FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Process Models and Project Management in Open Source Projects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Schönberger
Matrikelnummer 0325307

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 13.09.2012

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Process Models and Project Management in Open Source Projects

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Martin Schönberger
Registration Number 0325307

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig

Wien, 13.09.2012 ______________________ ______________________
(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Process Models and Project Management in Open Source Projects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Schönberger

0325307

ausgeführt am

Institut für Rechnergestützte Automation

Forschungsgruppe Industrial Software

der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 13.09.2012

Erklärung zur Verfassung der Arbeit

Martin Schönberger
Johannagasse 26/22, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

Open source software typically is developed in distributed communities, which employ specialized means of management and cooperation. The purpose of this research is to study and define a modern open source development process by identifying elements shared by open source projects and determining aspects in which they differ. Furthermore, the thesis investigates how open source development relates to structured processes on one hand, and agile methods on the other. Essential elements of the open source process are identified and compared with those of four process models which are widely used in proprietary development, namely the RUP, PMBoK, Scrum and XP. In an empirical case study grounded on qualitative interviews, developers of three active open source projects provide in-depth information on the lifecycle and practices used in the projects' development. This data is used to show how open source communities have created a distinct way of development, which is based on a shared culture and is uniquely adapted to its environment of distributed collaboration, voluntary contribution and developer participation. It is shown how open source development distinguishes itself from other approaches of creating software by a number of specific characteristics. Suggestions are offered how to improve both open source development processes and proprietary forms of development.

Kurzfassung

Die Entwicklung von Open Source Software wird hauptsächlich in verteilten Entwicklergemeinden vorgenommen, die eigene, spezifische Methoden des Managements und der Zusammenarbeit praktizieren. Diese Diplomarbeit verfolgt das Ziel, moderne Formen des Open Source Entwicklungsprozesses zu untersuchen und zu definieren, indem sie Gemeinsamkeiten und Unterschiede zwischen einzelnen Projekten analysiert. Weiters untersucht die Arbeit, in welchem Zusammenhang Open Source Entwicklung mit strukturierten Prozessen einerseits und mit agilen Methoden auf der anderen Seite steht. Für diesen Vergleich werden definierende Elemente des Prozesses ermittelt und zunächst theoretisch den vier Vorgehensmodellen RUP, PMBoK, Scrum und XP gegenübergestellt. In einer empirischen Fallstudie werden qualitative Interviews mit Entwicklern von drei aktiven Open Source Projekten durchgeführt, die Einblicke in den Lebenszyklus und die Methoden, die in den jeweiligen Projekten verwendet werden, bieten. Im Rahmen der vorliegenden Arbeit wird gezeigt, wie sich in Open Source Gemeinschaften eine eigenständige Art der Entwicklung etabliert hat, die einerseits auf einer gemeinsamen Entwicklungskultur beruht als auch in einzigartiger Weise an ihr Umfeld angepasst ist, das sich durch verteilte Zusammenarbeit, freiwillige Kontribution und Mitverantwortlichkeit der Entwickler auszeichnet. Darauf aufbauend werden klare Charakteristika herausgearbeitet, durch die sich Open Source Entwicklung definiert und von anderen Formen der Softwareentwicklung abgrenzt. Abschließend werden Anregungen zur Verbesserung von Entwicklungsprozessen insbesondere im Open Source Bereich, aber auch für proprietäre Entwicklungen gegeben.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Topic and Relevance

"The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions." - Frederick P. Brooks, Jr. [11]

In his essay "No Silver Bullet" [11] Brooks examined the technologies and management techniques of the time in order to evaluate if one of them had the potential to make software development radically more productive, simpler and more reliable over the course of the following decade. In the end, however, he concluded that the essential difficulties of the field would not be overcome by any single means, but would have to see gradual and slow improvement over a long period of time. In the 25 years since the text was written, technology has changed profoundly. Also, new methods of development and management concepts have emerged, were tested and refined, and have been adopted broadly. But still, software development remains an inherently complex and unpredictable task, and projects fail on an all too regular basis.

A traditional and widely accepted way to develop software is to plan it carefully and to apply processes and ideas from the field of engineering in its construction. Yet, the last decade also saw the rise of empirical methods, which aim to build software incrementally while being flexible enough to incorporate frequent change. Most of the approaches used today can be placed somewhere on a line between predictive and adaptive solutions, between structure and agility (see Boehm and Turner in [9]). However, there is no general agreement over which method should ideally be used in any given scenario, and proponents of each approach argue fiercely over the *right* way to develop software. This alone shows that none of the approaches in use magically solved the essential problems of software development.

