in commercial software development. Developers have to consider copyright laws and licensing carefully before reusing extraneous open source code.

Software development methodologies are the basis for high quality and controlled software development, with a big influence on development success. For this master's thesis, an agile development approach was chosen, to accommodate the uncertain user requirements at project start and to prepare for future development projects.

Another focus of this work is the integration of usability engineering into the development process, as another key factor for success. As a result of the development process, a software for costing and planning international film projects was produced in close cooperation with domain experts, which currently is distributed internationally.

Keywords

software development methodologies, agile software development, software testing, software licensing, usability engineering

ÖSTAT classification

1108 45%, 1140 40%, 1153 15%

ACM classification

D.2.5, D.2.7, D.2.9, H.5.2, K.5.1, K.4.1

This page intentionally left blank

Kurzfassung

Die kommerzielle Entwicklung von Software erfordert nicht nur eine qualitativ hochwertige technische Umsetzung, sondern auch die Erfüllung von rechtlichen und wirtschaftlichen Rahmenbedingungen um diese durchführen zu können. Diese Voraussetzungen machen es notwendig, fächerübergreifende Fertigkeiten abseits der Technik zu entwickeln, um alle Aspekte gewissenhaft berücksichtigen zu können.
In der heutigen vernetzten Welt und den fast unendlich verfügbaren Ressourcen im Internet, haben sich ein enormes Potential, aber auch rechtliche Risiken durch die Verwendung von Open Source in kommerzieller Softwareentwicklung ergeben. Es muss allerdings unter Entwicklern noch ein Bewusstsein für Problematiken, die sich durch diese große Verfügbarkeit von urheberrechtlich geschützten Werken ergeben, geschaffen werden.
Softwareentwicklungsprozesse haben maßgeblichen Einfluss auf den Erfolg und ermöglichen es erst, kontrollierte Entwicklung zu betreiben. Für das Projekt im Rahmen dieser Diplomarbeit wurde ein Einstieg in die agile Softwareentwicklung gewählt, um den wechselnden Anforderungen Rechnung zu tragen und auch für zukünftige Projekte konkurrenzfähig Software entwickeln zu können.
Ein weiterer Schwerpunkt ist die Integration von Usability Engineering in den Gesamtprozess. Mit Hilfe dieser Methoden sind eine benutzerzentrierte Entwicklung und die daraus resultierende gute Bedienbarkeit als weitere Erfolgsfaktoren gewährleistet. Als Ergebnis dieses gesamten Prozess ist eine Software zur Kalkulation und Planung von internationalen Filmprojekten entstanden, die in enger Zusammenarbeit mit Experten aus der Filmwirtschaft konzipiert wurde und international vertrieben wird.

Schlüsselwörter

Softwareentwicklungsmethoden, Softwarelizenzierung, Usability Engineering

ÖSTAT Klassifikation

1108 45%, 1140 40%, 1153 15%

ACM Klassifikation

D.2.5, D.2.7, D.2.9, H.5.2, K.5.1, K.4.1

This page intentionally left blank

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 02.05.2011

Martin Brugger

This page intentionally left blank

Acknowledgements

This thesis would not have been possible without my family who always encouraged me to cut my own path. I would like to thank Nina for being back again and supporting me all the years of my studies.

Special thanks also to my colleagues at work for providing me with time to finish my diploma thesis. Thanks to all my friends and especially Olivia for motivating me all the time.

Finally I would like to thank Andreas Holzinger for supervising and guiding my diploma thesis.

Martin Brugger

Graz, May 2011

This page intentionally left blank

Table of Contents

L

1. Introduction and Motivation for Research

Within the university studies of "Software Development and Business Management" basic concepts for developing software but also business management fundamentals are taught. A graduate in this field of study is qualified to manage a software project in terms of a technical point of view and also the non technical project management. A special challenge within creating this diploma thesis was to apply the studied theoretical knowledge on an actual project. Therefore, it was necessary to include basic management functions starting with planning, organizing, building and leading a project team, directing and controlling.
A lot of duties and responsibilities are not taught in courses at university but had to be applied. A high rate of improvisation and self education was necessary.
Even though business management is a central part of the study, this master's thesis does focus on the technical and project management aspects of the accomplished software project.

The aim of the project described in this thesis is a software designed to assist in the planning, costing and controlling of film projects. Film projects require a high level of flexibility and consist of managing a high amount of complex interrelated data. Several layers of film project management like time, resource and cost planning are tightly connected with impact on each other layer. The developed software automates the resolution of these dependencies. In addition, the enrichment of plain information with metadata allows reasoning about the given data revealing additional knowledge about the film production.

The project team was formed of different partners in different domains. Including members of the film school "HFF Konrad Wolf", a german film producer and several film students. The technical implementation was executed by graduates and students of Graz University of Technology.

In context of this development project different agile software development methodologies (SDM) were evaluated and applied in combination with usability engineering. This approach has already been published several times before. In the academic community extreme programming is the main choice of SDMs, although alternatives exist and are used in commercial software development.

As a student with special interest in usability engineering recognizing the need for established software development methodologies, combining both fields of research were an interesting challenge. Also several surrounding conditions like licensing issues and software testing are discussed because of the high relevance to this software project.

2. Theoretical Background and Related Work

This chapter gives a brief overview of the film production process based on Clevé (2005) and related work in the field of software development methodologies (SDMs) with a focus on appropriate methodologies for the current project. Especially SDMs, which allow a focus on user centered design and usability engineering will be presented. Important topics like software testing and licensing also form the basis for this master's thesis, therefore they are introduced in this chapter.

2.1 Film Production Process

In general the film production process is separated into several phases: *development*, *preproduction*, *production* or *principal photography* and *post production*. (Figure 2.1) Each of these phases has a budget as output. Therefore it is relevant for a software to cover the budgeting process of a film project.

- Within development an idea gets transformed into a package ready to be produced. One of the main tasks in development is the acquisition of rights. This package contains the main cast and creative staff like director, a final screenplay and financial backing. Within budgeting the costs of this package are often referred as *above the line* costs.

- Preproduction is the second phase. Important tasks within preproduction are script breakdown, creating shooting schedules, hiring a production team and creating a detailed budget. The budget will be discussed in more detail later on.

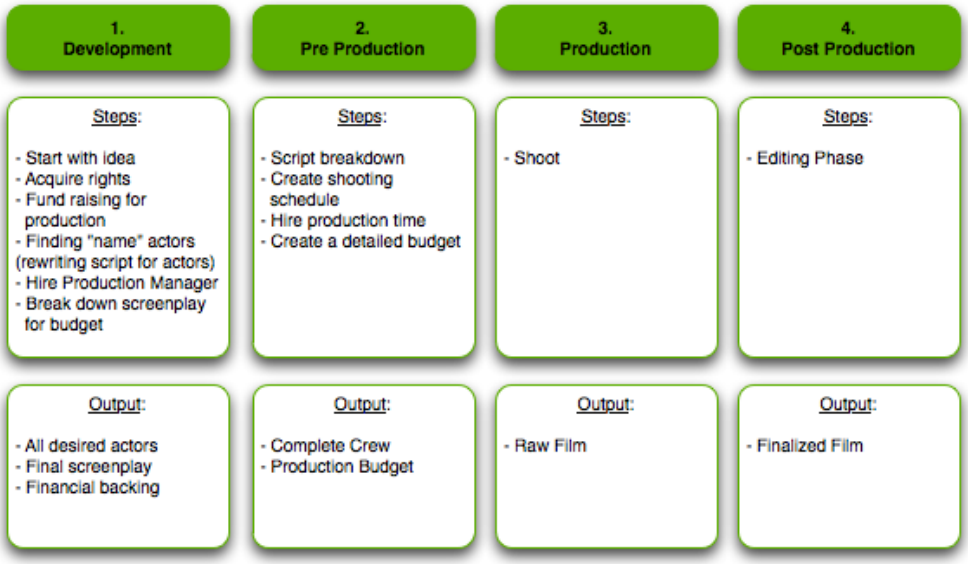


Figure 2.1: The Film Production Process

- The production phase names the actual shoot of the film strictly following the shooting schedule. The *production manager* is responsible for coordinating the whole production process and keeping the project in budget. Many external factors like weather influence the production phase, making it necessary to stay flexible and responsive to changing requirements.

- Postproduction is the final production phase. The result of editing is the finalized distribution ready film. With the increasing amount of special effects and digital postproduction this phase is gaining importance within a film project.

2.1.1 Film Budgeting

Film Projects always involves handling of large budgets and a vast amount of resources. Currently, most of the resource management is handled through the structure of a budget. A lot of decisions within a film project are connected to the budget. Therefore the production manager always needs to know the current state of the budget to be able to stay responsive to changing requirements which occur frequently during the phases of preproduction and actual production.

Cost controlling and up to date accounting therefore is a necessity for all film projects. Experience is a substantial requirement to make good decisions as production manager.
A typical film budget contains all cost information about a film project in a predefined structure, which is defined by the contracting entity of a film project.
American projects like Figure 2.2 typically follow a structure of *above the line* and *below the line* costs. Above the line costs describe the artistic value of a project containing rights, main cast and the director. Below the line costs typically contain the production costs and all overhead.

In Europe, a budget is typically structured to fulfill the requirements of film commissions to be able to acquire film subsidies. This structure is aligned to the chronology of a film production process starting with development ending in post production costs. Figure 2.3 shows the budget structure required by the German Federal Film Board (FFA).

EP Budgeting
Budget Title :

Script Dated :
Budget Draft Dated :
Production # :
Start Date :
Finish Date :
Total Days :
Post Weeks :
Holidays :
Travel Days :

Producer :
Director :
Location :
Prepared By :

Acct#	Category Description	Page	Total
600	STORY/WRITERS	1	0
605	STORY/WRITERS-NEWS/VAR/GS/OTHER	2	0
610	PRODUCERS	2	0
615	PRODUCERS-NEWS/VARIETY/GS	3	0
617	ATL SUPP STAFF-NEWS/VAR/GS/OTHER	4	0
620	DIRECTOR	5	0
625	DIRECTOR-NEWS/VAR/GS/OTHER	5	0
630	CAST	6	0
635	CAST-NEWS/VAR/GS/OTHER	6	0
640	FRINGE COSTS:ABOVE LINE	7	0
650	ABOVE THE LINE TRAVEL	7	0
670	AGENCY COMMISSIONS	7	0
	Total Above-The-Line		**0**
700	EXTRA TALENT	8	0
701	CONTESTANTS/CONTESTS	8	0
705	PRODUCTION STAFF	8	0
999	...	9	0
798	FACILITES FEES	10	0
	Total Below-The-Line Production		**0**
800	EDITING	11	0
801	POST PRODUCTION VIDEO	11	0
999	...	12	0
860	LABORATORY PROCESSING	13	0
870	FRINGE COSTS:EDITORIAL	13	0
	Total Below-The-Line Post		**0**
900	NON PRODUCTION TESTS	14	0
910	ADMINISTRATIVE EXPENSES	14	0
999	...	15	0
920	PUBLICITY	15	0
945	ALL SHOWS	16	0
946	MUSIC	16	0
947	OTHER COSTS	16	0
948	AMORTIZATION ACCOUNT	16	0
	Total Below-The-Line Other		**0**

Figure 2.2: US Sample Budget export from Movie Magic Budgeting

Budgeting (FFA Template)

	Description	Total
1	Vorkosten: (Page 2)	€ 0,00
2	Rechte und Manuskript : (Page 2)	€ 0,00
3a	Gagen Produktionsstab: (Page 2)	€ 0,00
3b	Gagen Regieastab: (Page 3)	€ 0,00
3c	Gagen Ausstattungsstab: (Page 3)	€ 0,00
3d	Gagen Sonstiger Stab: (Page 4)	€ 0,00
3e	Gagen Darsteller: (Page 4)	€ 0,00
3f	Gagen Musiker: (Page 5)	€ 0,00
4a	Atelier Bau: (Page 5)	€ 0,00
4b	Außenbau durch Atelier: (Page 5)	€ 0,00
4c	Atelier Dreh: (Page 5)	€ 0,00
4d	Abbau Atelier und Außenbau: (Page 6)	€ 0,00
5a	Ausstattung und Technik - Genehmigungen und Mieten: (Page 6)	€ 0,00
5b	Bau und Ausstattung: (Page 6)	€ 0,00
5c	Technische Ausrüstung: (Page 7)	€ 0,00
6a	Reise- und Transportkosten - Personen: (Page 7)	€ 0,00
6b	Reise- und Transportkosten - Lasten: (Page 8)	€ 0,00
7	Filmmaterial und Bearbeitung: (Page 8)	€ 0,00
8	Endfertigung: (Page 8)	€ 0,00
9	Versicherungen: (Page 8)	€ 0,00
10	Allgemeine Kosten: (Page 9)	€ 0,00
11	Kostenmindernde Erträge (./.): (Page 9)	€ 0,00
Nettofertigungskosten:		**= € 0,00**
Handlungskosten % v A:		+ € 0,00
Überschreitungsreserve % v A:		+ € 0,00
Zwischensumme:		= **€ 0,00**
Finanzierungskosten (Anlage) :		+ € 0,00
Treuhandgebühren:		+ € 0,00
Completion Bond Kosten :		+ € 0,00
Total Production Costs:		**= € 0,00**

Figure 2.3: FFA Sample Budget Export from LineProducer

Budget types

During a film project different types of budgets are created. In the development phase a rough estimate is made, which is detailed during preproduction. The budget estimation is also called preliminary budget and is built upon price lists, rate books, union rates and experience of the production manager. Most of the costs are subject to negotiation. The detailed budget is based on a finalized shooting schedule and is the basis for all contract negotiations. It is signed of by the production manager and sometimes by the producer and director.

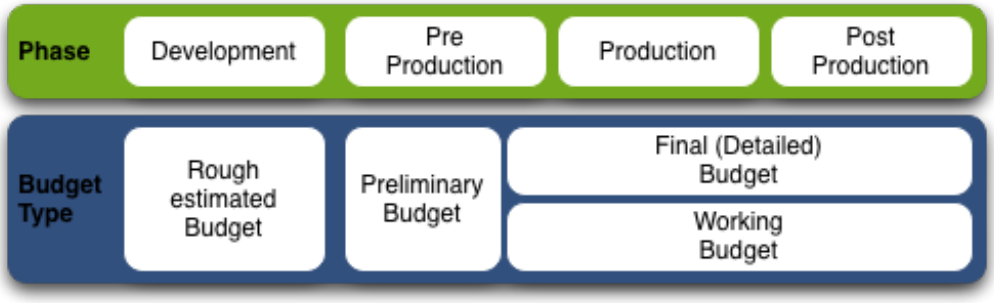


Figure 2.4: Different Budget Types within the Film Production Process

Until the end of the production phase, the working budget is changing a lot to represent the actual progress and cost changes of the project. It is the responsibility of the production manager to balance additional expenses with possible cost savings in other areas to keep the project in budget. Therefore a quick comparison of planned budget and working budget is necessary and a software budgeting solution nowadays is obligatory.

Cash Flow Planning

Another important task related to budgeting is cash flow planning. Cash flow planning is used to plan expenses and income, related to time. Therefore it is an important tool for controlling and forward-looking planning. There is currently no standardized way of cash flow planning. It is recommended to create a cash flow plan on a weekly basis to meet the actual cash flow of wage payments. All cost positions are additionally enriched with time information to allow the planning of a

future balance for the project. This allows an early identification of possible bottlenecks of liquidity, which is essential to every commercial project.

2.1.2 Current Software Products

There are currently software products available which claim to solve the problem of complex film production process. In the following section a selection of these software products will be introduced. All of the solutions are established in their distinct market segments.

- **Sesam**

  Sesam is a german film budgeting software developed by SESAM Software GmbH[1]. Due to the fact of it's long time development starting in 1996 the software is dialog based with all its usability consequences. The software features detailed planning of wages in a fixed cost structure. This fixed cost structure is perfectly suitable for german productions and its main target group of german television and cinema productions. From a technical point of view the software is written in a legacy framework with a Windows only availability.

- **Movie Magic**

  The software named Movie Magic[2] is targeted at American and international film productions. It is Java based and available for the Windows and Mac platforms. From a usability perspective the software features extensive keyboard navigation with a consistent user interface.

- **Microsoft Excel**

  Although the requirements of the target group are quite complex, a significant amount of productions still is planned using Microsoft Excel. The overwhelming flexibility of the software is also its major weakness. Handling several thousand positions with formulas result in copy and paste errors, wrong formulas and a lack of traceability of results.

[1] `http://sesamsoft.de/ueber_uns.htm`
[2] `http://www.entertainmentpartners.com/Content/Products/Budgeting.aspx`

2.2 Software Development Methodologies

This section covers common software development methodologies. Starting with one of the first established development process, the waterfall model, several SDMs are presented up to modern agile software development methodologies used in the current master's thesis development project.

2.2.1 Waterfall Model

The essential steps in software development regardless of complexity and size are described as analysis and coding. (Royce, 1987)
The limitation of this minimal development model is growing software complexity. Computer programs developed in these two steps can only be of limited complexity and mostly suitable for internal use only. A more sophisticated SDM for larger systems is the already mentioned Waterfall model which defines several consecutive steps leading to a planned destination.
Each step shown in Figure 2.5 is the base for further development and no iterations are planned, thus if a later step fails, the whole project has to start from the beginning.

For the current project this methodology was not suitable at all as the requirements developed throughout the whole development of the project. This is not a speciality of the current project. Rajlich (2006) mentions this requirements volatility as serious a problem for even large companies like Microsoft. The waterfall model therefore did not solve the general problems of software development.