When a number of large and successful projects emerged throughout the nineties that were based on free and open source software development, this movement and its methods quickly became a center of public attention. It presented itself as a radically new way of creating software, based on openness and voluntary contribution, and developed by distributed self-organized Internet communities. Naturally, some were eager to see the arrival of a new era of software development and the end of most problems troubling traditional software engineering. However, open source is not a *silver bullet*. A popular saying in the movement is that open source is not "magic pixie dust", being able to miraculously save a project from failing. It goes back to a letter in which former Mozilla developer Jamie Zawinski [68], frustrated and disillusioned, declared his resignation from the project. In the same letter, however, Zawinski asserted his confidence that "[o]pen source does work", given the necessary preconditions. And indeed, Mozilla's later history should prove him right. After the architectural structure and the development process was adjusted to fit its new open environment (as documented, for example, by [36] and [44]), the project gained momentum and became one of the world's leading browser solutions.

The huge success of some early open source projects led to increased interest from both the public and commercial developers and resulted in an outright hype during the first years of the millennium. Proponents soon claimed open source to be "faster, better and cheaper" [48] than software engineering; others, however, contested this view immediately [25]. On the whole, early research on the topic was said to be either "animated by partisan spirit, hype or skepticism" [17], while unbiased academic studies were rare. After the initial hype subsided, open source development continued to mature, and new projects helped to diversify the movement's landscape. Still, however, the now growing body of research built primarily on the roots of the movement, on major projects like Linux, Mozilla and Apache, and on the enthusiastic visions of its first few years.

There is an ongoing need for empirical and in-depth research, that takes modern projects and their adaptations and solutions into consideration. By examining several cases in detail, the thesis aims to contribute to a better understanding of how contemporary open source projects are managed and developed. Also, although open source projects are mostly developed in a special environment that is unique in several aspects, open source projects have to face many of the same challenges encountered by other types of software projects. Therefore, to discover the similarities and differences to other forms of development, the thesis will set the processes found in those projects into relation to the solutions of various agile and structured approaches. Finally, it will explore possibilities of transferring elements of open source processes to other project environments.

1.2 Research Questions

The goals of this thesis can be split into several areas of research. In order to set a specific focus and scope, the following four questions have been devised:

1.2.1 Shared Aspects of Open Source Development

Open source projects inherently differ from other endeavors of software development in a range of aspects. These differences are reflected in the way projects are organized and developed. One important question this entails is whether open source projects as a group follow the same standards and ideals, and whether they employ similar development processes. If such common elements can be discovered, a next step is to find out what grounds they are based on, and which concepts contribute to their prevalence. Can these shared characteristics be aggregated to form a distinct process model? What is the focus of open source development, and how are its core values and properties defined? Apart from these questions it shall also be assessed at this point what an open source project needs to be successful, and what the special risks and possibilities are that apply to this particular environment.

1.2.2 Agile Methods and Structured Processes in Open Source Projects

Structured processes and agile methods have developed radically different ideas about the best way to create software, which led them to thrive in different areas of application. Open source projects exist in a special environment, and follow rules which deviate from those of proprietary software development. Still, it seems plausible that common process elements are used in various areas. A primary goal of this thesis is to discover whether any and if so which characteristic elements of said approaches exist in open source projects, and how they influence the process of their development. More precisely it will be researched where developing an open source project resembles Agile Methods on the one hand, and Structured Processes on the other. Also is will be assessed if there are areas where this approach covers a sort of middle ground between the two, and in which aspects open source development follows a way that is distinct from both. Different levels of accordance will be taken into consideration, ranging from direct application of practices to an abstract transfer of ideas.

1.2.3 Variable Factors and Practical Constraints

It can be assumed, when various open source projects are inspected, that differences will be found in their practical execution, and that their development process is adapted to their respective project environment. Tasks of this thesis include examining these variable factors, finding out what their primary causes are and in which part of the process the most variation can be observed. To which degree do differences in the development depend on factors like project size, its maturity, the organizational structure or the type and intended use of the product that is being developed? It will also be assessed if the observed range of variation can give an indication for overall trends and causes in the scope of open source development processes.

1.2.4 Transferability of Results

A considerable portion of open source development methods is likely to be valid only within the specific constraints of their environment. In order to place the findings into a wider perspective, the final research question asks whether parts of the process are applicable to other tasks or if they can be adapted to fit into other fields of software development. In this respect it should be assessed if there are concrete practices which can be carried over into another environment, or if more generic ideas and methods are employed that could benefit a broader market. Can open source processes or communities contribute substantial impulses to more conventional forms of software engineering, or are their methods limited to only function well in their specific niche and environment?

1.3 Approach

In order to address these questions, the primary means of obtaining data will be through a case study, in which developers of open source projects are interviewed about the processes they employ. This case study is embedded into a framework of theoretical research, based on secondary sources, to back up the findings and help putting them into a broader context. In combination, these results will be used to define and explain common elements, variations and universal principles of open source development.

First, some of the process models used in proprietary software development will be described. Alongside general information about managing software projects and a short overview of its history, those models will be the primary focus of Chapter 2. On the one hand, examining different approaches of building software will provide information about their structure and the areas of development they cover. This knowledge will help identifying the elements that are necessary for defining open source processes. On the other hand, the two agile methods and two structured processes provide concrete reference points for later comparisons. Each approach is going to be explained through a general overview, its proposed lifecycle and its most characteristic practices. The information presented is based on a review of relevant literature, an important part of which being the original descriptions of the examined processes.

The second part of the theoretical background, laid out in Chapter 3, will cast a light on the open source movement, will describe its roots and further history, and will give a characterization of a possible development process. In addition, it includes an overview of topics usually dealt with in academic research on open source. Like the chapter before, it is based on a review of available literature. It forms the necessary foundations for the case study by defining the field of open source development and isolating the parts that are relevant for staking out its process. By comparing the findings from the literature to the other examined development approaches, the chapter arrives at a list of preliminary open source process elements.

Backed up by the theoretical findings, Chapter 4 will then describe the case study, which constitutes the core of this research. A close examination of how this case study is carried out, and what it aims to achieve, is the focus of the case study methodology, which forms the chapter's first part. Following this, the process of selecting the projects participating in the case study is laid out. Each of the three projects is then dealt with separately in a dedicated section. These sections contain a general overview, the description of the specific interviewing approach, and a summary of the development process as shared by the interview partner.

The analysis in Chapter 5 will draw conclusions from the case study data by examining it from a number of different perspectives. At first, it will separately inspect each part of the process as defined by the process categories that were previously identified. This is followed by a comparison of approaches, in which common grounds and differences between open source development and each of the examined process models are assessed. The results of the case study, combined with the findings and conclusions from the analysis, then allow answering the research questions posed in this introduction.

Finally, Chapter 6 concludes the thesis with a summary of findings. It provides a broad comparison to related work on the topic, gives suggestions for a range of possible future studies, and offers general recommendations for developers, based on the results of this research.

CHAPTER 2

Development Processes

"It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." - Abraham Maslow

Before the development of open source projects is examined and the elements of its processes are defined in detail, the following chapter investigates a number of different process models in order to establish the context and theoretical basis for further analysis.

After some introductory remarks on the nature of process models are made, a short overview on the history of software development illustrates the roots and evolution of the different approaches featured in this chapter. It is followed by a description of the characteristics of Structured Processes and Agile Methods, two quite contrary directions of development prominently used in contemporary project environments. Each of these approaches is represented by two popular process models, which are then laid out in detail to serve as a basis of comparison for the later findings of the thesis. The examples given for Structured Processes are the PMBoK and RUP, while Scrum and XP represent the group of Agile Methods.

2.1 Choosing a Process Model

Process models provide sets of guidelines and techniques to facilitate and organize project management, and increase the chance of successfully carrying out projects. In short, development processes define the lifecycle of a project and promote a set of activities considered to be best practice. However, there are many different views about how such methods of developing software should look like. Various process models, which are quite different in structure, scope and content, have been developed and formalized during the last decades. The nature of the applied process is only one factor responsible for the quality of the resulting software, but following a working methodology is key for delivering high quality reliably and repeatably.

It has often been discussed what the success rates of each development philosophy and of specific process models are. Ardent followers of each methodology have claimed that their preferred choice of process is the only rational way to successfully create software. Moderate voices, like Barry Boehm, have argued for both hybrid solutions [7], and for the concept of a certain *home ground* of different process models [9]. The latter denominates special project environments for which a specific approach is suited best. In this model, Agile Methods are preferably used in small, uncritical projects with a high rate of changes, while Structured Processes are best for large, stable, organized and critical environments. More often than not, however, real life applications are not as easily classifiable.

The question, which of the examined process models should be used in which environment, is not easily answered and lies outside the scope of this research. For the purpose of the topic at hand it suffices to say that all processes and methods presented in this chapter have been shown to be capable of delivering successful software solutions, that they are in wide use, and that the two presented groups of process models differ significantly from each other in several aspects. The last point is especially important for establishing points of reference, against which the process elements of open source development are set in comparison in the following chapters.