2.2.2 Spiral Model

The spiral model is based on refined versions of the waterfall model with a more robust foundation for software development. A key feature of the Spiral Model is risk estimation. Within the spiral model exist two dimensions: The radial dimension is an estimation of the cost originated, the angular dimension the project progress within the current cycle. (Figure 2.6)

The spiral model is executed in several cycles. Each cycle consists of four phases:

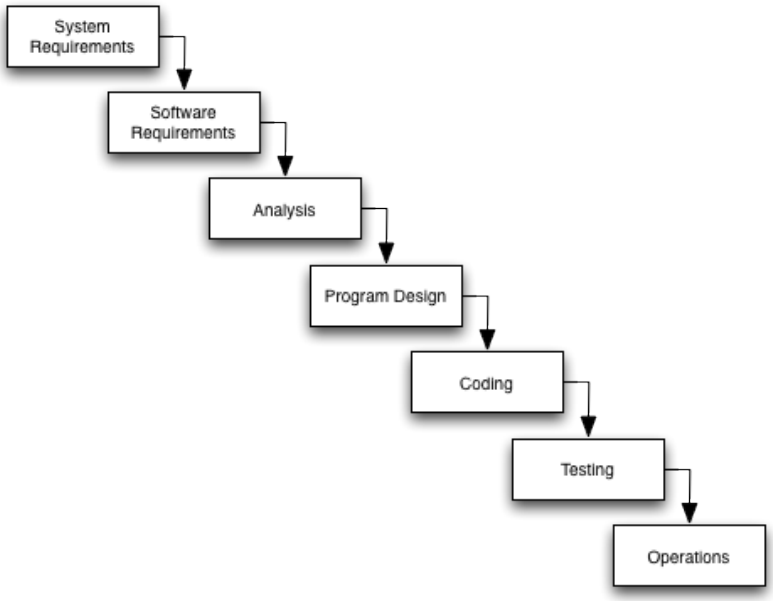


Figure 2.5: Waterfall Model, (Royce, 1987)

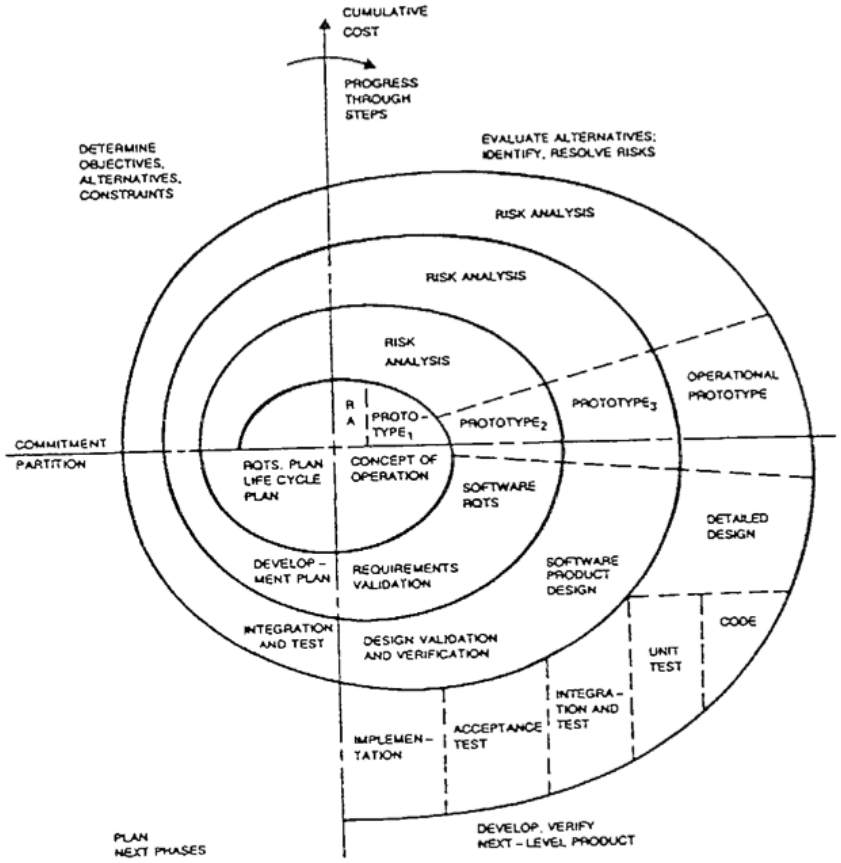


Figure 2.6: Spiral Model, (Boehm, 1986)

- The first phase starts with the identification of objectives and determination of alternatives and constraints.

- The second phase consists of risk analysis and prototype development.

- The third phase is the actual development phase. It can accommodate any appropriate development methodology. The exact methodology should be chosen by the projects needs.

- The fourth and final phase of each iteration consists of a review of the recent iteration.

2.2.3 Software Prototyping

Software prototyping as a methodology tries to solve the problem of uncertain requirements by creating a prototype in advance. The purpose of the prototype solely is to identify the requirements. In software prototyping Kordon and Luqi (2002) distinct two different approaches:
The advantage of the *throw-away approach* is the possibility to use techniques and assumptions not acceptable for the final product. This approach is used to prove feasibility at a relatively low cost. The disadvantage on the otherwise is apparently the waste of development effort.
The evolutionary approach tries to countervail the disadvantages of the first approach. Therefore, it tries to evolutionary develop several prototypes until the last prototype becomes the final product.

Kordon and Luqi (2002) mention Smalltalk in this context as a programming environment suitable for prototyping techniques as of its large set of predefined classes and simple programming language. As Smalltalk is not usable in production environment it is mentioned as tool for throw-away prototyping. The evolutionary approach benefits most from automatic program construction tools.
The experience of the practical project of this master's thesis fully supports these assumptions as all these technologies like a heavily Smalltalk influenced programming framework and code avoiding tools were used.

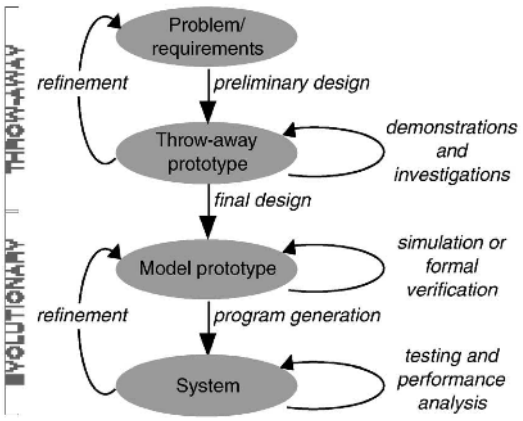


Figure 2.7: Evolutionary Approach, (Kordon and Luqi, 2002)

2.2.4 Agile Software Development

"Rapid change is a taunting task" states Poole (2006) and also a motivation for inventing new software development methodologies. Challenges might be new technologies but also other external influences like competition or legal requirements. From a business perspective, the most desirable business advantage is learning faster than the competitor. Therefore, reducing development cycle times allows faster adoption of new technologies resulting in business advance. Besides this, agile methodologies tend to increase quality of code at the same time. Agile software development is based on simple principles. First of all this methodology is based on small iterations. Each of these iterations defines a full software development process with working software at the end of the cycle. Poole (2006)
A team of software engineering experts formulated in 2001 the *Agile Manifesto* with the following words:

"We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more."

The first phrase "Individuals and interactions over processes and tools" also supports the name lightweight methodologies for agile methodologies. Lightweight in this context means methodologies with little formal requirements. From the second assumption "Working software over comprehensive documentation" results the characteristic of short iterations.
Short iterations with working software at the end of each, are a central element. Poole (2006) describes several advantages of short iterations. First of all, customer feedback can be obtained more frequently resulting in a faster solution of the development. Short iterations also imply shorter testing phases.

This can only be achieved by automated tests also influencing code quality in a positive way. Long iterations also tend to hide problems. Besides delaying feedback also the problem of "feature creep" comes into play. Long iterations implement large amounts of features. During this long periods always new features arise and create time pressure. With short iterations new features can be planned for future releases.

"Customer collaboration over contract negotiation" tries to mitigate the problem of unclear and changing requirements leading to the last phrase "responding to change over following a plan".

Although all agile software development methodologies have these principles in common several differences in complexity and level of detail exist. (Errath et al., 2004)
Some methodologies like the crystal series or scrum provide a higher level framework for software development, while others like Extreme Programming precisely describe a process. Several of these SDMs are introduced in the following subsections.

2.2.5 Crystal Clear

> *"Crystal Clear: A few key rules to get a small project into its safety zone."* (Cockburn, 2004)

Cockburn (2004) summarizes key features, which occurred frequently in successful projects as the following: Close communication, frequent delivery and reflective improvement. Crystal Clear (CC) does not pretend to be the best software development methodology but it claims to be easily adoptable by a team and leading to successful project completion.
Another interesting fact of CC is the motivation for its invention. CC does not try to cope with changing requirements as main motivation but with a focus on delivering projects in time at lower costs. Especially this motivation makes this SDM even more valuable for small teams, still allowing changing requirements due to the agile nature of the SDM.
Within the crystal series of SDMs, the "clear" methodology is suitable for small teams up to 8 people in less critical domains. Cockburn (2004) describes it as less demanding than XP but as a possible springboard to gain agile experience before starting with XP.

How does CC work?

Procedures are not the main focus of CC. If a project follows the CC methodology, it has to have certain properties independently of the procedures used to achieve them.
Three of the properties (frequent delivery, reflective improvement, osmotic communication) are mandatory for CC. The other four properties, strategies and techniques further increase the chance for success.

Properties

- **Frequent delivery** is a common agile practice and all the positive characteristics of it hold for this methodology as well. Cockburn (2004) defines a delivery as tested code delivered to real users every few months. The feedback

of real users is essential for development as well as the rate of progress that can be monitored. Also developers take advantage of this practice as they keep focussed and need to debug their development and deployment process.
The frequency of delivery depends on the type of application, as webapplications can be easily deployed in opposite to software installed on a large amount of client computers. In general, a "friendly user" is the best strategy which uses the software for testing purposes, but not in production.

- Cockburn (2004) mentions **Reflective Improvements** as critical factor. Disastrous projects already made a turnaround into successful ones just by reflecting on problems and working as a team to solve these. Meetings on a regular basis starting as soon as possible allow such drastic changes in projects.

- **Osmotic Communication** is suitable for small teams only, otherwise the background noise gets to disturbing, besides the physical limitations. It is all about background discussions where everyone can contribute its expertise or keeping focussed on its work. Every single question is posed to the whole team without asking. Therefore, the team has to be co-located in a single room with a supportive layout of the office. Cockburn (2004) mentions surprisingly little disturbance to the whole team from this practice.
  Expert users still need to be careful not to get distracted too much. If this is the case, a certain timeframe has to be established for expert users without distraction also mentioned as the "cone of silence" strategy.

The other optional properties are also be described briefly:

- **Personal Safety** is essential in a development team to allow reflective improvement. Trust is essential among development teams to freely speak about impediments to be solved.

- **Focus** is the combination of knowing the tasks with high business value and also the chance to work on these tasks without interruption for at least several hours a day. Also focus requires to assign developers to a maximum of one and a half projects at the same time.

- **Easy Access to Expert Users** grants multiple worthy assets to a development team. First of all, they provide testers for frequent delivery and therefore deliver feedback about the quality of a product. Design decisions can be made faster as feedback is always available within a reasonable timeframe and furthermore the user requirements are kept up to date as real users are already working with the product and reporting impediments in using the product.

- A **Technical Environment** is also a key property for successful software delivery. Source control management is the most essential basis for distributed software development. In addition, automated testing and frequent integration also increase the chances of success.

With these properties in mind, a project can easily be examined for possible improvements. In addition, several techniques and strategies are described in Cockburn (2004) to further improve the development process.

2.2.6 Extreme Programming

Extreme programming (XP) is probably the most widely used agile software development methodology. (Bates and Yates, 2008; Hussain et al., 2008c)
As an agile SDM, XP complies with the agile manifesto but still differentiates in its realization.
Beck (1999) admits that the individual practices used in XP are nothing new, but a combination of all these practices build a powerful approach to create software whilst requirements changing frequently. XP consists of a set of 12 major practices:

- Planning Game
- Small releases
- Metaphor
- Simple design
- Tests
- Refactoring

- Pair programming
- Continous integration
- Collective ownership
- On-site customer
- 40-hour week
- Open workspace
- Just rules

XP is based on strictly obeying all these principles. Nevertheless, one single process does not always fit all different types of projects and teams in its ”pure" form and therefore sometimes needs adaptation to be applied successfully. (Hussain et al., 2008a)

Process Description

Developers and customers unfortunately most of the time do not talk a common language. Therefore, XP defines user stories to cover the software features from a customers point of view. Stories have to follow three simple rules: They have to be business oriented, testable and estimable in terms of implementation effort. Typically an index card should cover all information necessary for implementing a task. Each iteration starts with the planning game. The customer defines a minimum set of features with the highest priority to create a usable software product. Beck (1999) compares this initial feature definition to shopping. Limited development resources can be defined as the budget and each item (feature) has a certain cost (development time). Therefore the customer can decide what to be implemented first.
Afterwards the developers divide the stories into tasks.
A task is the largest software fragment developers are working with. For each task a team of two programmers takes responsibility. The implementation of a task must not exceed a planned implementation time of up to three days.
After all tasks have been assigned, the development phase starts. Unit tests are

always written first. A task has to pass all tests to be accepted and completed. Automated unit testing creates confidence in the software and allows refactoring and code changes without worrying about introducing new bugs. All functionality covered with tests will remain intact. Another important aspect of unit tests is the documentation automatically created. Whilst integration, the customer defines functional tests for the current release.
As soon as all tasks are integrated a new release of a software can be created, or at least handed to the customer for evaluation and the next iteration starts again with the planning game.
Communication is a critical aspect of XP. The customer is integrated into the development process, not only for delivering information but also for decision making.

2.2.7 SCRUM

Scrum is another lightweight software development methodology, which resembles Boehms spiral model of software development.
The process can be summarized based on Rising and Janoff (2000) with the following steps:
At the beginning all initially planned features of the software are collected as tasks in the backlog. The customer prioritizes the tasks and therefore defines the order of implementation. A Task is the basic unit for features. Each task must be completed within one week. If a feature takes more developing effort, it has to be divided into different tasks. The backlog might change over time in terms of feature count and also priority of tasks. The rearrangement of the backlog takes place after each development iteration called *sprint*. This rearrangement allows the integration of changed requirements and new feature requests at later development stages. Furthermore the backlog allows tracking the current progress.
A typical iteration within this SDM lasts one to four weeks. At the start of each sprint the development team selects the amount of tasks to be completed during the sprint. The result of each of these iterations has to be a usable and deliverable software release. Time delays are not acceptable within an iteration. If the planned tasks exceed the assigned schedule, tasks are removed from the current sprint.

The daily scrum meeting always handles three questions about what has been accomplished so far, what where the obstacles to get over, and what is planned until the next scrum meeting. Within a scrum meeting, discussions of potential solutions are not welcome, not to exceed the given time frame of 30 minutes maximum. Another important factor of these meetings is the social aspect.
The scrum master leads the scrum meetings, is responsible for tracking the progress and recording decisions and actions.
The team consists at most of ten developers. This size has shown to be very effective in SW development.
Rising and Janoff (2000) describe a team as a tight integrated unit with well defined roles focusing on a single goal. Several components of the scrum methodology like scrum meetings support these team aspects. Also social loafing is mentioned in this context, which can be described in few words: The outcome per person of a team decreases as the size increases. Therefore, the small team size is an advantage over this social phenomenon.
Rising and Janoff (2000) characterizes the benefits of scrum as follows:
The frequent iterations allow adaptation of the development to changing requirements. Also the estimates tend to get better over time. Customers receive frequent releases and feedback gets delivered. The customer can experience the project progress and deliveries are always on time and trust between customers an developers builds.

2.2.8 Test Driven Development

Test Driven Development (TDD) is another heavily discussed SDM. It relies on the principle, to write automated tests before the actual implementation of production code also found in XP. An integral part of TDD is an automated test framework allowing quick execution of tests resulting in short development cycles. Several frameworks like *JUnit* exist to support test automation. The focus in TDD lies on units, small pieces of functional code to be developed.
Janzen and Saiedian (2005) also made some interesting findings on developer productivity and code quality. In general the code quality and error rate improved while productivity only slightly decreased.

According to Beck (2002), the process of TDD contains the following steps:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication

Quickly adding a test requires the test to cover only a small piece of new functionality. In the second step, the test suite is executed to make sure the required functionality is not already implemented and the new test case executed properly. The test is expected to fail at this point. Given the test a solution with minimal effort is now developed to satisfy the test requirements thus succeeding all tests. To restore proper code design and remove duplicated code, another phase is added to refactor the newly added code with tests ensuring functionality.

2.3 Reuse of Existing Software Components

Developing software often includes reusing existing software components. Code reuse has several advantages over newly developed components.
Besides reducing development time and costs also the quality of existing components often outperforms newly developed components (Morad and Kuflik, 2005).
Also the time to market can be drastically reduced by code reuse resulting in an advantage over competitors. Thus a main goal of software development is using and creating reusable components to generate synergies between multiple projects and increase productivity in the long term.
From a legal perspective reusing software components can create major problems. (Di Penta et al., 2010)
To identify possible threats to commercial software products resulting from code reuse systematic observation of reused code fragments and their license terms need to be considered. The research of Sojer and Henkel (2011) identified the reuse of "Internet code" as common source of code reuse. Even if internet code is freely

available and intended for public use, copyright laws still apply and need to be obeyed. Software developer professionals often ignore this fact by reusing Internet code in commercial products with uncertain license terms. If no explicit permission is given by the author, reuse of code is generally not allowed. Therefore, permission for usage and distribution has to be requested by the author.
To resolve this uncertain legal situation, several standardized open source licenses have been established and clearly state the general conditions for code reuse of available software components. Still the license has to be explicitly referred by the author to create legal protection.

2.3.1 The Open Source Definition

The Open Source Initiative[3] formulated the current open source definition. (OSI, 2011).

1. Free Redistribution (royalty free distribution of software)
2. Source Code (source code must be available for distribution)
3. Derived Works (modifications must be allowed and redistributable under same terms)
4. Integrity of The Author's Source Code (source-code distribution in modified form may only be restricted if build-time "patches" are allowed)
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavor (no restrictions in field of use)
7. Distribution of License
8. License Must Not Be Specific to a Product
9. License Must Not Restrict Other Software
10. License Must Be Technology-Neutral

[3] `http://www.opensource.org`

The open source definition itself does not require derived work to be distributed under the same license as the original. If software components should be reused in closed source software products, the licenses of all used software components need to be examined for certain properties. With closed source software components, the terms of use and distribution most of the time are clearly stated by the distributor often in specialized license agreements.

2.3.2 Comparison of Open Source Software Licenses

Several standard open source software (OSS) licenses exist for different purposes and most of them qualify for commercial reuse in closed source software projects. Only the GPL License requires the whole "work", in which to software component is integrated, to be published under the terms of GPL. All other licenses allow the commercial and closed source usage of the source code. There needs to be a distinction between "usage" and "modification" of the reused code. If the code is modified as well, the modifications must be redistributed depending on the original license still allowing closed source usage. Almost all OSS licenses require the original limited liability and copyright notice to be distributed with all derived work. Table 2.1 lists the most common open source licenses and their suitability for close source development.

License	Reuse in closed source	Publish modified source
Apache License, 2.0	Yes	No
BSD licenses	Yes	No
GNU General Public License (GPL)	No	Yes
GNU "Lesser" General Public License (LGPL)	Yes	Yes
Mozilla Public License (MPL)	Yes	Yes
MIT License	Yes	No

Table 2.1: Comparison of Open Source Software Licenses

2.4 Software Testing

Testing is an important activity in software engineering. Its target is to detect malfunctions and validate a systems behavior. Studies on software testing estimate the budget on software testing as high as up to 50 percent of total development costs. (Bertolino, 2007)
With such a large influence on development costs, testing has a very high potential for cost savings and should be taken into special consideration.
Software testing covers several aspects aiming at different objectives.
This section describes the software developers' point of view of testing for functional requirements. There are also non-functional requirements like performance, stability and finally usability explained in the subsequent chapter.
Software testing and validation has become a large field of research also discussing several other methods for software validation. For example, static and dynamic analysis inspecting software behavior or model checking working with formal specifications and for example abstraction to reduce complexity. Given the requirements of the developed system in this master's thesis relevant test methodologies are discussed in more detail.