2.2 History of Software Development

The development of software began as a small set of tasks necessary for handling the large and expensive computers of the 1950s. These were maintained by mathematicians and engineers, who applied the principles of hardware engineering to the new field and started to *engineer* software. Those methods (like using detailed and careful up-front specifications and thoroughly testing product when they are finished) turned out to add value to the process, and were implemented eagerly in increasingly standardized ways. It became apparent, though, that software differed in several important points from hardware. One difference was that it was much easier to copy and modify code than tangible products. This allowed writing software with barely any preparation beforehand, while dealing with program errors later on. As Boehm points out in [8], the resulting *code-and-fix* programming sped up the development process and allowed people without engineering background to fill the ranks of the rapidly growing business, but led to hardly maintainable code and was not able to cope with the larger projects and more complex requirements of the late 1960s.

To avert an imminent software-crisis caused by slipping schedules and overdrawn budgets, a NATO conference on software engineering was called up in 1968, where it was postulated that software should not be tinkered, but (once again) engineered "in a clean fabrication process" [4]. Following this conference, a number of formal and structured approaches were developed in the 1970s. One of them, which became quite influential, was [46] by Winston Royce. While the original paper included prototyping and basic iterative ideas, it was often misapplied as a pure sequential *waterfall process*, based on a model Royce only recommended for smaller and simple projects. The 1980s brought a series of government standards, commissioned by the U.S. Department of Defense to normalize software processes and define the documents necessary for compliant projects. The steps needed for milestone progression and their inspection were expensive, therefore enforcing sequential, waterfall-like development [8]. Capability Maturity Models for software development were created in order to assess and improve process quality on an organizational level. The software market, however, kept growing rapidly, while new technologies and the need to deliver better quality in lower time-to-market put new constraints on software engineers and the processes they used. Iterative and Incremental Design, while not a new concept, as Larman points out in [34], gained prominence. Applying the development phases iteratively (or concurrently) allowed prioritizing high-risk or high-value features, adaption to change and early testing. The approaches introduced in this chapter, however different, all encourage iterative development to various degrees.

Two different approaches are predominantly used in present-day development. On one hand, *structured standards* are typically complex and heavy-weight and rely on detailed planning and control to ensure high quality and a predictable project outcome. On the other hand, the late 1990s saw the beginnings of *agile methodologies*, which are light-weight and empirical, and rely on rapid feedback and flexibility to be able to react to increasingly changing environments.

2.3 Structured Processes

Structured or well-defined processes as an approach to create software have existed since the early days of software development and have long been considered to be the only way to produce high-quality software with foreseeable and repeatable success. Incorporating concepts from other fields of engineering, they are based on detailed plans, extensive documentation, a rigid structure and gradual improvement of the process.

Standardized procedures are applied to transform the software from initial requirements into the finished product in a number of steps. The nature of these steps and the activities they encompass depend on the specific approach and can vary significantly. Early stages typically involve detailed planning and risk assessment, while managing in later phases focuses on processes monitoring and execution. During the project's course a number of artifacts are created and maintained in order to provide documentation for guidance and traceability. Workflows, roles and responsibilities are defined in detail to allow a higher degree of control.

The following sections highlight two popular process models further. First, the PMBoK is described, which consists of a highly formalized set of processes aimed at providing a general and complete approach for project management. The RUP, which is presented next, is a structured, but highly adjustable process, which is specialized for developing software.

2.4 PMBoK

2.4.1 Overview

The Project Management Body of Knowledge (PMBoK), developed and maintained by the Project Management Institute (PMI), is a well-known and widely used collection of processes and guidelines to plan and execute projects. Since its foundation in the late 1960s, the PMI published a number of versions of the standard. This overview is based on the fourth edition of the "Guide to the PMBoK" [42]. Collected in it are 42 processes, which are considered to be generally applicable best practices. The PMBoK is not specific to software engineering or any other field, but instead focuses on general aspects of project management, regardless of the developed product. Every process belongs to one of nine defined Knowledge Areas and is tied to the one of five Process Groups, in which all of its activities take place. Described within each process are the inputs required for its initiation, the outputs created through its completion and the tools and techniques supporting the transition from inputs to outputs. Depending on the project's size and complexity, the processes and the rigor with which they are followed have to be tailored to fit its specific needs. In their entirety the processes provide a complete coverage of traditional project management areas.

2.4.2 Lifecycle

Managing a project according to the PMBoK is highly structured. It is achieved by applying and integrating the processes to control ongoing activities, and to progress project development. The processes are logically grouped into five Process Groups, are interdependent and executed in a set order, where the output of one process becomes the input of the next. On a higher level, depending on the project's characteristics, it can be organized in several Project Phases, where each cycle of process application constitutes one phase. Project Phases differ in length and number and are either stringed together in a strictly consecutive way or overlap to varying degrees to enhance their flexibility.