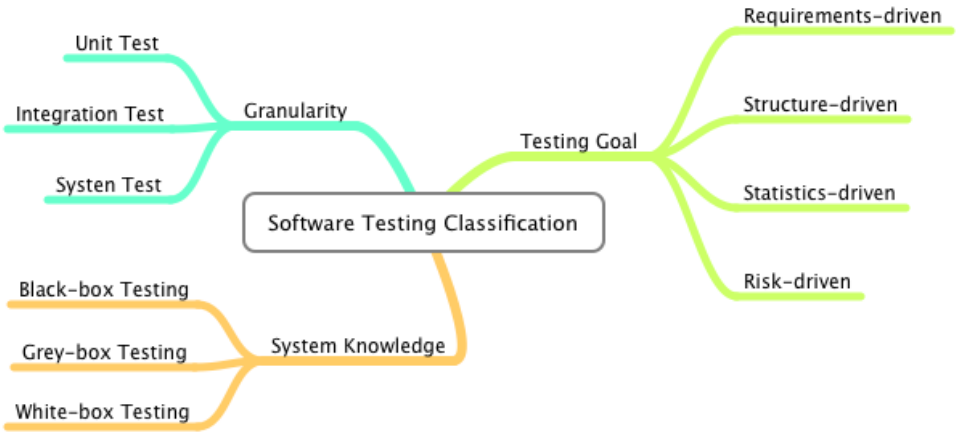


Figure 2.8: Software Testing Classification

Software testing for functional requirements can be structured under different aspects. Figure 2.8 represents the classification used in this master's thesis.

The test persons knowledge about the system under test separates white-box and black-box testing. White-box testing implies large knowledge about the system under test while black-box testing implies no background knowledge about the system at all. Both approaches are reasonable in different test scenarios and scopes.

Glass (2009) introduces two other dimensions for classifying testing. The first dimension classifies four goal driven approaches.

- Requirements-driven testing determines if the requirements of a software are fulfilled
- Structure-driven testing determines if separate software modules work as defined
- Statistics-driven testing determines the required reliability in typical usage scenarios
- Risk-driven testing determines the risks and vulnerabilities in high-reliability settings.

The second dimension of testing defines the phase in which testing takes place. Whereas *unit tests* focus on the smallest amount of functionality being tested often requires a *white-box* approach, testing complete systems should be preferably tested without insight into the system to reflect the end users knowledge about a system. There are also other layers of testing in between to regulate the granularity of tests for different scenarios.
Integration testing takes place at an intermediate level to verify the collaboration of different software modules. Still some insight in the system being build is necessary at least at a *grey-box* testing level.
System testing is performed on software products integrated in larger systems. This can be performed in a black-box approach where no knowledge about the internal functionality of a software is required.

2.5 Requirements and Usability Engineering

> *"The gap between Software Engineering and Human-Computer Interaction should be closed through the integration of usability engineering and*

requirements engineering." (Paech and Kohler, 2003)

Based on this quotation the following section gives a short introduction to the two domains of requirements and usability engineering. While software development methodologies mitigate the technical risks in software engineering, requirements and usability engineering are necessary to create a suitable environment for a successful software development project. Both are closely related and of extraordinary importance to software projects.

2.5.1 Requirements Engineering

> *"The requirements for a system, in enough detail for its development, do not arise naturally. Instead, they need to be engineered and have continuing review and revision."* (Bell and Thayer, 1976)

The target of Requirements engineering (RE) is the identification of goals to be achieved by the designated system. Therefore, it contains the processes of analysis, elicitation, specification, assessment, negotiation, documentation and evolution. (van Lamsweerde, 2000)
Nuseibeh and Easterbrook (2000) even more emphasize the importance of requirements engineering by defining the primary factor of success of a software system as the degree to which it meets the purpose for which it was intended. Most of the requirements engineering work is conducted at initial stages of a software development project, as project changes are more expensive in later stages of the project lifecycle.
Elicitation is the process of identifying goals, objectives and motives for building a software system. The most important result of the elicitation process is the identification of stakeholders, all individuals and organizations involved in development and usage of the software product and their needs.
Several elicitation techniques are available to gain these desired results. Traditional techniques include questionnaires, interviews, surveys, organizational charts, process models, standards and manuals of existing systems.
Prototyping is a technique suitable in situations with a large level of uncertainty of requirements. Group elicitation techniques can be combined with prototyping to

provoke discussions or as basis for questionnaires. The selection of a certain technique always depends on the availability of time and resources.
The next step after elicitation is the phase of analysis and specification of requirements, also referred as modeling. Categories of modeling are enterprise modeling, data modeling, behavioral modeling, domain modeling and modeling of non-functional requirements. The benefit of modeling is the possibility to analyze them afterwards. Negotiation of elicited and specified requirements is another complex topic in requirements engineering as stakeholders tend to have conflictive goals.
The final task in requirements engineering concentrates on managing change of software systems to keep up with their environment. This also requires change and evolution of their requirements. (Nuseibeh and Easterbrook, 2000)

2.5.2 Usability

> *"Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."* (ISO, 1998)

Usability can not be defined by a single property. Nielsen (1993) therefore explains it with the following five properties:

- Learnability: The time a user needs to start productive work with the system
- Efficiency: It should allow a high level of productivity
- Memorability: Once learned, operations should be easy to remember after a certain period of time
- Errors: A low error rate is obligatory, it must be easy to recover from errors
- Satisfaction: Users should like using the system to make them subjectively pleased

For good usability all of the five attributes must be considered equally. None of the properties can be neglected.

Usability Engineering

Usability engineering is an iterative process throughout the whole lifecycle of a product. Gould and Lewis (1983) initially defined the following principles for design:

- Early Focus on Users and Tasks: Designers must understand who the users will be and what the work is that needs to be accomplished
- Empirical Measurement: Performance and reactions of users using the system should be observed recorded and analyzed
- Iterative Design: The phases of design, implementation and test must be repeated until a sufficient level of quality in terms of usability is achieved

There are many usability methods suitable for different development situations. Holzinger (2005) gives a good overview of these methods and their optimal fields of use.

	Inspection Methods			Test Methods		
	Heuristic Evaluation	Cognitive Walkthrough	Action Analysis	Thinking Aloud	Field Observation	Question-naires
Applicably in Phase	all	all	design	design	final testing	all
Required Time	low	medium	high	high	medium	low
Needed Users	none	none	none	3+	20+	30+
Required Evaluators	3+	3+	1-2	1	1+	1
Required Equipment	low	low	low	high	medium	low
Required Expertise	medium	high	high	medium	high	low
Intrusive	no	no	no	yes	yes	no

Table 2.2: Comparison of Usability Evaluation Techniques, (Holzinger, 2005)

- Heuristic Evaluation

  Heuristic Evaluation (HE) is performed by looking at a user interface to be evaluated. The goal is to form an opinion about the inspected user interface.

HE is often performed on intuitive criteria but by using standardized metrics results become comparable and objective. Nielsen and Molich (1990) mentions in this context the collection of usability guidelines to be used.

- Cognitive Walkthrough
  Lewis et al. (1990) describe the Cognitive Walkthrough (CW) as structured evaluation process. It takes a list of questions to focus the designer's attention on individual aspects of the interface. First tasks to evaluate the design are specified. Second the series of actions performed by a user to perform the tasks are collected. Each step in the series of actions is evaluated if usability problems are expected or not.

- Action Analysis
  In the field of Action Analysis, Holzinger (2005) differentiates two methods. Formal and back-of-the-envelope action analysis. Formal analysis records each user interaction in its most basic form, for example move mouse, select menu, to complete a certain task. Therefore it is also called key-stroke analysis. Back-of-the-envelope action analysis is less detailed thus less time consuming but also gives less precise results. In general, action analysis allows prediction of how long it takes to perform certain tasks.

- Thinking Aloud
  Lewis (1982) characterizes thinking aloud as a method to analyze the mental processes of participants executing predefined tasks by asking them to make spoken comments on the tasks performed. The method identifies cognitive problems of people learning how to use a given user interface.
  Video recording is used to capture the content of the screen and the participants comments which he is requested to give as an ideally constant stream of feedback on each operation he performs or what he expects to happen next. According to Holzinger (2006), the recording of facial expressions and gestures also allows inference on working habits and therefore should be recorded as well. Still the method has limitations. The degree of realism is limited due to the presence of an observer and the participant commenting his actions in contrast to real work situations. The accuracy is also heavily dependent on the participant performing the tasks.

As benefits of the methodology, Lewis (1982) mentions the following advantages. The method not only helps to identify problems but also why they occur and what has caused the trouble. The vocabulary of the user is additionally captured to make the interface more appropriate for the targeted user group. The attitude of a participant towards the interface can also be analyzed, which is critical to commercial projects.

To detect a reasonable amount of usability issues, Nielsen (1994) defines the minimum amount of test participants as three to five which is relatively small and allows frequent testing.

Another advantage is the possibility to use mockups instead of working systems for testing interface behavior as time and task success is not the main focus.

- Field Observation

  Field observation consists of taking notes about users interacting with the system in their natural working environment. The results of observation are also influenced by the Hawthorne effect (Adair, 1984), therefore observers should attract as little attention as possible. Observation is used to detect major usability problems. Video recording of observations is an option but given the time consuming process of analyzing, time is better spent in testing more subjects. Data logging can also be used in field observations as addition for generating more detailed usability information. (Holzinger, 2005)

- Questionnaires

  The measurement of the subjective user satisfaction can be conducted with questionnaires. Usability questionnaires are classified as indirect testing method as not the user interface itself is examined, but what users actually think about a user interface. The major disadvantage is the sufficient large amount of responses to validate the test. In general less usability issues are identified with questionnaires than with other testing methods. Holzinger (2005) therefore recommends the combination of questionnaires with direct testing methods.

2.6 Agile Software Development and Usability Engineering

Agile software development tries to satisfy the customer by integrating him into the development process. However fulfilling a customers needs does not always result in usable software. (Patton, 2002)
Therefore, dedicated usability engineering is also added to the process of agile software development hopefully resulting in better software.
There has already been some research on this topic. (Lee and McCrickard, 2007; Holzinger and Errath, 2004; Holzinger et al., 2005; Wolkerstorfer et al., 2008; Fox et al., 2008; Hussain et al., 2008b)
Mostly academic software development projects have been examined from this point of view. Therefore, XP was the main choice of SDM.
Fox et al. (2008) is an exception to this, researching ten different projects in a commercial context using scrum related software development methodologies.
The main gap between agile methodologies and usability engineering is the initial effort put into requirements analysis and overall design, before starting an iterative development process. Both methodologies afterwards conduct iterative improvements until a final product emerges. (Fox et al., 2008)
Lee and McCrickard (2007) mentions another discrepancy between developers and usability engineers in terms of communication mitigating this problem with a common feature description as link for both perceptions.
Obendorf and Finck (2008) take the same approach creating scenarios for supporting a common understanding of a work task.
Fox et al. (2008) further distinguished between three possible combinations of user centered design specialists (UCDS) and software engineers.

1. The Specialist Approach consists of customers, a UCDS and a group of software engineers. The UCDS manages the initial phase of requirements gathering with the customer and lo-fi prototyping. After this step the development team meets with the UCDS creating iteratively an UI prototype. The result of each iteration is evaluated by the UCDS working in parallel with the software

development team generating requirements for future iterations. Communication is a critical challenge in this approach with the UCDS as bridge between customers and the software engineers.

2. The Generalist Approach consists of two main roles, the customers and the developers also taking responsibility for usability engineering.
   Fox et al. (2008) mention this approach as very responsive to usability issues. On the other hand the developers were not formally trained on the topic of user centered design resulting in downsides in terms of usability engineering. Another interesting finding was that always multiple developers acted as UCDS getting involved with usability engineering.

3. The Generalist Specialist Approach consisted of the customer, and at least one developer formally trained in UCD besides general software developers. This is also mentioned as the difference to the Specialist Approach always only involving one single UCDS.

This separation of roles in Agile Development Projects conducting user centered design can also be applied to this master's thesis development project. As all of the developers had at least some experience in usability engineering, the project can be categorized as the Generalist Specialist Approach.

3. Materials and Methods

This chapter gives a detailed description of the development infrastructure in use and the development process. Several aspects regarding software development and usability engineering are discussed.

3.1 Development Process

Developing a successful software product requires not only a technical solution but also user satisfaction. Therefore, the user plays a central role in software development called user centered design and should be integrated into the development process from scratch. (Holzinger and Brown, 2008)
Almost all modern software development methodologies have the user centered approach in common. Therefore, it is necessary to define a development process which integrates the user into the development process. There have been several approaches on integrating usability engineering with software development methodologies before. (Holzinger et al., 2005)
This is an approach from a generalist specialists view within the classification of Fox et al. (2008). Adding additional usability engineering methodologies as extension to the CC principles seamlessly integrates with the philosophy of CC.
The whole development methodology is based on adding additions in critical regions. It does not require adapting each aspect in an extreme manner, but adapting the principles with the largest improvement for a given effort. If usability is a relevant criteria for a development project, the team has to apply relevant techniques to increase projects value for the customer. (Figure 3.1)

Taking the same approach as Holzinger et al. (2005), each iteration is extended by relevant usability engineering methods. In contrast to the educational back-

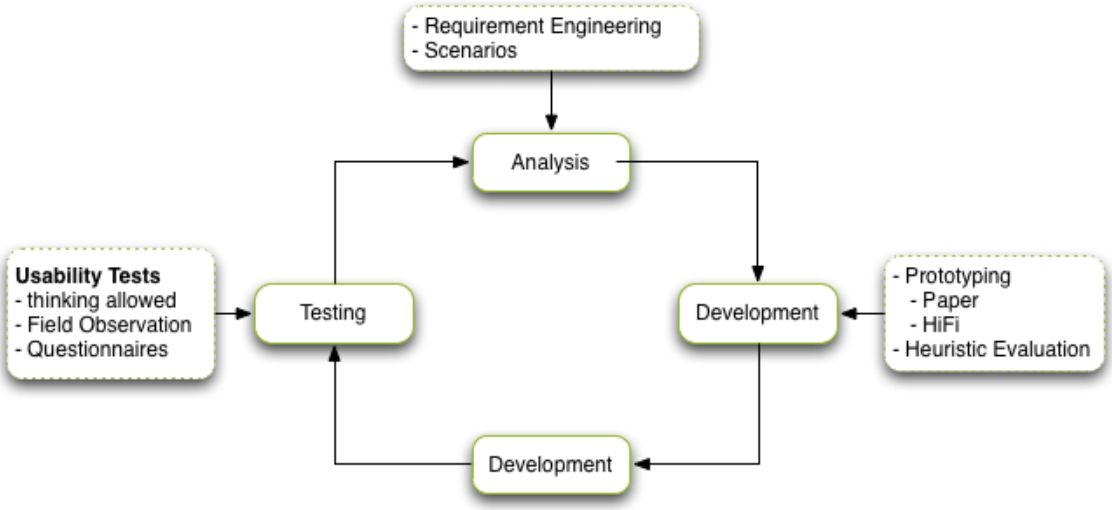


Figure 3.1: CC + UE Methodology

ground, also economical application of usability methodologies is relevant. Users can contribute to the software development process in several ways. Starting in the early design phase of each iteration, identifying users actual needs is the first step. User requirements need to be elaborated to define development goals for further development.

3.1.1 Analysis

The analysis phase needs special consideration for combining UE and CC. With agile software development, the usually long lasting analysis phase has to be shortened to meet the agile property of frequent iterations of tested and working software. To bridge this gap, the amount of work done is reduced to the least necessary set of features. Each identified requirement is documented as scenario. Given a fixed delivery date, only as much scenarios are selected to be implemented as possible within the planned timeframe as with the SCRUM methodology. (Rising and Janoff, 2000)

The value of actual user feedback on software being used in its planned environment, tends to be higher than theoretically gathered user requirements. Also the problem of users not knowing their actual needs can be avoided this way.

3.1.2 Design

In the design phase solutions for previously selected requirements are developed. For new user interfaces, paper prototyping is an efficient tool for development. On the software development side, classical development tools like UML modeling have proven their usefulness. To reduce cost and the necessary amount of iterations, all new user interfaces need to be carefully evaluated with the end users mental models in mind.
Several developers create individual mockups. Afterwards the different solutions are evaluated. The more possible solutions to choose from, the better. At least three independent solutions should be developed. Afterwards, a heuristic evaluation is performed on all designs to identify pros and cons of each solution.
Given those results, a solution using the best of all proposals is created. Personal involvement can be an issue within this process as the own proposal often is considered the best solution. Therefore, objective criteria for selection solutions should be established.
For best results, an experienced end user on site would increase the quality of results a lot. With modern means of communication, the end user can also be located far away with regard to different time zones. Screen sharing, webcams and VoIP solutions allow fast responses with reasonable limitations.

3.1.3 Development

The development phase is not directly influenced by usability engineering methodologies. Therefore CC recommendations are followed. Still development is important as the forming phase within each iteration.

Code Reuse in LineProducer

In LineProducer, a lot of existing software components have been reused during the development. As this is a sensitive topic in commercial software development, all components and licenses are listed in Table 3.1.
All licenses used allow the commercial distribution of the software without redistribution of the source code. In addition, the authors are credited within the software

SW Component	License Type	Component Description
Sparkle	MIT License	Sparkle is an Framework providing automated Client updates
Flot	modified MIT License	Flot is a Javascript Framework for graph plotting included as source
Amber Framework	BSD License	Amber adds custom UI controls and networking wrappers
BGHudAppKit	BSD License	BGHudAppkit adds specialized HUD styled UI components
SSCrypto	BSD License	SSCrypto adds an Objective-C wrapper for cryptographic operations
BWToolkit	BSD License	BWToolkit is a UI Component Framework
FeedbackReporter	Apache License	FeedbackReporter is used for automatically reporting bugs to the developers
MGTemplateEngine	BSD license	MGTemplateEngine is used for creating HTML styled export documents via templates.
JRSwizzle	MIT License	JRSwizzle adds wrappers for Objective-C runtime features
DDLog	BSD License	DDLog adds a more sophisticated logging functionality to the software

Table 3.1: Reused Software Components in LineProducer

to meet the terms of the licenses where the attribution of the original author is claimed.

3.1.4 Test

Testing is an integral part of every software development project. Within the development of LineProducer, different test targets have been identified. Testing for correct functionality was one important target. Therefore testing facilities have been established at different levels to meet functional requirements.

Unit and Integration Testing

At the lowest level, unit tests deliver reasonable verification of correct functionality. Given a large software product covering the entire functionality with unit tests would result in tremendous effort in development. Also a lot of functionality can not be covered with unit tests, because of the limited scope of a unit test. Within LineProducer the calculation of budgets has been identified to be critical for correct operation. Therefore the model layer in the classic model-view-controller (MVC) design pattern was identified to be worth the effort of writing automated tests. Within the model layer, all classes and methods related to calculation of budgets are covered with tests. In addition, some controller relevant for calculation also have been added to the testing framework.

Most of the test cases require a fully functional model layer for testing. The model of LineProducer is based on Core Data, an object relational framework developed by Apple. Therefore a set of common base classes (see Figure 3.2) was established to allow easy testing of model classes. This base classes deliver a fully functional Core Data stack containing necessary prerequisites of a Core Data application. Also the integration of custom model classes with Core Data is covered in the unit tests.

System Testing

To verify correct functionality of LineProducer as a whole system, a manual test setup was designed. All required functionality was specified within test cases and

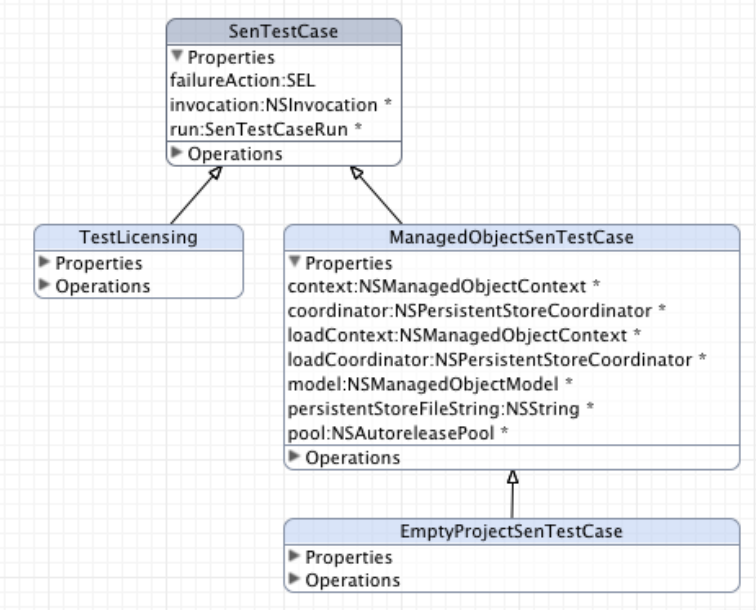


Figure 3.2: Test Classes Diagram

organized by the Testlink[1]. A test case consists of a precondition, necessary steps to reproduce the test case and a brief description of expected results. If the expected result is not achieved, a bug can be noted in the test system and the final report delivered to the development team. For each release an external tester performed all specified tests and therefore added additional verification of functionality.

Usability engineering adds new aspects to this phase. Not only functionality is verified but also nonfunctional requirements as standard usability properties are being evaluated. (See 2.5.2 for additional information on usability properties.) During this phase mostly usability testing methodologies are relevant. Selecting an adequate method for evaluation always depends on available resources. Holzinger (2005) already evaluated usability engineering methodologies for their best suited field of use. Newly detected usability issues during the testing phase are added to the list of development tasks for the next iteration.

3.1.5 Integrated Development Process

Although the typical software engineering process is separated into distinct phases, each iteration through the four phases have to be seen as an integrated process.

[1] `http://www.teamst.org`

The process typically starts with a feature request or bug report. Both types result in a defined work package or ticket recorded in the ticketing system waiting for being completed. Software contains bugs, therefore efficient handling of existing and avoiding future bugs is a key element of software development. (Figure 3.3)

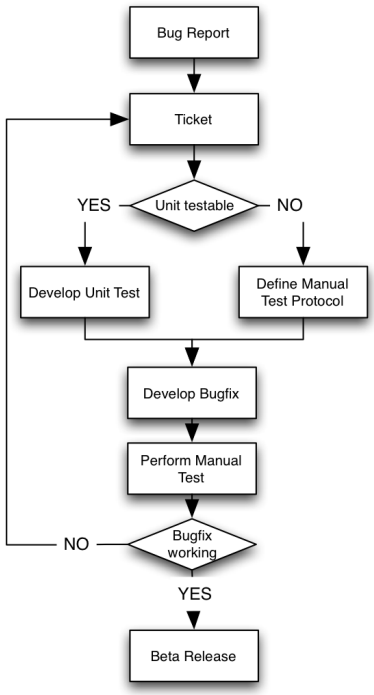


Figure 3.3: Bug Fixing Process

A feature request typically requires the development team to develop a solution for a new problem. (Figure 3.4) Creating good solutions takes several attempts often not available because of time and cost constraints.

3.1.6 Crystal Clear in context of the current Software Project

Within this section the project is evaluated regarding compliance with the CC software development methodology. The following properties were established during the development of the software.

Frequent Delivery

Initially frequent delivery unfortunately was not established. During the first five months not a single release was built to be delivered to test users. Development

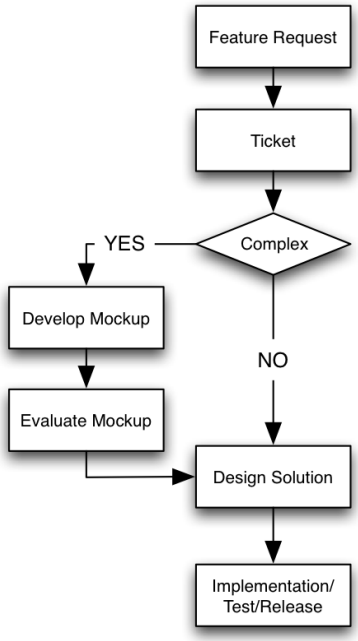


Figure 3.4: Feature Request Workflow

was completely based on input of expert users, not using the software in productive environments. With the first real deployment to external test users, several efforts were taken to achieve the property. The deployment system was completely automated with scripts. With a single click a complete release build can be created and uploaded for distribution to customers and test users. To support end users updating the software, an update framework was integrated into the software to allow updating with a single click as well. The project timetable at this point required frequent delivery as certain presentation versions needed to be completed.

A typical deployment process of a software product requires several steps.

1. Build project with release build settings: When building software for distribution, several compiler options are set to optimize a binary for execution. Also debug symbols are removed to hide internal structure of a software product.

2. Create package for distribution: Multiple packages are created. Mac OS X software typically is deployed as a disk image. Such images contain an application bundle, documentation and a well designed installation instruction. For the auto update process, an additional compressed archive is generated containing the new binary to be installed automatically.

3. Cryptographically sign package for automated update installation: To secure the automated update process, the bundle is signed by the developer to protect the auto update process from interception by third parties. An initial public key for validation is delivered to the customer with the original software download.

4. Write release notes: Release notes summarize the most important changes from a users perspective and are essential to inform customers of changes and bugfixes. Often this information is the only channel to communicate with customers.

5. Upload package to webserver for distribution: Also such trivial tasks as uploading a new binary are automated to simplify the process of distribution.

Also from a client perspective, this process has to be as convenient as possible. Therefore a third party framework[2] has been added to the software to automate the notification, download and installation of updates.

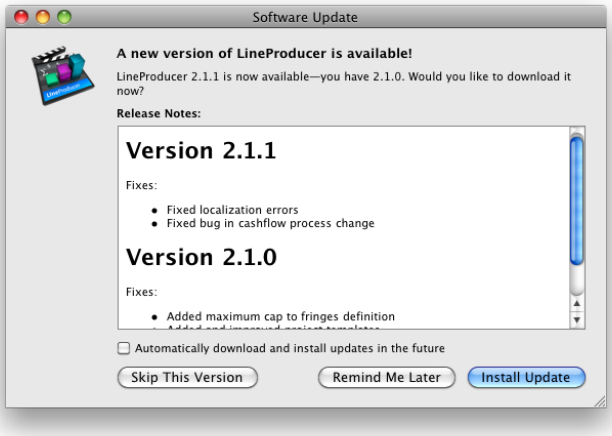


Figure 3.5: LineProducer Client Update

Frequent delivery is aimed at generating customer feedback already in early stages. Therefore the communication channel from the customer to the developer has to be convenient to use for both sides. Especially with software testers it is essential to deliver as much information about an error or problem as possible from the user to the developer. A stacktrace often simplifies the tracking of an error a

[2] `http://sparkle.andymatuschak.org/`

lot. Within test builds, the logging of an application can deliver great insights on application usage. Therefore it is necessary to integrate frameworks for delivering customer feedback into the software.
Standardized usage logging would even further enhance the quality and usefulness of information. Holzinger et al. (2011) describe an approach to use aspect oriented programming for extracting usage information. Using this new approach the amount of code and complexity necessary for integrating logging mechanisms spread across the whole system could be drastically reduced.
For LineProducer the FeedbackReporter[3] framework was integrated. (see Figure 3.6)
This framework allows seamless integration into existing software. If an error occurs the feedback reporter automatically asks the user for permission to send error information to the developer. As this information can contain privacy related information full disclosure to the user about information delivered is essential.

The delivered information includes:

- System information
- Custom message to be entered by the user to reproduce a problem
- Optional contact information
- Logfiles
- Application settings

Reflective Improvement

Reflective improvement occurred most of the time by informal meetings. At the end of feature discussions, team members explained critical parts and necessary changes from their point of view. For example, establishing the unit test system resulted from such informal meetings. Crystal Clear recommends two or more meetings per iteration. Within the current project, this property can be considered fully achieved.

[3] `http://vafer.org/projects/feedbackreporter/`

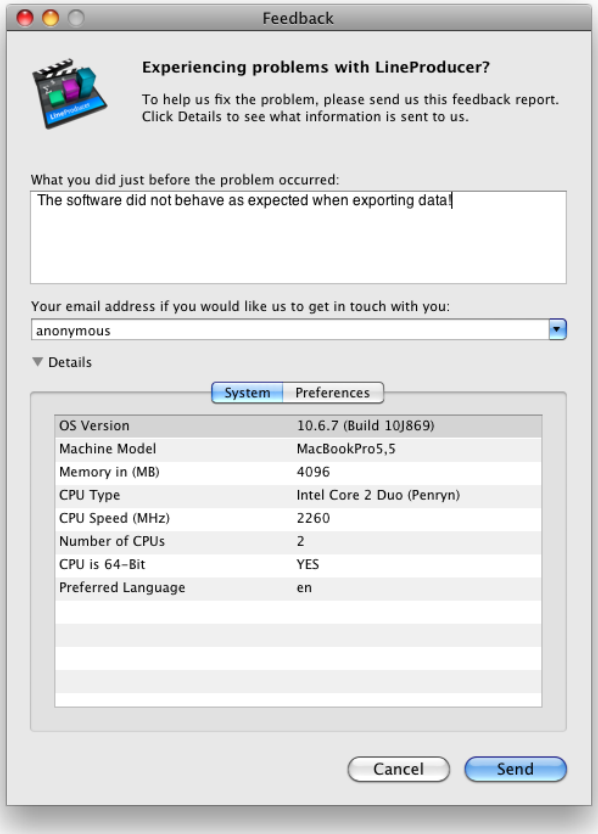


Figure 3.6: LineProducer Feedback Reporter

Osmotic Communication

Osmotic Communication is more or less contradictory to other agile development methodologies, as it admits the expertise of different team members. Not every team member needs to know each part of the system, but there is always someone within range to give answers to urgent questions. The current layout of workplaces shown in Figure 3.7 also supports this property. All four developers are seated next to each other. This layout could be further improved by creating a U-shaped desk arrangement giving easy access to each others monitor. Cockburn (2004) warned of the risk of overloading experts making them unable to complete their own tasks. No formal rules to handle this problem had to be established. The interrupted developers themselves made others aware of their situation unable to complete their own tasks. Such situations always were handled in a friendly manner and accepted by other team members.

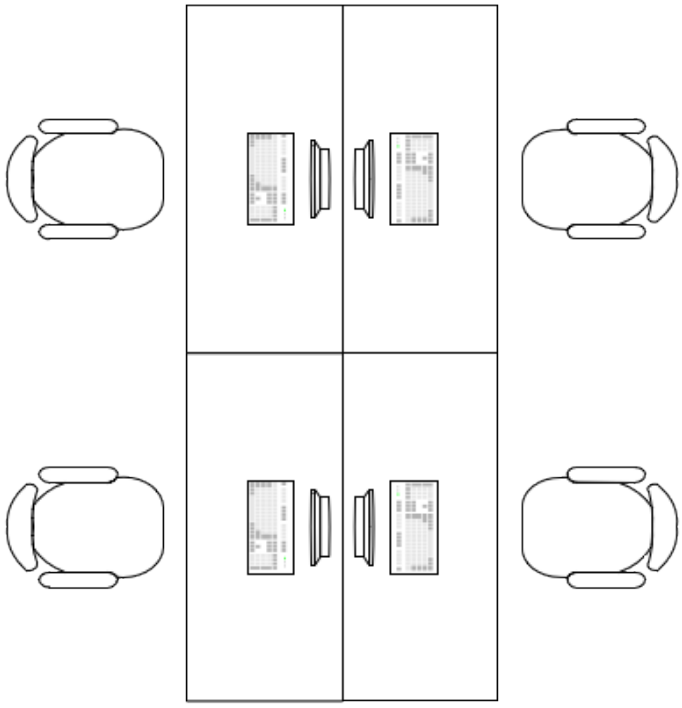


Figure 3.7: Office Layout for Osmotic Communication

Easy Access to Expert Users

Direct access to expert users is essential for a software projects success. (Keil and Carmel, 1995)
Within the current project, the property was fulfilled by integrating an expert user into the development team. Feedback from the expert most of the time could be received within a day. Although the expert user was not on site, using VoIP communication technologies and especially screen sharing drastically increased the quality of information. Even with expert users of a certain domain, the questions often were not answered as desired by the developers. This was caused by communication difficulties between the partners backgrounds. Asking questions in a way to receive usable answers pointed out to be a important competency. The decrease of interactions throughout the project lifetime mentioned in Cockburn (2004) was also experienced within the current project.

Technical Environment

The infrastructure provided for this project is described in Section 3.2, therefore will not be explained in this section. The minimum requirements by CC were definitely fulfilled.

3.2 Development Infrastructure

Establishing the infrastructure for developing a software product includes a lot more than providing a text editor and compiler. Especially with several developers involved measures have to be taken to coordinate collaboration and sustainable development. This section therefore describes the infrastructure used to develop LineProducer.

3.2.1 Documentation

Documentation is essential for a software project to remain maintainable. This includes general process descriptions, software architecture documentation and specification on interfaces. For internal documentation a wiki system (see Figure 3.8)

has been used and has proven to be suitable. A lot of documentation was written directly in the source files and as commit messages in the source control system as well.

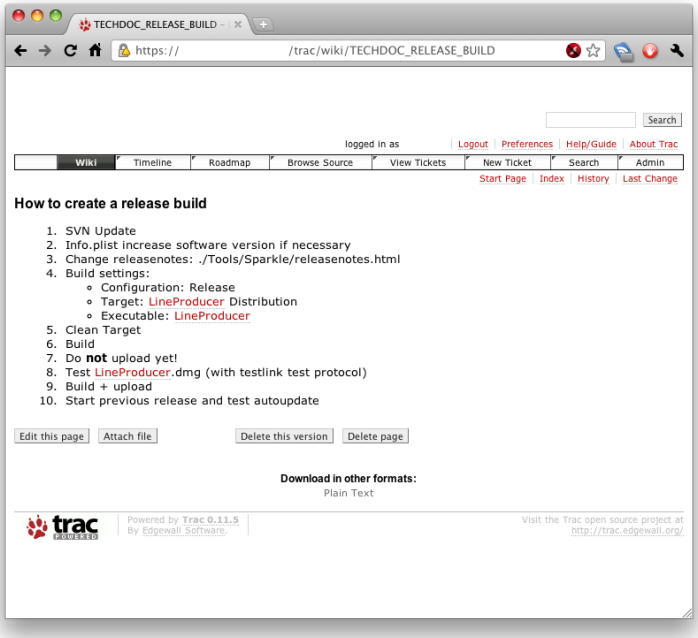


Figure 3.8: Developer Documentation using a Wiki System

3.2.2 Source Control Management

For team development a server running source control management is essential. In this project Subversion (SVN) was used to manage development collaboration between multiple developers.

This choice was based on experience from past projects. In depth practice is essential for efficient working with source control systems, therefore other probably more powerful systems like git were postponed to future projects. Some general guidelines for using SVN were established like descriptive commit messages and never to check-in not working code can be assumed generally accepted. The team was also encouraged to check in as often as possible, to reduce the chance of data loss and increase code quality as functionality had to be divided into smaller pieces. Another very useful feature was the possibility to track changes in case of newly introduced bugs and also understanding changes made by other team members. The commit

messages turned out to be a valuable part of the documentation. Also the mechanisms of branching and merging turned out to be valuable to create a testbed for larger refactoring tasks and newly introduced features not ready to be deployed to the main branch of development. (see Figure 3.9)

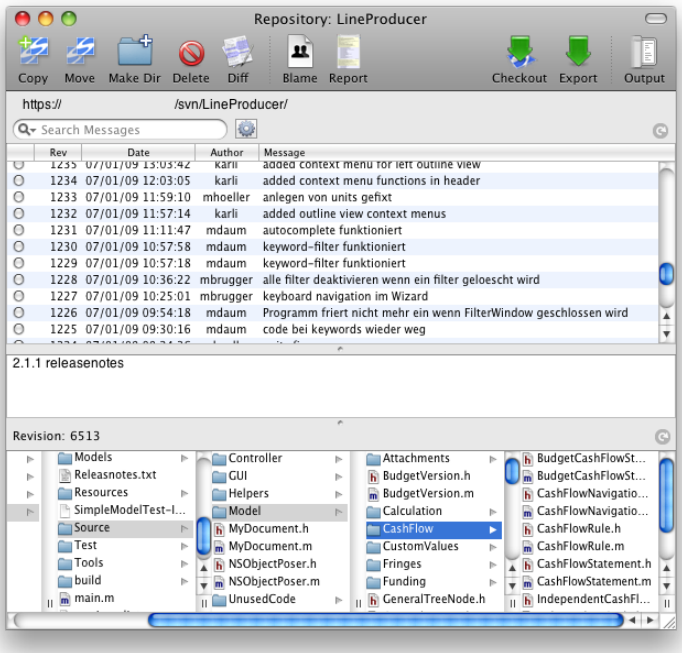


Figure 3.9: Source Control Management using Subversion and svnX

3.2.3 Bugtracking and Task Management

To track development progress and distribute tasks trac[4], a ticketing system is used. The system is closely coupled with the source control system to create a comprehensive work history. The combination of ticket and source control management allows broad performance measurements. (see Figure 3.10)

Tickets were grouped by severity, target release, software component priority and type. Software components described different aspects of the software like import/-export, budgeting related functionality or general preferences and several others. Types of tickets contained enhancement, task and defect. This classification of tickets granted an overview of tasks to be done and their priority to the project. Also

[4] `http://trac.edgewall.org/`

the workload of different developers could be measured and better distributed. Trac only supports general descriptions of tasks and bugs. Feiner et al. (2010) describe a system to include special information on usability issues in ticketing systems and would be an interesting extension to the current system.

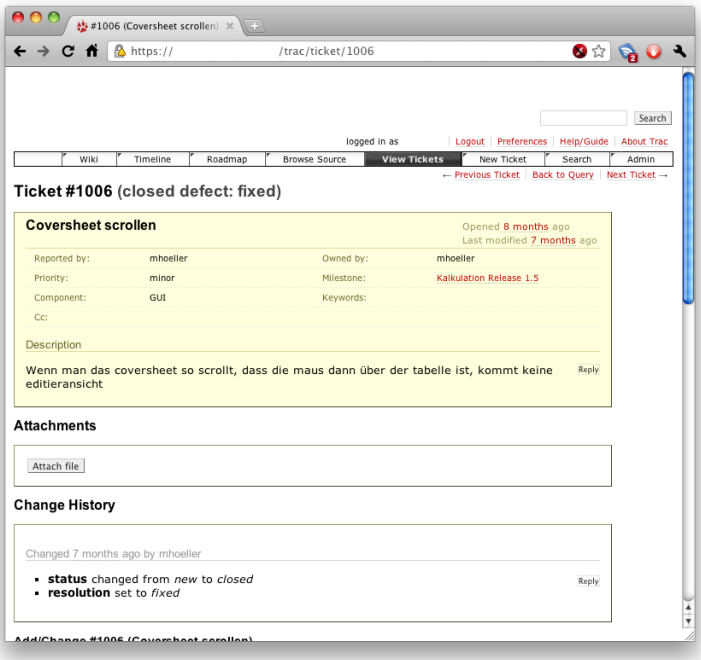


Figure 3.10: Trac Bugtracking

3.2.4 Integrated Development Environment

For developing Mac OS X software, Xcode is the preferred development environment of most developers. It contains all necessary tools of a modern IDE (Integrated Development Environment). The code editor supports syntax highlighting, intellisense like command completion and automatic code formatting. Therefore it allows comfortable writing of code. Also a direct access to the Cocoa reference documentation is available with a single mouse click allowing fast access to API specification of a certain class or method. All these features greatly enhance the development speed. Navigating, searching, replacing and refactoring code gains relevance especially in larger projects. All these operations are supported by the Xcode IDE (see Figure 3.11).