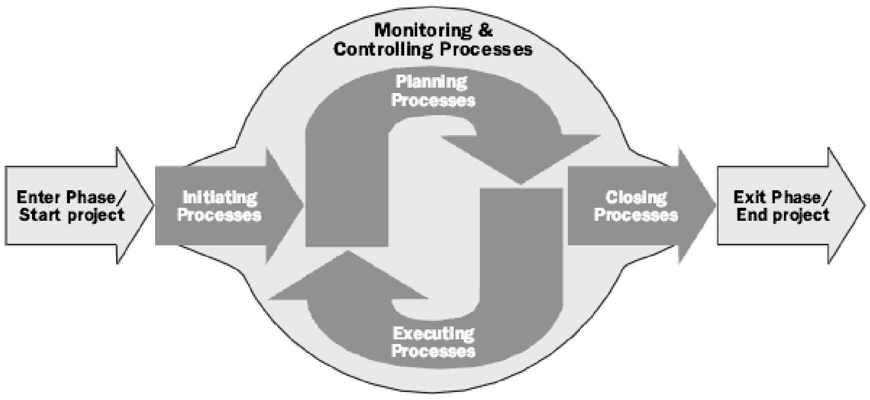


Figure 2.1: Process Groups of the PMBoK [42]

Figure 2.1 shows the flow and interactions of the five Process Groups of PMBoK. Together they form either the whole project or one of its phases, depending on the project's complexity. The Initializing Process Group contains the processes necessary for starting every project or phase. Their goals include defining the scope, identifying stakeholders, and initiating financial commitments. Scope and objectives are continuously defined and refined through activities of the Planning Process Group. Project management plans and documents are created early on, and maintained and updated throughout the project. While the project's work is carried out, the processes of the Executing Process Group help coordinating team and resources. They keep the project on track and are used to realign the plans with reality. Carried out in parallel to other activities, the Monitoring and Controlling Process Group is responsible for observing the project's progress and performance. If upcoming problems are identified, the project flow can be altered by employing preventive or corrective measures. In the end, the phase or project is formally closed and activities are finalized through the processes of the Closing Process Group. After the completed work has been reviewed, the gathered information is used to enhance the process for future applications.

2.4.3 Practices

The PMBoK is heavily structured and focuses on the description of its processes, which are employed as best practices. These processes are extensive, and can be grouped into project management knowledge areas. The following section gives a short overview on each of these areas and the processes they contain:

Project Integration Management
: This knowledge area includes the necessary processes to combine and coordinate all management activities. Integration is important when processes interact with each other, and when their goals are in conflict. The project charter, a high-level description of the project, is one of the first documents to be created. It is one of the foundations for the project management plan, in which further plans of the different knowledge areas are collectively documented. The plans are devised early, but are updated throughout the course of the project. If the performance fails to conform to the plans, corrective measures in the execution of the project have to be undertaken. Changes in the project's course have to be requested and assessed, and only if they are approved they are incorporated into the management plans. When all objectives are completed, the ongoing project or phase is formally closed.

Project Scope Management
: The main task of managing the scope of a project is making sure that work concentrates exactly on the deliverables necessary to fulfill the project's requirements. First, these requirements have to be identified in cooperation with stakeholders and through various methods, like customer workshops and interviews. The resulting detailed list serves as basis for the project scope statement, which describes deliverables, acceptance criteria and project constraints. The work is then divided and hierarchically decomposed to create a work breakdown structure, in which all project tasks have to be documented. Throughout the lifecycle of the project it is monitored whether the results are congruent with the defined scope, and whether they are accepted by the customer. If not, changes of the plan have to be approved and carried out in order to bring the project back on track.

Project Time Management

Time management gives temporal structure to the project, making sure that activities are completed in a timely manner. The processes it contains are responsible for creating a schedule, and they also ensure that it is kept. Starting from the work breakdown structure, the project manager defines a list of activities and milestones. This list is then arranged into a network diagram, which takes dependencies and constraints into consideration and allows creating a definite sequence of all tasks. A variety of methods and tools are employed to estimate the duration of activities and which resources they probably need. This data is combined into a schedule, which gives specific dates for milestones and start and end dates for activities. It is completed in the planning phase, but may be refined at a later time. The progress of the project schedule is measured and controlled throughout its execution.

Project Cost Management

Controlling the costs of the project and making sure that expenditures stay within acceptable limits is an important area of the PMBoK. Cost management starts early in the project lifecycle and is the foundation for many other plans. Based on the scope baseline and the project's schedule the costs for all activities are collected and separately estimated. This list is then aggregated and extended in order to form a complete budget. A further evaluation tries to determine the necessary funding over time. During the course of the project it is monitored how much money has been spent and how this relates to the value created by the project. This information is gained by utilizing statistical analysis, forecasts, performance reviews and budgeting tools. If possible, cost overruns are detected and measures are undertaken to minimize the damage.