Besides code generation, several other tools are closely integrated into the de-

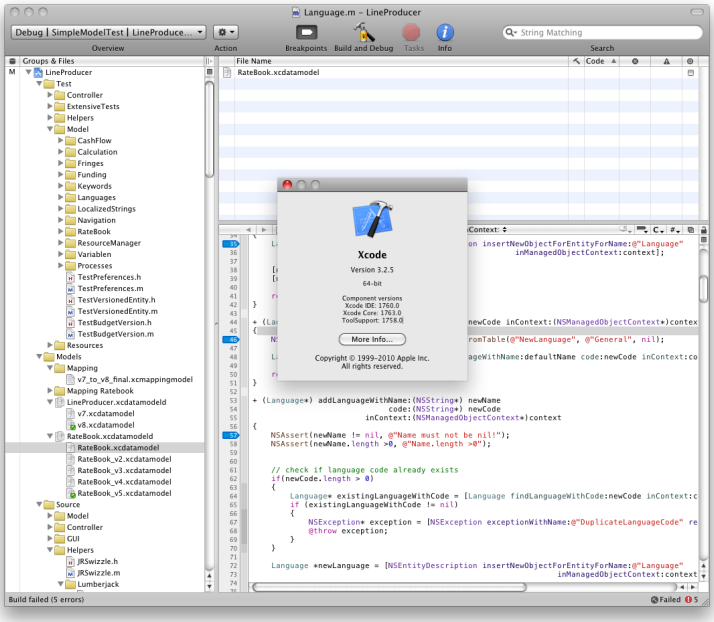


Figure 3.11: Xcode IDE

velopment workflow with Xcode. Important for development teams is the SCM integration into the development environment allowing quick access to the source repository to track and push changes. It is also very helpful to see available changes in the SCM repository. Xcode allows the SCM integration of Subversion, CVS and Perforce. The latest version of Xcode also integrates direct access to git repositories. Some SCM operations are not supported by the Xcode wrapper and have to be performed in a specialized SCM tool or terminal.

Another substantial task in developing software is build automation. Xcode allows the management of multiple build configurations for release and development. Automating the build process is essential for an integrated development workflow. Using custom scripted build phases, the whole deployment process can be automated.

Often more time is spent on debugging software than actually writing code. Therefore the integration of a debugger is essential for an IDE. Xcode provides a graphical frontend to GDB, the standard debugger for most UNIX-like systems. The most used tasks, like controlling the execution flow with breakpoints or manually stepping through the code execution can easily be performed. (see Figure 3.12)

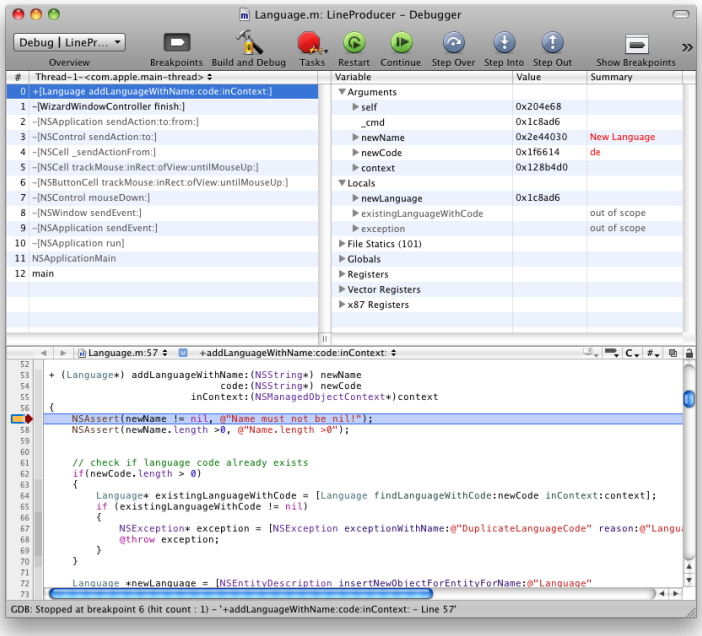


Figure 3.12: Xcode Frontend for GDB

Another high-level tool is the integrated data modeling tool for Core Data (Figure 3.13), a persistence framework provided by Apple. Using this tool, data models can easily be created and modified with a graphical user interface. Configuring data migration is another key feature of the Core Data model editor.

Xcode also provides a GUI designer called Interface Builder. (see Figure 3.14) Using drag and drop to arrange interface elements allows rapid prototyping of new GUIs with easy integration of the result in the final software system.

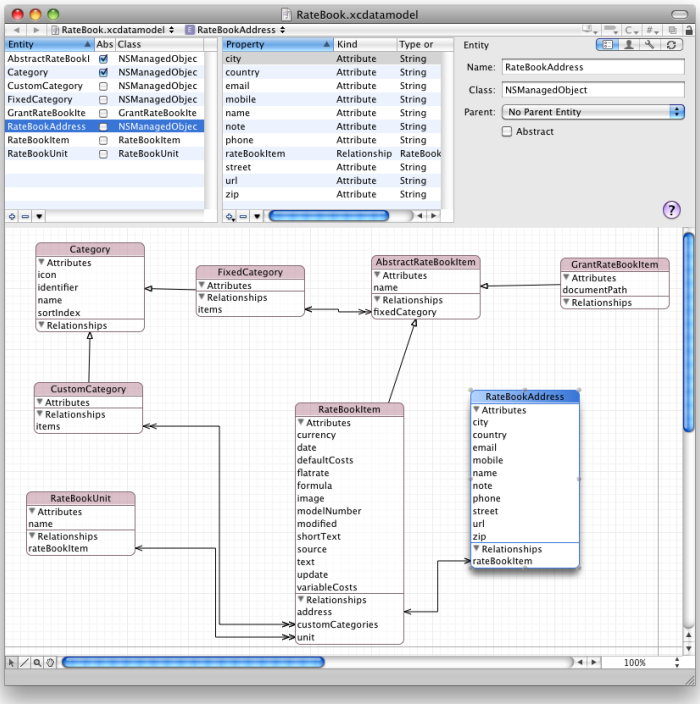


Figure 3.13: Core Data Model Editor

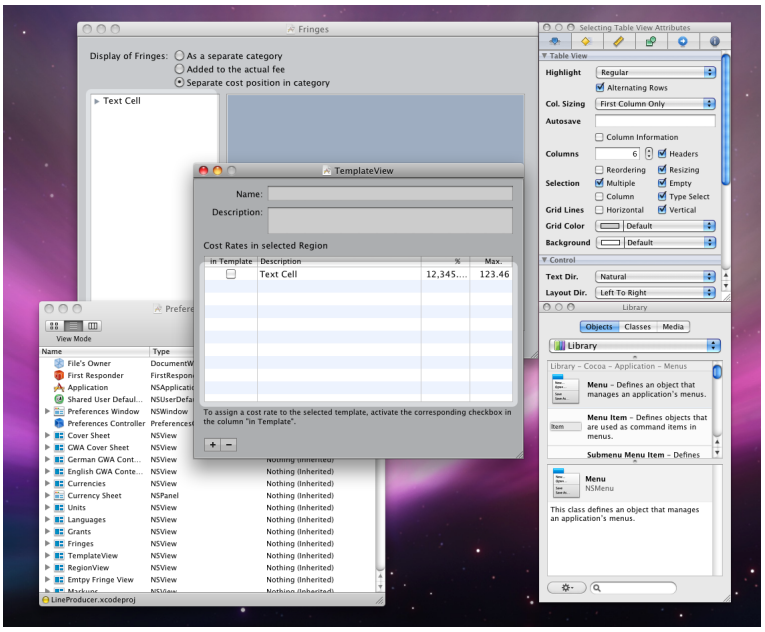


Figure 3.14: Interface Builder

3.2.5 Testing

Delivering reliable software requires extensive and coordinated testing. Several test procedures have been established to ensure high software quality. Unit tests are performed automatically, manual testing by an external tester and usability tests with external test subjects by the development team.

Unit Testing

During later phases of the project, refactoring became more and more important as existing code had to be changed. To guarantee sustainability and increase quality of the software, a unit test system was introduced.

To secure the most critical parts with unit tests first, the model was chosen to be covered with unit tests. From this point on every newly introduced model feature was developed in a test driven development approach. Also changes necessary due to bug fixes were covered with unit tests before the actual implementation.

The automated unit testing framework OCUnit, a testing framework included in Xcode, was used to automate the execution of unit tests. For every model class, a distinct test class was written covering all nontrivial functionality. Especially financial calculations were validated using precomputed results for comparison. (Figure 3.15)

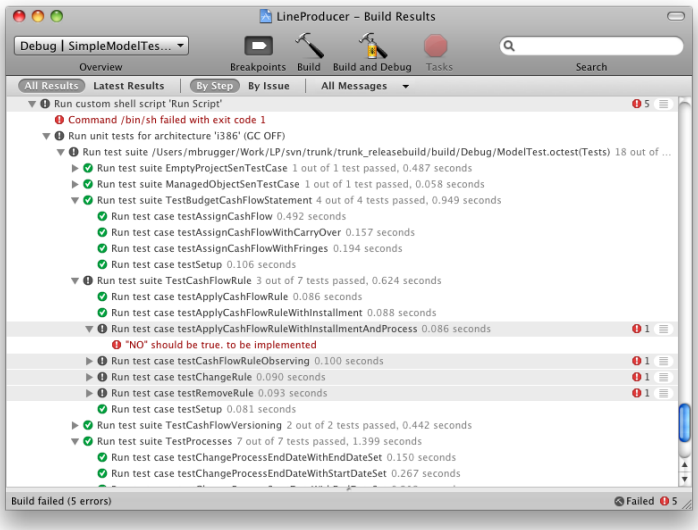


Figure 3.15: Running Unit Tests with OCUnit

Manual Testing

Manual testing was, as mentioned before, performed by an external tester. The test procedure was organized using a test management tool named TestLink[5]. Within the system, test cases for manual testing can be defined and grouped by modules. For every release, a test plan is generated and executed manually. Each step in the test plan is documented and a result of the execution stored. After executing all test cases the automatically created test protocol is delivered back to the developers to be analyzed for critical bugs to be fixed before delivery. (Figure 3.16)

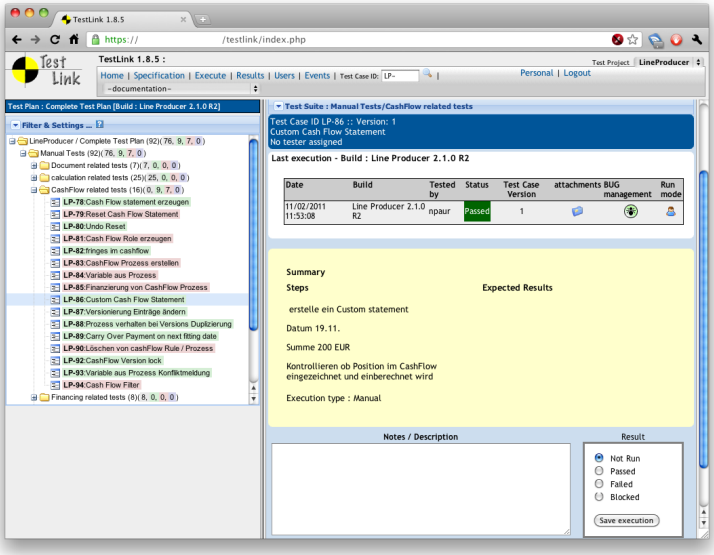


Figure 3.16: Testlink

3.3 Software Architecture

LineProducer is based on the Cocoa framework and adheres to its design guidelines and patterns. As the Cocoa framework itself relies on the model-view-controller pattern (Holzinger et al., 2010) also the whole application architecture is based on it to match design guidelines and integrate with the existing framework.

The application is organized as document based application using the default document architecture. (Apple, 2010)

Document based applications allow the management of multiple documents within

[5] `http://www.teamst.org`

one application instance. This is the typical application behavior for content creating applications. For better readability only simplified schematics of the class model are presented.

As a convention all Cocoa framework classes are prefixed with NS. Custom LineProducer classes also follow this specification by using a LP prefix.

The main runloop for each Cocoa application is implemented in the NSApplication class which is instantiated and executed in the main function of every Cocoa application and responsible for creating the basic infrastructure of the application. As subclassing NSApplication is highly unrecommended, a delegate handles all application global custom behavior.

Within LineProducer several application global functionality is located at this level.

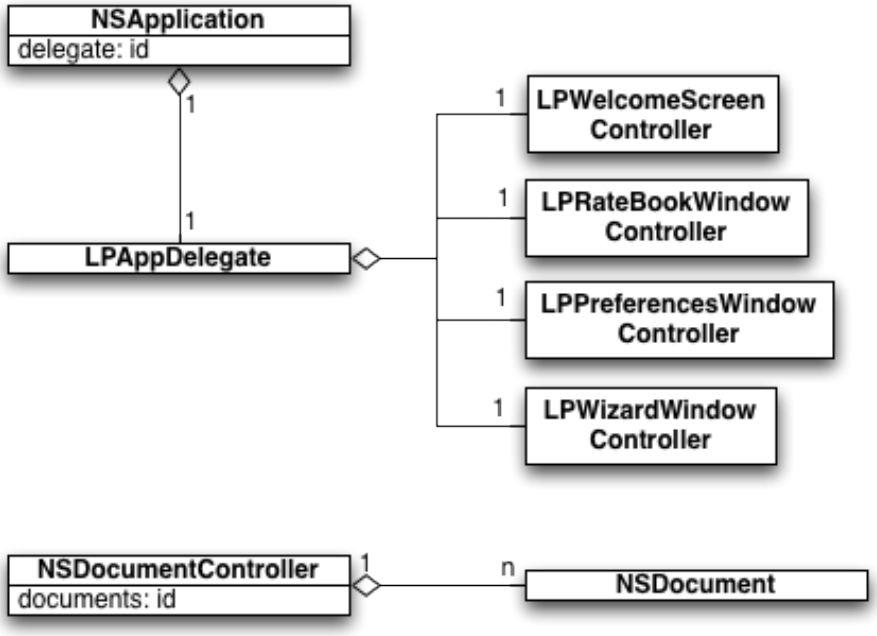


Figure 3.17: LineProducer Application Class Diagram

(see Figure 3.17)

- *LPPreferencesWindowController*: This controller is responsible for application preferences. It uses the NSUserDefaults system storing application preferences
- *LPRatebookWindowController*: As ratebook data is not connected to documents and available at any times this class manages the ratebook. The rate-

book contains default rates for often used items in a typical film budget like typical wages and equipment prices.

- *LPWizardWindowController*: When starting LineProducer without a project, a wizard is presented to allow the easy creation of a LineProducer Project. At the end of the wizard a LineProducer document with customized settings from the wizard is instantiated.

- *LPWelcomeScreenController*: At the top level, licensing is an important concern. Each user has to obtain a valid license and register his copy of LineProducer. This is obviously another application- and not document-related operation.

Within the document based application a document controller is responsible for creating, loading, saving and closing of documents. Each document represents a LineProducer project and is connected to multiple window controllers for the specific tasks within a LineProducer project. A document typically has a custom window controller managing the document window. If necessary additional windowcontrollers are used to manage secondary windows related to a document. (see Figure 3.18)

In the case of LineProducer several secondary windows for managing document

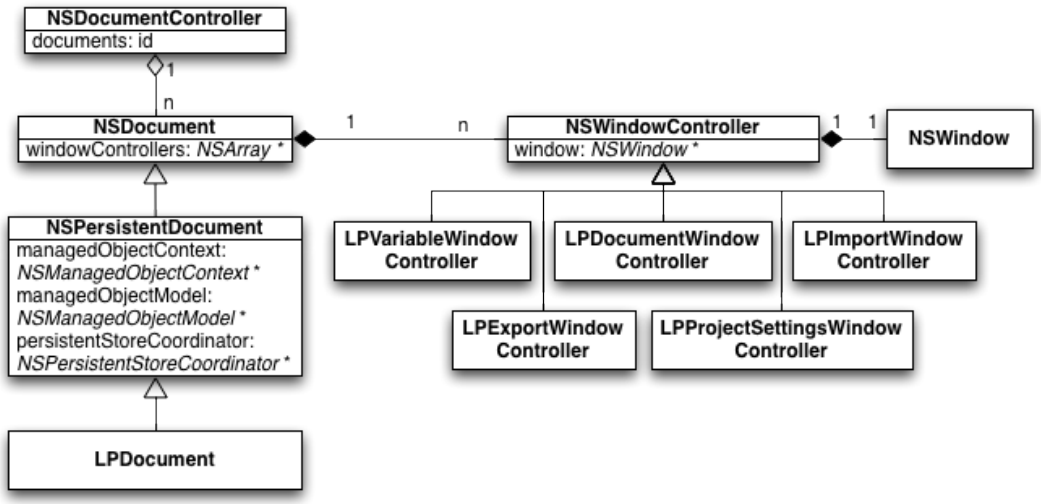


Figure 3.18: LineProducer Document Class Diagram

data exist.

- *LPVariablesWindowController*: The variable system defines values to be used across the whole project. Therefore this is the central location to manage these values.
- *LPVersioningWindowController*: As each LineProducer project can be maintained in multiple versions, this controller is responsible for creating and deleting multiple project versions.
- *LPExportWindowController*: Although this controller does not manage a lot of user interfaces, it is responsible for exporting the project data to other formats like Microsoft Excel and formatted pdf documents.
- *LPProjectSettingsWindowController*: Every LineProducer project has custom settings. This controller manages all the information related to a document.
- *LPDocumentWindowController*: Finally this is the core of every LineProducer project. The DocumentWindowController manages all functionality of a document. Therefore it is also responsible for the main window of a document.

The document is also responsible for creating the underlying datamodel and connecting it to the different window controllers. Core Data is used for managing and persisting project related information. (see Figure 3.19)

A subclass of NSDocument called NSPersistentDocument already implements the

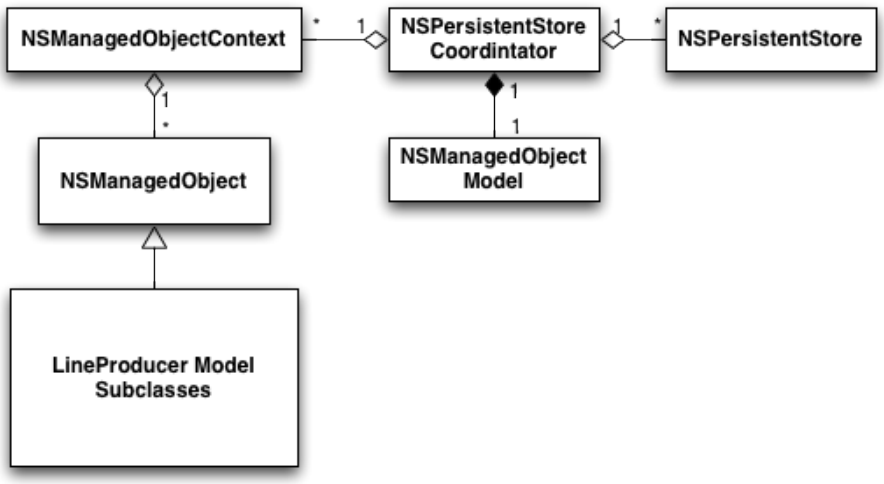


Figure 3.19: LineProducer Model Class Diagram

main components of a typical Core Data infrastructure.

- *NSManagedObjectModel*: The Core Data model describes the datastructure used for storing all LineProducer project related information. It is created using a graphical modeling tool described in Section 3.2.4.

- *NSManagedObjectContext*: The context is a scratchpad for handling data objects. Therefore it is responsible for adding and removing objects. It also manages referential integrity and on demand loading of objects.

- *NSPersistentStoreCoordinator*: This class is responsible for persisting all objects present in the context. Therefore one or more persistent stores are added to the coordinator. Each store can be written to a file in the filesystem.

- *NSManagedObject*: All custom model functionality is implemented in classes inherited from NSManagedObject. Each managed object represents a single aspect of a LineProducer document. The main components of a document can be separated into Navigation objects, responsible for the rough structure of a document, budget and financing related objects.

The user interface in Cocoa applications is composed of NSView subclasses. LineProducer is mainly based on data organized in table and outline views. Other frequently used components include NSTextView and NSButtons for user interactions. Besides these standard classes, a vast amount of LineProducer specific subclasses have been implemented to customize the view behavior and present information to the user.

4. Results

Within this chapter the resulting software project and also a review of used SDM and UE methods are presented. From a technical point of view the software is finished and user feedback is gathered for further development. As the software development has been finished the development process has proven to create a usable output and therefore can be seen as successful.

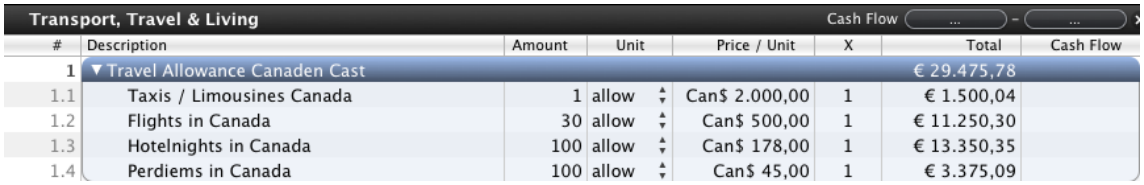
4.1 Developed Software Product

The developed software is already in use by several film production companies in Germany. Several usability techniques have been used to improve the user experience. Most of the requirements are covered by the product. Some tradeoffs in terms of features had to be made and postponed for later development. The phase of software maintenance has been reached and currently only minor changes and bugfixes are made.

4.1.1 Budgeting

A typical film project requires the capture of all cost relevant positions. The highest level of each Budget consist of cost categories already described in section 2.1.1. Within the cost categories exists a hierarchical structure of cost groups and individual cost positions. This hierarchy allows creating a individual cost structure. Cost positions typically consist of a description, amount and price per unit. Storing a currency with each position is an essential feature for international film projects. Within larger film production companies the numbering of cost positions allows linkage of cost position between different systems. Data exchange between accounting

and budget planning is performed on the basis of this numbering.



Transport, Travel & Living — Cash Flow

#	Description	Amount	Unit	Price / Unit	X	Total	Cash Flow
1	▼ Travel Allowance Canaden Cast					€ 29.475,78	
1.1	Taxis / Limousines Canada	1	allow	Can$ 2.000,00	1	€ 1.500,04	
1.2	Flights in Canada	30	allow	Can$ 500,00	1	€ 11.250,30	
1.3	Hotelnights in Canada	100	allow	Can$ 178,00	1	€ 13.350,35	
1.4	Perdiems in Canada	100	allow	Can$ 45,00	1	€ 3.375,09	

Figure 4.1: Cost positions in LineProducer

Besides the hierarchical structuring of cost positions separating all costs by its type, additional information is often needed about groups of costs across the hierarchy. Therefore a tagging system has been included, allowing quick access to information about distinct modules of the film production not fitting the standard cost structure. An example would be to gather all costs related to a shoot on a specific location. This includes cost positions spread across categories like transport, locations, cast, art department and many more. Tagging has proven to be a tool easy to use for categorizing information without a controlled vocabulary. Marlow et al. (2006) This allows a great amount of flexibility without a negative impact on usability.

Adding meta information to cost positions allows querying for certain attributes. The combination of such properties creates a powerful tool for controlling. With this additional information it is possible to identify inconsistencies in the planned budget. It is for example possible to find cost positions for staff on a certain location with missing accommodation. Another possibility is to identify exceptionally expensive travels. As a production manager this reveals numerous potential savings. (Figure 4.2)

Filtering hierarchical data for certain attributes was achieved by hiding not matching individual rows and recalculating visible costs. Cost groups therefore also are hidden if no matching entry is contained. The quick filtering area on screen allows the temporary application of filters to a project. Three different settings are possible. "And" and "or" filtering is applied with its natural meaning. "Not" filters allow a combination of multiple criteria A and B resulting in a NOT (A AND B) combination of filter elements.

Another requirement for a Budget is to be easily modifiable. Costs are often time dependent, resulting in groups of cost positions being related in terms of time

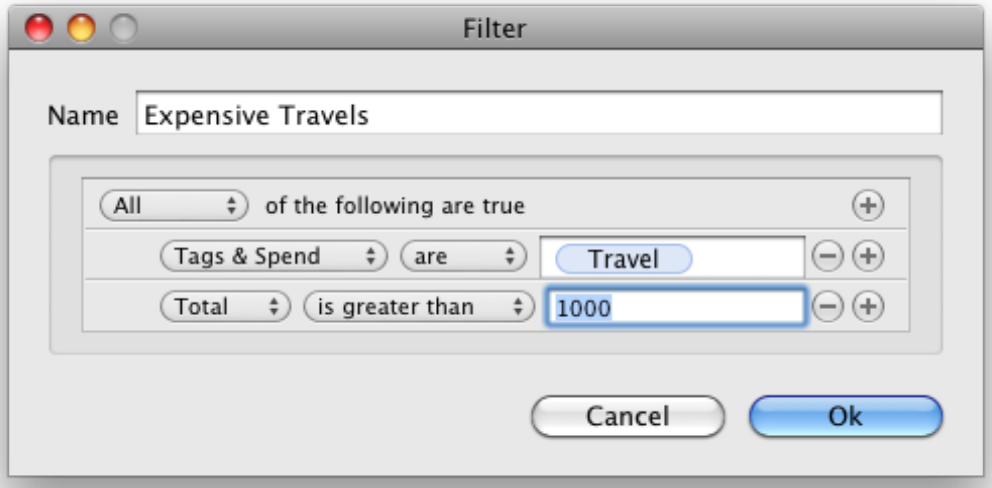


Figure 4.2: Combined Queries in LineProducer

but not cost height. Given the example before, the length of a shoot at a certain location influences several cost positions. Location fees, wages and rentals depend on the amount of shooting days. Software developers are familiar with variables, still the target group does have to learn how to abstract the usage of them, resulting in an increase of flexibility and time savings. (Figure 4.3)

During the production of a film, several budgets are created. An essential feature is the maintenance and comparison of different budgets. The prototyping phase identified several key concepts for versioning and comparison. The numbering and hierarchy delivers a comparison attribute reasonable to the user. Using more sophisticated attributes for comparison resulted in easier maintenance of budgets, but in some cases created misleading comparison information to the user resulting in distrust to the software. Visualizing comparison results is another difficult task. The amount of information visible on screen displaying two budgets side by side creates an information overload. Therefore essential information for display has to be identified. (Figure 4.4)

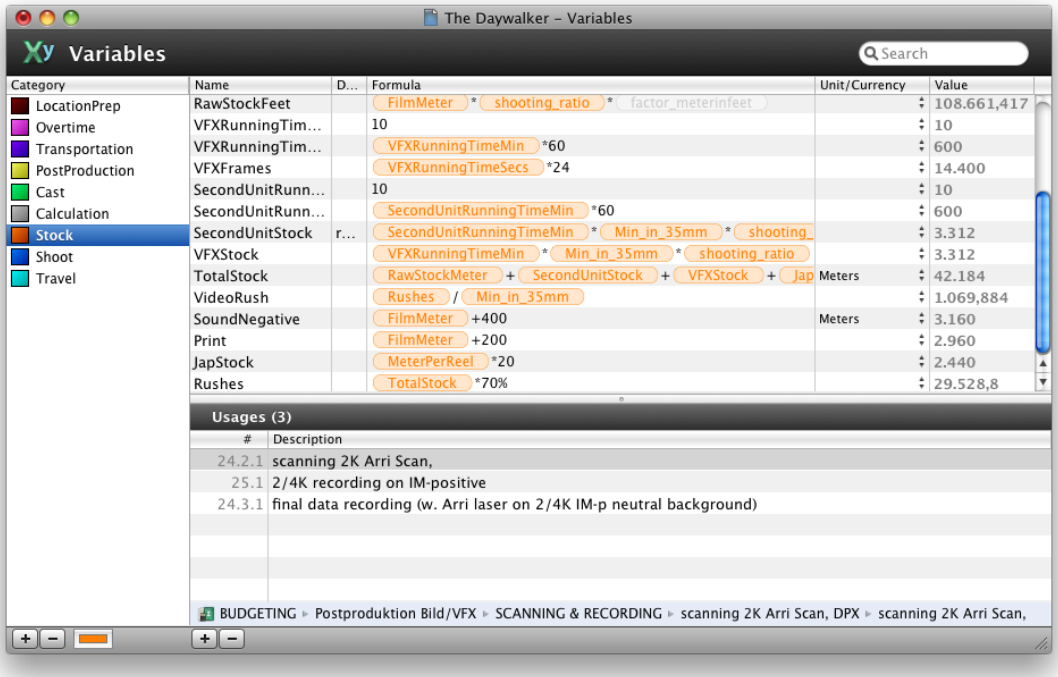


Figure 4.3: Variables in LineProducer

Figure 4.4: Budget Comparison in LineProducer

4.1.2 Financing

Financing within a film project can have a complex structure. Several influencing factors have to be taken into account. Simple budgets only have one producer with a fixed budget and internal financing. If several coproducers are involved, there is a need to exactly represent the actual financing situation. This combination of different sources of funding is the basis for recoupment negotiations. Therefore several key figures need to be calculated and traceable at all times. Costs and funding are directly related. For every cost position there needs to be a responsible person. In co-production situations, there is a need to strictly separate this responsibilities. LineProducer therefore allows the assignment of cost positions to different coproducers creating clarified matters. In return, the cost responsibility can easily be measured and taken into account for negotiations.

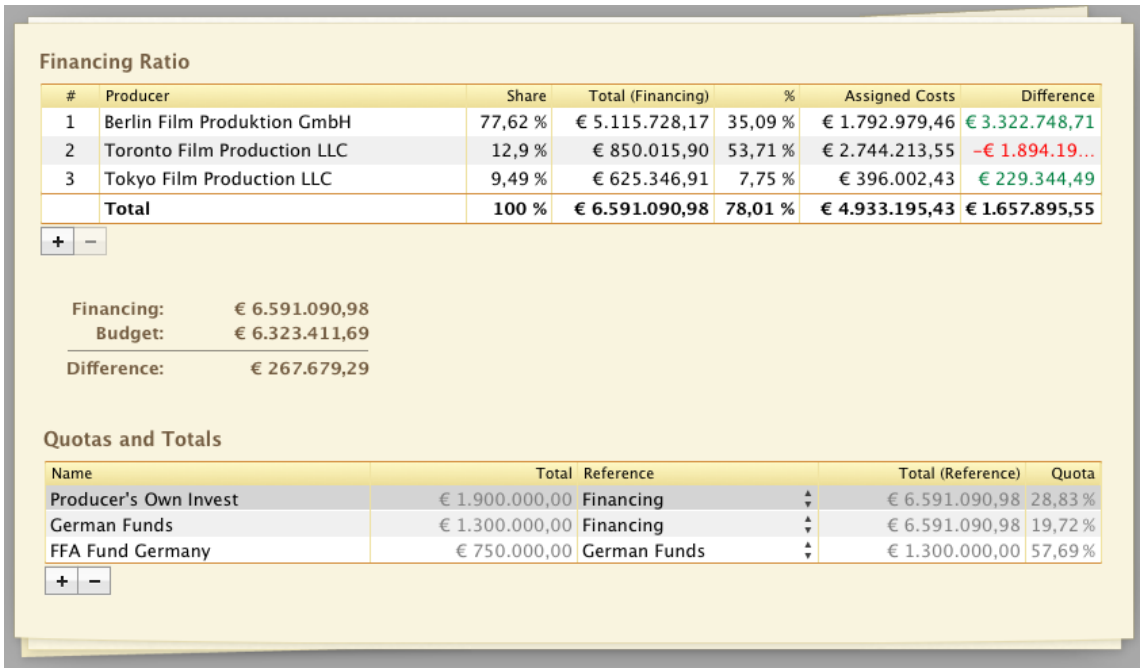


Figure 4.5: Financing Key Figures in LineProducer

A typical financing position contains a description, an amount and several properties for the type of financing. (Figure 4.6) These types are all user customizable and represented on the top sheet. Typical types of financing positions are internal versus external or subsidized funding.
Those properties are essential to a film project and need to be within certain limits

not to risk the whole project.

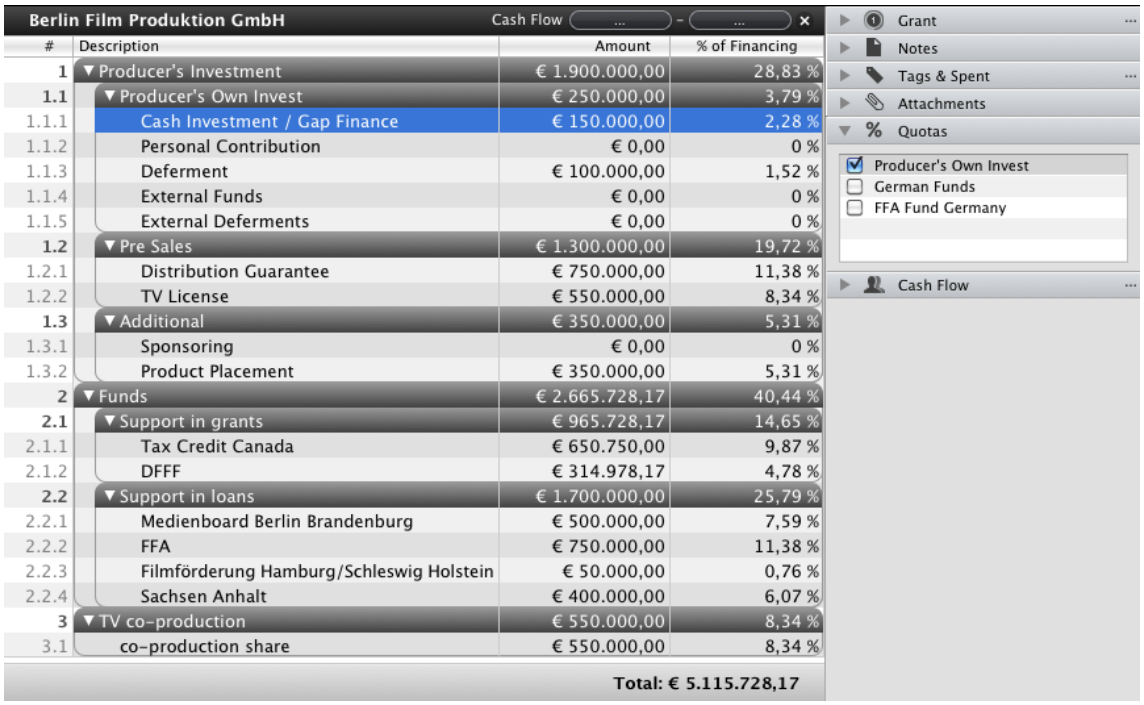


Figure 4.6: Financing in LineProducer

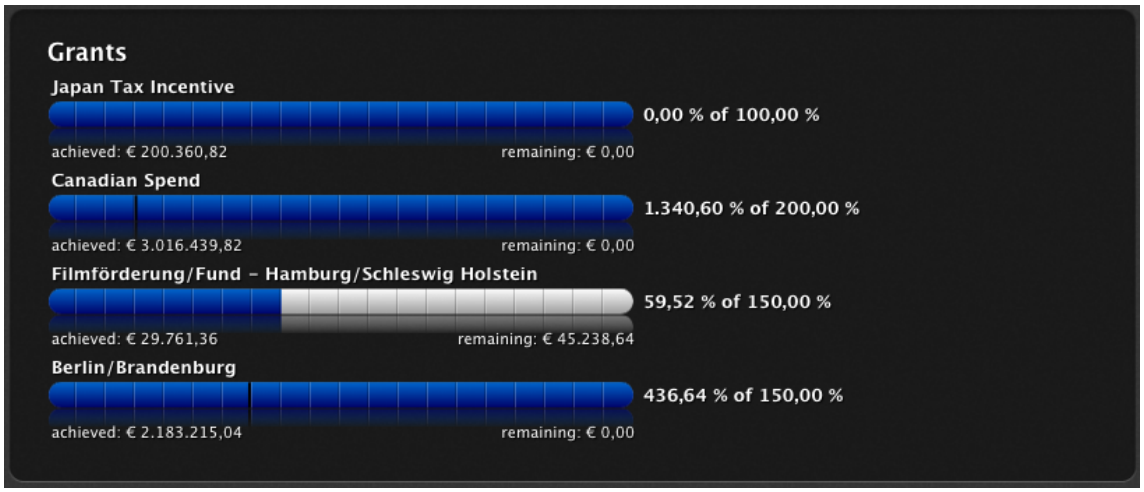


Figure 4.7: Grants in LineProducer

Another aspect of film financing is the gathering of subsidies. In Europe, several institutions distribute subsidies for film projects. To receive subsidies several aspects have to be taken into account while planning a budget. Receiving a certain amount of subsidy requires to generate costs of a significant amount higher than the subsidy.