Project Quality Management

Setting up quality policies and standards, and making sure that they are fulfilled throughout the project, are the main objectives of quality management. The quality of the final result has to be high enough to satisfy the demand of the customers and to meet their acceptance criteria. Planning considers the cost of implementing quality raising methods, and compares it with the loss caused by failures. In combination with several analysis methods, statistical samples, flowcharts and tools this leads to the creation of a detailed quality management plan and corresponding metrics. Processes which control the adherence to quality standards are carried out throughout the execution of the project. At the same time they collect data in order to improve quality management over time, to benefit current and future projects.

Project Human Resource Management

Human resource management combines the processes that are necessary for forming and managing the team of people who carry out the project. It tries to take human factors into consideration, but also aims at influencing them for increased success. Specific roles and responsibilities that are required for project completion are determined at the start of the lifecycle. These positions are arranged in a fixed hierarchy and are then filled with readily available or specifically hired personnel. The ongoing tasks include keeping up the performance of team members and further developing both the skills of individuals and their work efficiency as a team. How well the team performs is monitored through observation, reviews and reports of the project status. Managing the team is based on a combination of interpersonal and leadership skills, and formal methods.

Project Communications Management

A primary task of the project manager is to collect information and distribute it through appropriate channels. It focuses on communicating the information that is most important for project success in an efficient way. An early goal is to find out who the project's stakeholders are and what they expect from its development. This becomes the main source for the communications management plan, which also defines the methods, models and technologies used for information transfer. Feedback from stakeholders is integrated in the form of change requests, which lead to changes in the project if they are approved. Process reports are a different part of communication management. They are based on performance information and measurements, and are distributed in varying levels of detail to different audiences.

Project Risk Management

The main goals of risk management are to identify the risks that can possibly happen in the project, to analyze them in detail and to plan appropriate countermeasures. At the same time it tries to raise the likelihood and impact of project opportunities. The process tries to detect most risks in the early phases of the lifecycle. Before that, however, a risk management plan has to be formed, which describes the risk-related activities carried out in the project and the level of information stored with identified risks. Reviewing project plans methodically results in the list of possible risks and opportunities. Risk analysis is carried out qualitatively, and quantitatively if necessary, to prioritize them. The probability and impact of risks is assessed, and they are categorized accordingly. Appropriate responses are individually set for all entries of the list. During the course of the project, risks are continuously monitored and are reassessed if necessary.

Project Procurement Management

Procurement includes processes which organize buying or otherwise acquiring products and services from third parties. It helps managing contracts over the whole duration of their lifecycle. The first point of consideration is deciding which portion of the product is developed inside the project, and which should be acquired from outside. This results in a procurement plan and accurate descriptions of the desired parts. Analyzing the offers of different sellers and carrying out negotiations leads to the signing of a contract. The performance of all contracts is then observed in an ongoing process, and it is being ensured that obligations are kept. Contracts are formally closed when they have been fulfilled.

2.5 Rational Unified Process

2.5.1 Overview

The Rational Unified Process (RUP) is a process framework for software engineering, originally developed by Rational Software, and now maintained by IBM. Its creators describe the RUP in [32] to be both a structured software process and an iterative software development approach. It is driven by use-cases and risk and is highly flexible with an adaptable level of ceremony. It does, however, clearly define roles and responsibilities, and it provides a structured time frame for the lifecycle with major milestones and decision points. In addition, the RUP is treated and sold as a software product. As such, it is regularly updated, can be browsed and searched online, and customized to business needs.

At an architectural level the RUP can be viewed in two dimensions. The static structure groups process elements logically into disciplines: it defines activities, roles, artifacts and workflows. Also, it provides guidance in the form of templates, tool mentors, roadmaps and similar concepts. The second dimension structures the project's temporal flow and forms a lifecycle, which is described in the following section.

2.5.2 Lifecycle

The lifecycle of RUP is highly structured and describes the phases Inception, Elaboration, Construction and Transition. Every development cycle of these four phases results in the creation of a new product generation. Phases are ended with the completion of certain predefined milestone activities. It is possible to partition each phase into several iterations. This iterative approach mitigates risks early and helps managing change whenever it occurs. The development flow, its activities and their significance over time are illustrated in Figure 2.2.

While the milestones finalizing each phase and a number of other process elements are typically present in each project, the workflow on the whole is not fixed, and is subject to change. Artifacts are maintained and constantly revised during the whole course of the project. Both the workflow used and the artifacts created should be tailored to reflect the actual needs of the project, and to incorporate the desired level of ceremony. The following section describes the phases of RUP, their corresponding milestones and the activities they focus on.