Typically, there is a percentage defined for a subsidy which has to be compensated for in costs within a certain region. Therefore cost positions can be assigned to different subsidies. The current state of subsides can be displayed in the project overview. (Figure 4.7)

4.1.3 Cash Flow

The inclusion of costing and financing within a single project in combination with time information allows the creation of cash flow plans. Section 2.1.1 already mentions the importance of cash flow planning within a film project. As there is no standardized form of cash flow planning, a self defined system has been established. Every position in budgeting and financing can be connected to time information. Within the cash flow overview a cash flow plan is calculated. To meet the most common requirements for recurring payments, a distribution of costs over time is essential. This requires the definition of a cash flow period and some sort of payment rule. (Figure 4.8)

For financing positions a fragmentation of payments is often required. Each position can be separated into several user defined parts. Changing the height of a position is correctly reflected in cash flow through recalculation of cash flow relevant sums with the same ratios as originally defined. During development several scenarios have been discussed with fixed sum shares but automatic recalculation has been decided to be the most important aspect allowing quick changes to the budget without creating inconsistencies. (Figure 4.9)

The cash flow plan itself is represented in a table with user selectable granularity of weeks and months. During the critical shooting phase where cash flow is far higher than in the other production phases, a weekly cash flow plan is essential. Additionally a graphical representation is also available to give a general overview of the current projects state. (Figure 4.10)

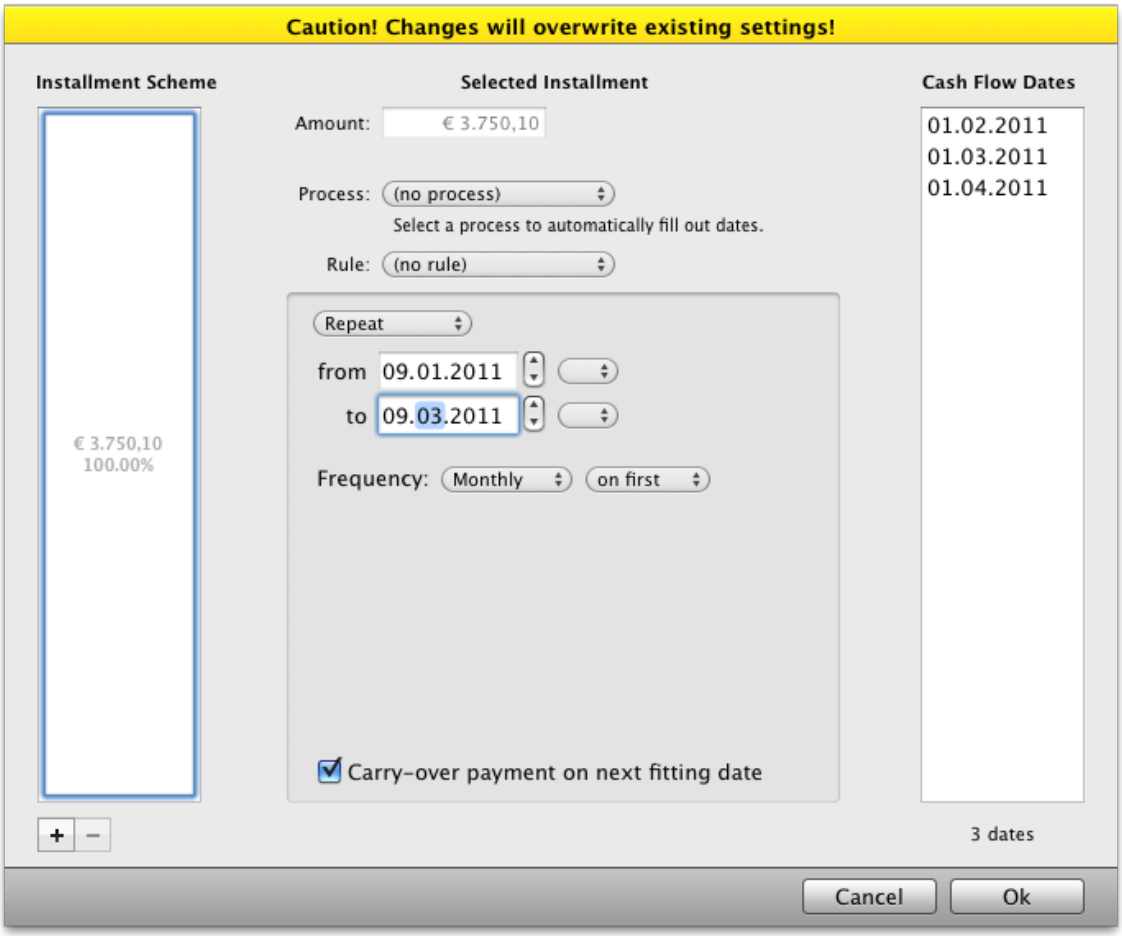


Figure 4.8: Cash Flow Rules in LineProducer

Figure 4.10: Cash Flow Overview in LineProducer

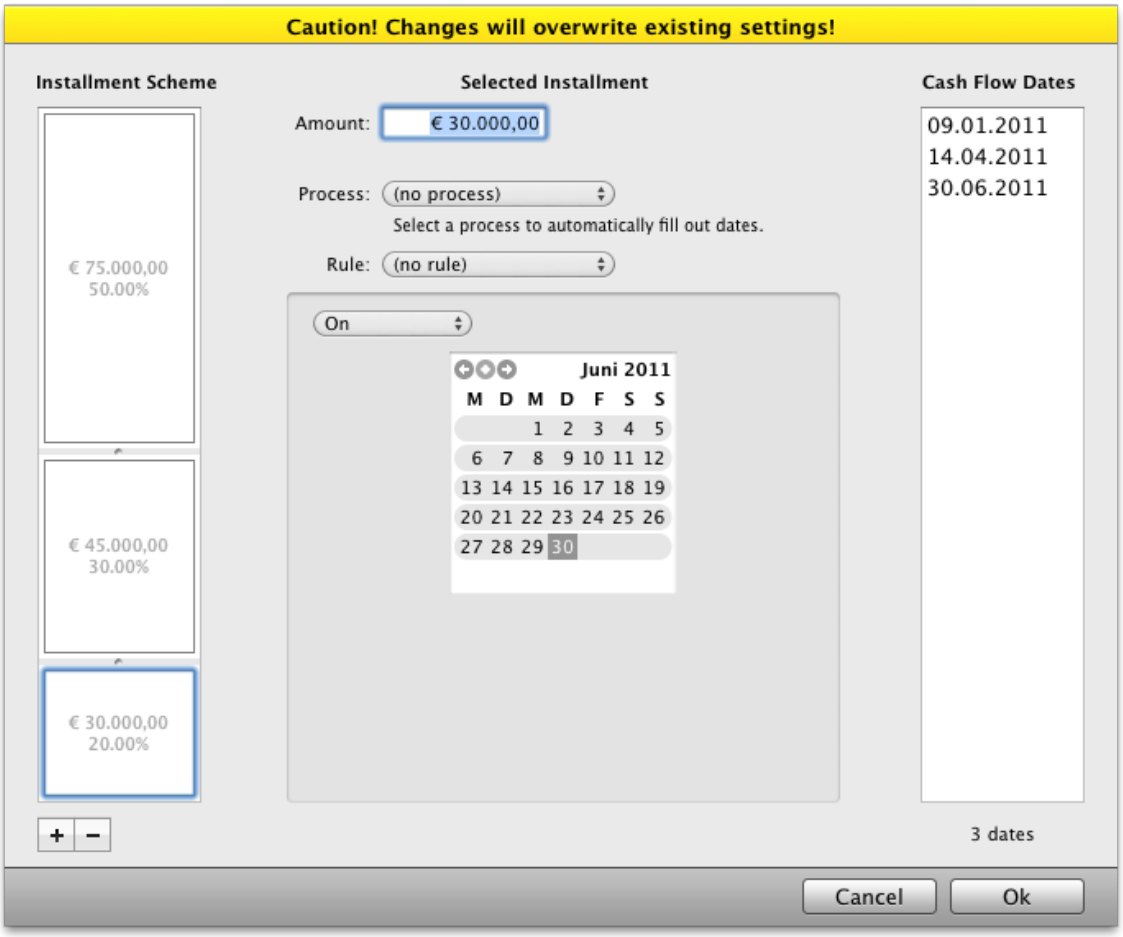


Figure 4.9: Cash Flow Fragmentation in LineProducer

Cash flow bottlenecks can easily be identified and appropriate measures can be taken.

4.2 Established Development Process

With the project team size growing a need for coordination appeared. First a simple task and bug tracking system was established to keep track of work to be done. Within this phase, also coordination became more important for development. Therefore common agile software development methodologies were evaluated for their usage within the current team. Also usability engineering shaped up as vital for this project.

4.2.1 Software Development

Crystal Clear was an interesting approach. Leading a project team with this background information in mind worked for this project, still revealed shortcomings of a freely changeable development methodology.
The colocation of the development team was very well accepted by the team. The close communication within the team assisted the development process.
Short iterations for frequent delivery resulted in a streamlined testing and deployment process.
The average iteration length was about four weeks after introducing Crystal Clear in the second iteration. Figure 4.11 shows the duration lengths of the first 12 iterations which lasted 388 days in total. Iterations tended to last significantly longer ahead of major releases. Iteration eight resulted in the first private beta test with film students and iteration twelve in the first public release.

Using the information of the ticketing system described in Section 3.2.3 perfor-

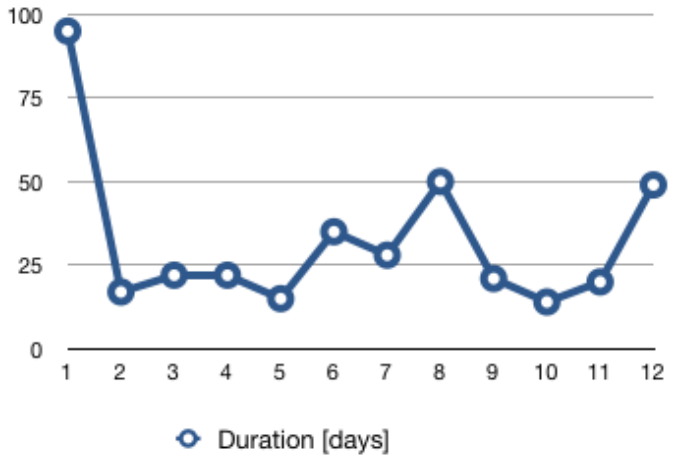


Figure 4.11: Iteration Duration

mance in terms of tickets resolved per day within an iteration can be calculated. (Figure 4.12)
In total, 806 tickets were closed during the development period.

Within the ticketing system, each ticket was classified as bug or task. The average ratio of task tickets was about 30 percent and ranged from 10 up to 54 percent. A high ratio identified iterations with focus on new feature development. (Figure 4.13)
Also internally detected bugs were added to the ticketing system, therefore this does

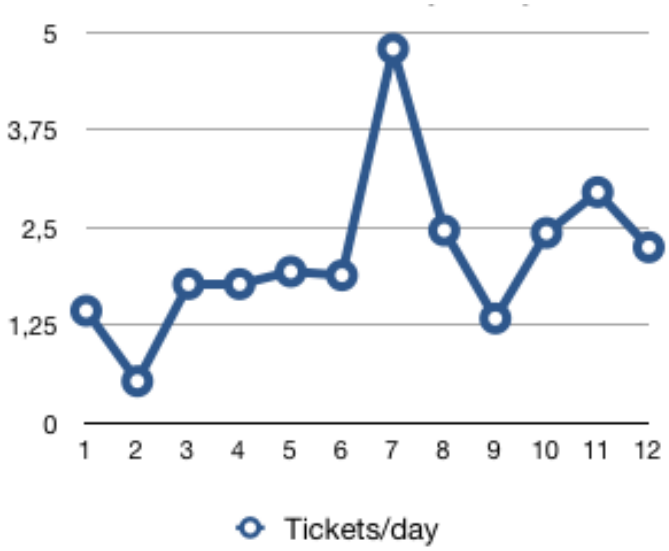


Figure 4.12: Average Tickets Resolved per Day

not reflect the amount of errors in released software versions.

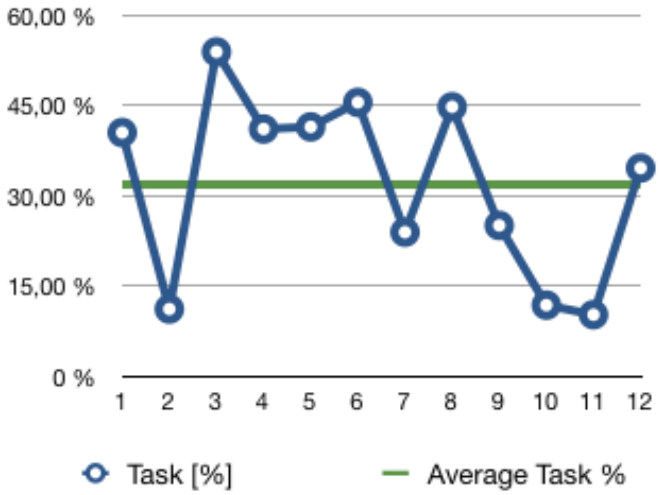


Figure 4.13: Task Ratio of Total Tickets

Using the source control management system described in Section 3.2.2 additional statistics can be created. The average commit rate per day was 13.36 and varied from 10 to a maximum of 20 during the iterations. (Figure 4.14)

Another interesting figure is the growth of the codebase. Starting with initially 17k lines of code, after the twelfth iteration a total amount of 58k lines of code was reached. (Figure 4.15)

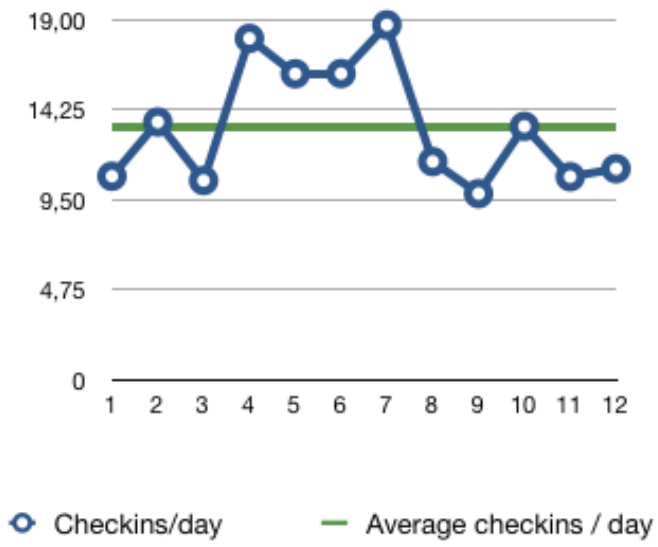


Figure 4.14: SCM Commits per Day

This statistic was generated using the tool CLOC[1]. CLOC implements an algorithm to count physical lines of code removing comments first. Therefore the actual count also depends on code formatting in opposite to logical lines of code. Still these metrics give an impression of the total amount of code written for the current project.

Using this information the code increase per iteration can also be visualized. Figure 4.16 shows a decrease in new lines of code over time. Also the holiday season

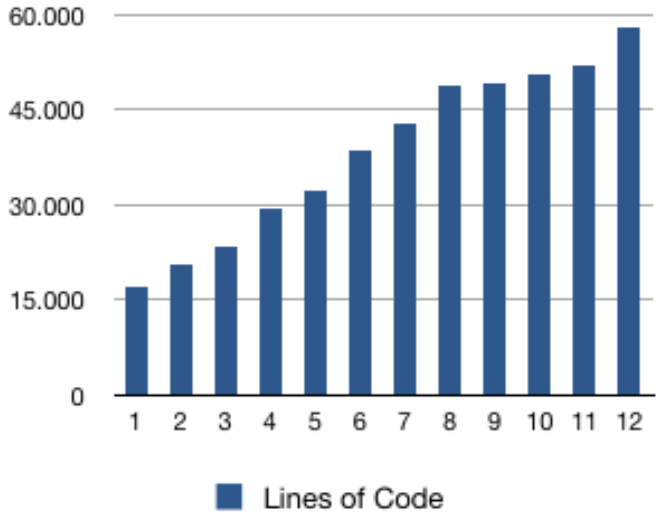


Figure 4.15: Lines of Code in LineProducer

[1] `http://cloc.sourceforge.net/`

in iteration nine can be clearly identified in this graph as the written lines of code drastically decreased in this period.

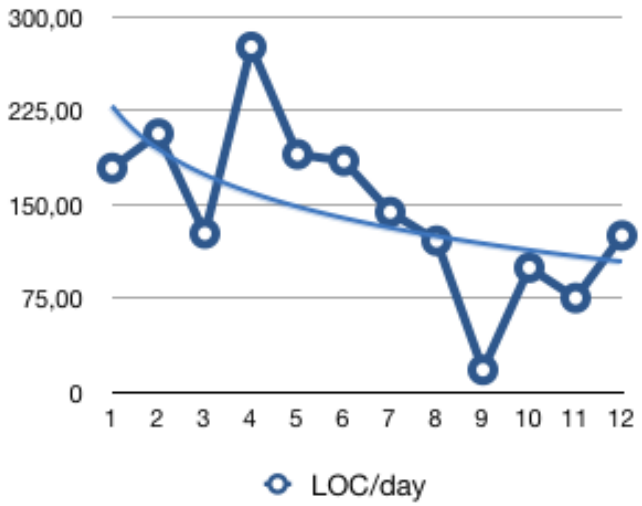


Figure 4.16: Lines of Code Written per Day

Within the MVC Software Design especially the model layer was covered with unit tests.
Unit testing is essential for frequent deliveries to maintain high quality. Still from a developers perspective unit testing is often perceived as an annoying task. Therefore the test system has to be as easy to use as possible. Unreasonable impediments must be eliminated to support the usage of a unit test system or the test coverage will decrease drastically. Some information about the current test coverage is displayed in Table 4.1.

	Model	Controller	Total
Relevant classes	42	7	49
Test classes	30	7	37
Tests	157	22	179
Tests/Test class	5.23	3.14	4.86

Table 4.1: Test Coverage

Using the lines of code statistics also the amount of test code written can be measured. With a refactoring of the folder structure of the project in iteration four

the separate tracking of the model layer was easily possible. Developing unit tests started in iteration six resulting in a basic test coverage of the model layer. (Figure 4.17).

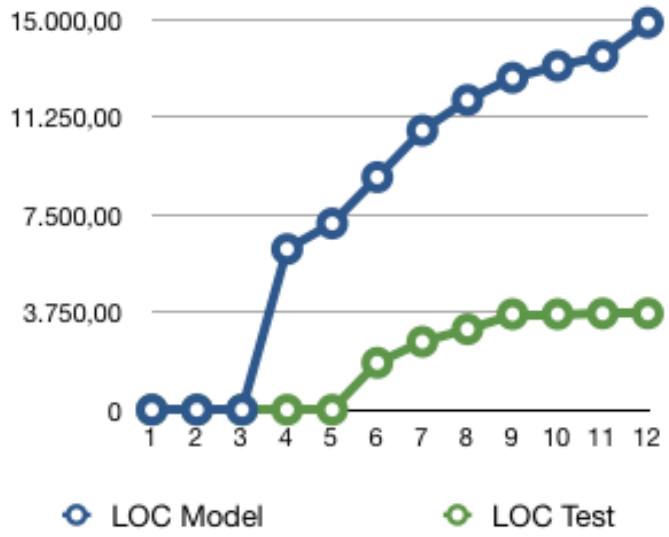


Figure 4.17: Lines of Code for Testing

Reflective Improvement was not explicitly performed still within regular meetings possible improvements were discussed. The main three properties of crystal clear can be assumed established and the development process finished.

4.2.2 Usability Engineering

With increasing application complexity the usability aspect became more important.

A thinking aloud test was performed during development before final release and discovered several issues developers would not have thought of. Still the time consuming process is hardly suitable for frequent releases. Only new components should be tested thoroughly with end users.