Inception

The main objective of the first phase, which usually consists of only one iteration, is to gather information about the general scope and course of the project, about its desired functionality and which basic requirements need to be fulfilled to achieve it. At least one possible solution is sketched, and it is evaluated whether it

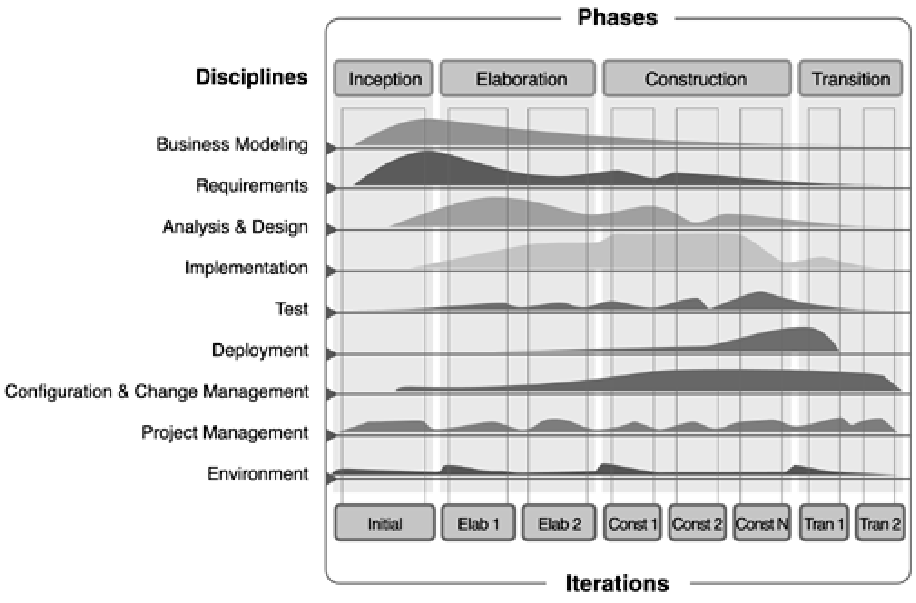


Figure 2.2: Phases of the RUP [33]

is realistic and financially viable. If this is the case, the *Lifecycle Objective Milestone* is achieved, and the project enters the next phase. Otherwise, the project is to be reconsidered or aborted.

Elaboration

During this phase, requirements are reassessed and refined. This information is used to design, implement and test a working prototype. In later phases, this prototype is evolved into the final product; therefore it is already built on a stable and working architecture. A subset of architecturally significant or otherwise critical scenarios is implemented to identify and mitigate technical risks early. Also, the development environment is set up at this point, and the process is customized. Depending on project complexity, elaboration can consist of one or more iterations. When the architecture and the project vision are stable, the *Lifecycle Architecture Milestone* is reached.

Construction

Starting from the foundation created before, the phase of Construction aims to develop this prototype into a complete, operable version of the system. It is usually the most time-consuming phase, requiring the highest amount of work and the highest number of developers. It therefore typically consists of several iterations. At this point, most major risks have already been mitigated. During Construction, the RUP is used primarily to minimize the cost of development, optimize the use of resources and ensure a high quality of work-products. When the product fulfills the targeted functionality, a beta version is released. This forms the basis for the next milestone, the *Initial Operational Capability Milestone*.

Transition

At this point, the system should already be stable and sufficiently tested. The final phase of the RUP lifecycle therefore consists of fine-tuning and preparing the product for release. Beta tests are the basis for user-feedback and ensure that all their needs are met. Other tasks include user training, deployment, and possibly marketing and packaging. Insights gained in the project are collected to increase future process performance. When stakeholder satisfaction and project goals are reached, the project ends with the *Product Release Milestone*. Afterwards, either a new development cycle is started, or the project is considered to be finished and maintenance begins.

2.5.3 Practices

The following list of practices is based on process components endorsed by [33]. Although more practices are included in the product, those given provide a starting point and can be followed in most projects.

Iterative Development

As the lifecycle suggests, the RUP is designed to be used iteratively. The process model recommends timeboxed iterations of two to six weeks and encourages creating early executables for feedback, which can be used to detect risks and flaws. The goals and duration of each iteration, as well as their total number, are factors which are defined by the process, but can be adjusted to accommodate for incoming changes.

Requirements Management

A number of methods are used in the RUP to find functional and nonfunctional requirements of a system, including workshops and use-case modeling. As requirements get more detailed, or change during the course of the project, it becomes a major task to organize the requirements based on risk and priority, while their status is tracked continuously. This process is heavily tool-supported.

Architecture and Components

An early emphasis is put on developing a structured and cohesive architecture, in order to discover technical risks as soon as possible, and to be able to address them accordingly. An executable prototype with limited functionality is created using this architecture. Later it is extended and finally developed into the final product. A component-based approach is chosen to encapsulate functionality and facilitate reuse.