During regular iterations heuristic evaluations became the methodology of choice. Several paper mockups were produced by each team member and evaluated as a group.

To increase the paper mockup quality, templates with standard user interface elements were created and used.

5. Discussion & Conclusion

Several important aspects of software development were discussed and their application presented. Some of them have proven to be of exceptional usefulness, others failed for one or the other reason. Still the project as a whole can be considered successful and the development team evolved to a new level of professionalism. In general the awareness for usability and also development process has been enhanced.

5.1 Software Development Methodologies

Establishing a defined software development process was obligatory, choosing a rather soft transition into agile software development with Crystal Clear, probably lowered barriers against agile development methodologies but was not successfully established as a process.
The different aspects of Crystal Clear have proven its usefulness.

- **Frequent Delivery** can be considered established, all relevant processes for distribution were automated to create a one click distribution solution. Automated testing was introduced at a later stage, saving critical parts of the project with heavy interdependencies from being not maintainable at all. Also a certain level of code quality was introduced by securing the code foundation of the project. With all these tools in place, the feature of frequent delivery was used to gather feedback information from several testers, also being able to respond to bugreports within reasonable time.

- **Osmotic communication** was well accepted by the team. Required by space constraints at the project start, in later stages the team preferred to stay within close communication ranges, even if it was not a general requirement

after project finish. At some point, the constant background noise tended to influence some developers in a negative way, but the team managed to balance this problem by establishing silent phases for undisturbed productivity.

- **Reflective Improvement** was unfortunately given too little attention from the beginning. As an open communication existed among the team the lack of explicit reflection did not have large negative impact. Still the introduction of regular retrospectives revealed interesting improvements within the team and the development process as well.

5.2 Usability Engineering

The awareness among the team for usability engineering existed from the beginning. Scientific usability tests were initially planned and carried out with students and revealed to be of less use with domain experts. With the relationship of developers being dependent on input and time of these experts, it was hard to follow strict testing protocols. Nevertheless the information was of great use in terms of requirements engineering. Heuristic evaluations were a matter of course, as creating good and usable interfaces for complex problems were a primary concern of all team members.

5.3 Code Reuse

Code reuse and licensing were a relevant topic for this project, as several third party code was integrated into the current software product. Especially the problem of reuse of public available code without explicit license was often neglected by developers, creating possible legal threats for the project. With minimum guidelines for code reuse in place, all team members perceived the possible risks and carefully selected code fragments. In general the development time was drastically reduced by reusing third party code and also a solid foundation of own code ready for reuse was established. Not all third party code proved to be of the expected quality and often had to be fixed by the team. Reporting bugs and submitting patches to the original authors was self-evident, but not always appreciated. Some of the reported

bugs have not been fixed until today.

5.4 Developed Software Product

With the developed product already finished for a first release, some topics on the product itself could be discussed. Functionality is still a key to user satisfaction in our target group. Only if all basic requirements are met, they can benefit from all the additional possibilities the software offers to them. Our first customers very much appreciated the attention we gave them, eliciting feedback for further improvement. Nevertheless feedback had to be carefully filtered for its general applicability by potential customers. The target group in general seems to be more fragmented than expected. Domain experts we expected to deliver valuable general applicable knowledge exposed to have a very self centered view on the topic of film production with their projects in mind. Therefore some essential features are currently missing. As the software is just a part of a larger workflow, the subject of interfaces was underestimated. With a first large company as customer, interfaces were developed to communicate with existing systems. User satisfaction was very high. Customers using LineProducer emphasized the ease of use and especially time savings due to the multiple options for filtering and exporting partial budgets. In general the product seems to fulfill the basic requirements and expectations of the target group, still quite some effort is necessary to integrate the software into the film production process in a better way.

6. Future Work

As software almost never can be considered finished, work on this project will also continue. Basic tasks of future work are maintaining the current system, extending its functionality for future releases and putting some effort into research on new possibilities for innovation.

Maintenance for existing users of the software is a key factor to success, gathering feedback for improvement and increasing customer satisfaction.
Treating customer feedback in a way the customer feels taken care of, increases the products reputation and also quality. With a software product being in productive usage, newly discovered bugs need to be fixed. Therefore the current feedback system delivers extensive information for debugging if necessary. Also new feature requests need to be recorded and integrated into the future development plan.
Not every feature request has to result in an immediate implementation as most of them are far too specific for the majority of users. Therefore also customer specific implementation is offered as a service to existing customers.

For successful sales a roadmap for future development has been established, delivering new features for keeping an edge over competitors.

Also internal development work has to be done, several parts of the software need refactoring simultaneously, extending the test coverage on existing code. Maintaining a release version requires extensive testing, not to break any current functionality. Automating the test process is an interesting approach for cutting costs in this phase of the product lifecycle. Several technologies exist for automated user interface testing, requiring quite some effort in introducing such a system at a useful level.

There has already been put some effort into research for a more complete support of the film production process by supporting the script breakdown phase.
A diploma thesis has been authored, covering this part of the film production process, delivering interesting results. Semi automatic support in classification of text elements can drastically speed up this process and increase its quality.
Also the scheduling task within the film production process could be improved, by integrating academic knowledge into the film production process. From a usability perspective several parts of the interface could need testing to prove their usefulness. With modern remote usability testing methods the end user could be directly involved into this process.
The software itself and its current design strongly encourage the integration of such methods to gather valuable usage information for future improvements.

A. Software Metric Tables

Iteration	SVN Revision	Iteration Duration	Revisions/Iteration	Revisions/day
1	1019	95	1019	10,73
2	1250	17	231	13,59
3	1481	22	231	10,50
4	1877	22	396	18,00
5	2119	15	242	16,13
6	2684	35	565	16,14
7	3208	28	524	18,71
8	3783	50	575	11,50
9	3989	21	206	9,81
10	4176	14	187	13,36
11	4390	20	214	10,70
12	4935	49	545	11,12

Table A.1: LineProducer Iterations and Subversion Revisions

Iteration	Bugs/Iteration	Tasks/It.	Tasks/Tickets	Tickets/It.	Tickets/Day
1	81	55	0,40	136	1,43
2	8	1	0,11	9	0,53
3	18	21	0,54	39	1,77
4	23	16	0,41	39	1,77
5	17	12	0,41	29	1,93
6	36	30	0,45	66	1,89
7	102	32	0,24	134	4,79
8	68	55	0,45	123	2,46
9	21	7	0,25	28	1,33
10	30	4	0,12	34	2,43
11	53	6	0,10	59	2,95
12	72	38	0,35	110	2,24

Table A.2: LineProducer Iterations and Tasks

Iteration	LOC	LOC/It.	LOC/Day	Files	Files/It.	Files/Day
1	17.000	17.000	178,95	197	197	2,07
2	20.509	3.509	206,41	243	46	2,71
3	23.294	2.785	126,59	275	32	1,45
4	29.359	6.065	275,68	326	51	2,32
5	32.207	2.848	189,87	339	13	0,87
6	38.667	6.460	184,57	394	55	1,57
7	42.692	4.025	143,75	425	31	1,11
8	48.732	6.040	120,80	482	57	1,14
9	49.102	370	17,62	484	2	0,10
10	50.492	1.390	99,29	502	18	1,29
11	51.993	1.501	75,05	514	12	0,60
12	58.116	6.123	124,96	570	56	1,14

Table A.3: LineProducer Lines of Code and Files

Iteration	LOC Model	LOC Test
1	0	0
2	0	0
3	0	0
4	6.172	0
5	7.158	0
6	8.935	1.804
7	10.733	2.616
8	11.899	3.094
9	12.754	3.673
10	13.206	3.673
11	13.588	3.700
12	14.868	3.702

Table A.4: LineProducer Lines of Code for Testing

List of Figures

List of Tables

References

Adair, John G. [1984]. *The Hawthorne effect: a reconsideration of the methodological artifact. Journal of Applied Psychology*, 69, pages 334 – 335.

Apple [2010]. *Document-Based Applications Overview.* `http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/Documents/Documents.html`.

Bates, Christopher D. and Simeon Yates [2008]. *Scrum down: a software engineer and a sociologist explore the implementation of an agile method.* In *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 13–16. ACM, New York, NY, USA. ISBN 978-1-60558-039-5. `http://doi.acm.org/10.1145/1370114.1370118`.

Beck, Kent [1999]. *Embracing Change with Extreme Programming. Computer*, 32(10), pages 70–77. ISSN 0018-9162. `http://dx.doi.org/10.1109/2.796139`.

Beck, Kent [2002]. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0321146530.

Bell, T. E. and T. A. Thayer [1976]. *Software requirements: Are they really a problem?* In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 61–68. IEEE Computer Society Press, Los Alamitos, CA, USA. `http://portal.acm.org/ft_gateway.cfm?id=807650`.

Bertolino, A. [2007]. *Software Testing Research: Achievements, Challenges, Dreams.* In *Future of Software Engineering, 2007. FOSE '07*, pages 85 –103. `10.1109/FOSE.2007.25`.

Boehm, B [1986]. *A spiral model of software development and enhancement. SIGSOFT Softw. Eng. Notes*, 11(4), pages 14–24. ISSN 0163-5948. http://doi.acm.org/10.1145/12944.12948.

Clevé, Bastian [2005]. *Film Production Management. Third Edition.* Focal Press.

Cockburn, Alistair [2004]. *Crystal clear a human-powered methodology for small teams.* Addison-Wesley Professional. ISBN 0201699478. http://portal.acm.org/citation.cfm?id=1406822.

Di Penta, Massimiliano, Daniel M. German, Yann-Gaël Guéhéneuc, and Giuliano Antoniol [2010]. *An exploratory study of the evolution of software licensing.* In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 145–154. ICSE '10, ACM, New York, NY, USA. ISBN 978-1-60558-719-6. http://doi.acm.org/10.1145/1806799.1806824.

Errath, Maximilian, Andreas Holzinger, and Wolfgang Slany [2004]. *Agile Softwareentwicklung. OCG Journal*, 29(5), pages 4–6.

Feiner, Johannes, Keith Andrews, and Elmar Krajnc [2010]. *UsabML: formalising the exchange of usability findings.* In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 297–302. EICS '10, ACM, New York, NY, USA. ISBN 978-1-4503-0083-4. doi:http://doi.acm.org/10.1145/1822018.1822065. http://doi.acm.org/10.1145/1822018.1822065.

Fox, David, Jonathan Sillito, and Frank Maurer [2008]. *Agile Methods and User-Centered Design: How These Two Methodologies are Being Successfully Integrated in Industry.* In *AGILE '08: Proceedings of the Agile 2008*, pages 63–72. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3321-6. http://dx.doi.org/10.1109/Agile.2008.78.

Glass, Robert L. [2009]. *A Classification System for Testing, Part 2. IEEE Software*, 26(1), pages 104, 103. ISSN 0740-7459. http://doi.ieeecomputersociety.org/10.1109/MS.2009.1.

Gould, John D. and Clayton Lewis [1983]. *Designing for usability—key principles and what designers think.* In *CHI '83: Proceedings of the SIGCHI conference on*

Human Factors in Computing Systems, pages 50–53. ACM, New York, NY, USA. ISBN 0-89791-121-0. http://doi.acm.org/10.1145/800045.801579.

Holzinger, A., K.H. Struggl, and M. Debevc [2010]. *Applying Model-View-Controller (MVC) in design and development of information systems: An example of smart assistive script breakdown in an e-Business application.* In *e-Business (ICE-B), Proceedings of the 2010 International Conference on*, pages 1 –6.

Holzinger, Andreas [2005]. *Usability engineering methods for software developers. Commun. ACM*, 48(1), pages 71–74. ISSN 0001-0782. http://doi.acm.org/10.1145/1039539.1039541.

Holzinger, Andreas [2006]. *Thinking-aloud - eine Königsmethode im Usability Engineering. OCG Journal*, 31(1), pages 4–5.

Holzinger, Andreas and Stephen Brown [2008]. *Low cost prototyping: part 2, or how to apply the thinking-aloud method efficiently.* In *BCS HCI Conference*, pages 217–218. doi:10.1145/1531826.1531897.

Holzinger, Andreas and Maximilian Errath [2004]. *Extreme usability. OCG Journal*, 29(4), pages 16–18.

Holzinger, Andreas, Maximilian Errath, Gig Searle, Bettina Thurnher, and Wolfgang Slany [2005]. *From Extreme Programming and Usability Engineering to Extreme Usability in Software Engineering Education (XP+UE->XU).* In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, pages 169–172. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2413-3-02. http://dx.doi.org/10.1109/COMPSAC.2005.80.

Holzinger, Andreas, Wolfgang Slany, and Martin Brugger [2011]. *APPLYING ASPECT ORIENTED PROGRAMMING IN USABILITY ENGINEERING PROCESSES On the example of Tracking Usage Information for Remote Usability Testing.* In *ICE-B'11.*

Hussain, Zahid, Martin Lechner, Harald Milchrahm, Sara Shahzad, Wolfgang Slany, Martin Umgeher, and Thomas Vlk [2008a]. *Optimizing Extreme Programming.*

In *ICCCE 2008: Proceedings of the International Conference on Computer and Communication Engineering, Kuala Lumpur, Malaysia*, pages 1052–1056. IEEE. ISBN 978-1-4244-1691-2.

Hussain, Zahid, Martin Lechner, Harald Milchrahm, Sara Shahzad, Wolfgang Slany, Martin Umgeher, and Peter Wolkerstorfer [2008b]. *Integrating Extreme Programming and User-Centered Design.* In *PPIG 2008, The 20th Annual Psychology of Programming Interest Group Conference, Lancaster University, UK. 10th - 12th September 2008.*

Hussain, Zahid, Martin Lechner, Sara Shahzad, and Wolfgang Slany [2008c]. *Inside View of an Extreme Process.* In *Agile Processes in Software Engineering and Extreme Programming*, pages 226–227. LNBIP, Springer Verlag. 9th International Conference, XP 2008, Limerick, Ireland.

ISO [1998]. *ISO 9241-14:1998 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 14: Menu Dialogues.* Technical Report, International Organization for Standardization.

Janzen, David and Hossein Saiedian [2005]. *Test-Driven Development: Concepts, Taxonomy, and Future Direction. Computer*, 38, pages 43–50. ISSN 0018-9162. doi:10.1109/MC.2005.314. `http://portal.acm.org/citation.cfm?id=1092229.1092262`.

Keil, Mark and Erran Carmel [1995]. *Customer-developer links in software development. Commun. ACM*, 38(5), pages 33–44. ISSN 0001-0782. `http://doi.acm.org/10.1145/203356.203363`.

Kordon, Fabrice and Luqi [2002]. *An Introduction to Rapid System Prototyping. IEEE Trans. Softw. Eng.*, 28(9), pages 817–821. ISSN 0098-5589. `http://dx.doi.org/10.1109/TSE.2002.1033222`.

Lee, Jason Chong and D. Scott McCrickard [2007]. *Towards Extreme(ly) Usable Software: Exploring Tensions Between Usability and Agile Software Development.* In *AGILE '07: Proceedings of the AGILE 2007*, pages 59–71. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2872-4. `http://dx.doi.org/10.1109/AGILE.2007.63`.

Lewis, C. [1982]. *Using the thinking-aloud method in cognitive interface design.* Technical Report IBM Research Report RC 9265, IBM, Yorktown Heights, NY.

Lewis, Clayton, Peter G. Polson, Cathleen Wharton, and John Rieman [1990]. *Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces.* In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 235–242. ACM, New York, NY, USA. ISBN 0-201-50932-6. http://doi.acm.org/10.1145/97243.97279.

Marlow, Cameron, Mor Naaman, Danah Boyd, and Marc Davis [2006]. *HT06, tagging paper, taxonomy, Flickr, academic article, to read.* In *Proceedings of the seventeenth conference on Hypertext and hypermedia*, pages 31–40. HYPERTEXT '06, ACM, New York, NY, USA. ISBN 1-59593-417-0. http://doi.acm.org/10.1145/1149941.1149949.

Morad, Shlomit and Tsvi Kuflik [2005]. *Conventional and Open Source Software Reuse at Orbotech - An Industrial Experience.* In *Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering*, pages 110–117. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2335-8. doi:10.1109/SWSTE.2005.11.

Nielsen, Jakob [1993]. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 0125184050.

Nielsen, Jakob [1994]. *Estimating the number of subjects needed for a thinking aloud test. Int. J. Hum.-Comput. Stud.*, 41(3), pages 385–397. ISSN 1071-5819. http://dx.doi.org/10.1006/ijhc.1994.1065.

Nielsen, Jakob and Rolf Molich [1990]. *Heuristic evaluation of user interfaces.* In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM, New York, NY, USA. ISBN 0-201-50932-6. http://doi.acm.org/10.1145/97243.97281.

Nuseibeh, Bashar and Steve Easterbrook [2000]. *Requirements engineering: a roadmap.* In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46. ACM, New York, NY, USA. ISBN 1-58113-253-0. http://doi.acm.org/10.1145/336512.336523.

Obendorf, Hartmut and Matthias Finck [2008]. *Scenario-based usability engineering techniques in agile development processes.* In *CHI '08 extended abstracts on Human factors in computing systems*, pages 2159–2166. CHI EA '08, ACM, New York, NY, USA. ISBN 978-1-60558-012-8. http://doi.acm.org/10.1145/1358628.1358649.

OSI [2011]. *The Open Source Definition.* http://www.opensource.org/docs/osd.

Paech, B and K Kohler [2003]. *Usability Engineering integrated with Requirements Engineering.* In *Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, volume 0, pages 36–40. IEEE Computer Society, Los Alamitos, CA, USA.

Patton, Jeff [2002]. *Hitting the target: adding interaction design to agile software development.* In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, pages 1–ff. ACM, New York, NY, USA. ISBN 1-58113-471-1. http://doi.acm.org/10.1145/604251.604255.

Poole, Damon [2006]. *Breaking the Major Release Habit. Queue*, 4(8), pages 46–51. ISSN 1542-7730. http://doi.acm.org/10.1145/1165754.1165768.

Rajlich, Vaclav [2006]. *Changing the paradigm of software engineering. Commun. ACM*, 49(8), pages 67–70. ISSN 0001-0782. http://doi.acm.org/10.1145/1145287.1145289.

Rising, Linda and Norman S. Janoff [2000]. *The Scrum Software Development Process for Small Teams. IEEE Softw.*, 17(4), pages 26–32. ISSN 0740-7459. http://dx.doi.org/10.1109/52.854065.

Royce, W. W. [1987]. *Managing the development of large software systems: concepts and techniques.* In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0-89791-216-0. http://portal.acm.org/ft_gateway.cfm?id=41801.

Sojer, Manuel and Joachim Henkel [2011]. *License Risks from Ad-Hoc Reuse of Code from the Internet: An Empirical Investigation. SSRN eLibrary.*

van Lamsweerde, Axel [2000]. *Requirements engineering in the year 00: a research perspective*. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 5–19. ACM, New York, NY, USA. ISBN 1-58113-206-9. http://doi.acm.org/10.1145/337180.337184.

Wolkerstorfer, Peter, Manfred Tscheligi, Reinhard Sefelin, Harald Milchrahm, Zahid Hussain, Martin Lechner, and Sara Shahzad [2008]. *Probing an agile usability process*. In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 2151–2158. ACM, New York, NY, USA. ISBN 978-1-60558-012-X. http://doi.acm.org/10.1145/1358628.1358648.