Visual Modeling

The RUP uses a number of different models to simplify complex systems, and to provide different perspectives to the various roles and activities. It encourages doing some kind of visual modeling before programming in each iteration. Usually those models are based on the UML, which provides a common vocabulary and form. Different tools and techniques are described and can be used depending on the complexity of the project.

Configuration and Change Management

As a project evolves, so do its artifacts. Keeping them updated and synchronized with each other requires some work, which is covered by this practice. Configuration management involves version control and maintenance of interdependent artifacts. Other closely related activities include the management of change requests, and the collection of information for status assessment. Again, tools are provided to automate most aspects of this practice.

Verify Quality

Testing starts early and is carried out throughout the whole project course. At the start of the lifecycle, standards are defined and a testing plan is created. Conforming to project specifications and finding errors soon in order to reduce their costs are the main objectives of the testing process. Also, the tests are used to monitor the project's progress. Although tests preferably cover all use cases, risky areas of the code are tested most extensively.

Use-Case-Driven Development
Use cases provide a way to talk clearly about problems and requirements with different stakeholders. In the RUP they drive various activities and are used in design, implementation and as a basis for acceptance tests. Related use cases are organized in a use-case model, which serves as an interface to other model representations of the system.

Process Configuration
RUP is highly customizable and should be tailored to fit specific needs. Depending on the project's size and complexity, artifacts and workflows can be omitted if their cost exceeds their value. However, the process can also be enhanced with custom practices and rules. As long as the proposed practices and guidelines are followed, the RUP can cover a wide array of development styles.

Tools Support
Many activities in the RUP can be automated using tools, especially creating and maintaining artifacts and models. The process description lists tools for various tasks and gives detailed guides explaining their usage. Tool-support is optional, though. In smaller projects it can be substituted or complemented by hand-written models, paper cards and similar methods.

2.6 Agile Methods

Since the late 1990s, a number of Agile Methods for developing software emerged and became popular. Although they differ from each other in form and scope, they share common roots and concepts. Many of the ideas picked up by the movement, like iterative development, continuous integration and the focus on delivering executable software rather than byproducts, were not new as such. However, by combining these ideas and pushing them further, by including some new practices, and by holding them together with common sets of values, a novel approach of developing software was created [34].

In 2001, a number of creators of popular agile methods met to agree upon a common vision of their way of developing software, and wrote it down in the Agile Manifesto [2]. This manifesto contains several weighted values and a set of principles, with a strong focus on human factors and other key aspects shared by Agile Methods. It is included in its entirety in Appendix D.

> "[. . .] 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."

Although the individual methods may differ in their chosen path to reach these goals, the principles and values agreed upon in this initial meeting provide a common scaffolding for Agile Software Development. Since the publication of the Manifesto, diverse methods with agile methodology have been devised and used, with varying success, as Abrahamsson notes in [1]. The use of agile methods is on the rise ever since, and although their adoption often is incomplete or mixed with other approaches, as West declares in [64], he sees this upwards trend to continue further. Two of the most prominent and most widely used approaches, Scrum and XP, are described in the following section as typical examples of this family of development methods.

2.7 Scrum

2.7.1 Overview

Scrum is an agile project management method and is based on an empirical process control model. It was first referenced by Takeuchi and Nonaka in [59] to describe a development process observed in Japanese companies. The method was named after a strategy in rugby, underlining the claim that in agile development, as in the ballgame, teams have to be adaptive, quick and self-organizing in order to succeed. Scrum was formalized and evolved by Ken Schwaber, Jeff Sutherland and Mike Beedle during the late 1990s and early 2000s and has since then become the most popular Agile Method in use [64].

The emphasis of Scrum, according to its creators in [51], lies in project management and the project's lifecycle. It does, however, not provide explicit programming practices. Because of this nature, it can be easily combined with other methods. The method emphasizes iterative design, self-organizing teams and a daily measurement of progress. Although it can be used for larger projects by combining several teams, as described by [58], it is usually employed in small, co-located project environments. The next section explains the lifecycle used in projects managed by Scrum, and introduces some of its roles, practices and artifacts. Subsequently, it describes these practices in detail, and lists the values given by the definition of Scrum.

2.7.2 Lifecycle

As the first activity in the Scrum lifecycle, a vision for the developed product is articulated. Desired features, functionality and technology are collected and composed into an initial Product Backlog. This prioritized list is managed by the Product Owner and is constantly updated and refined during the course of the project. Development is structured into Sprints, time-boxed iterations of several weeks length. At the beginning of each Sprint a Sprint Planning Meeting is held, in which the project team chooses items from the top of the Product Backlog to implement during the next iteration.

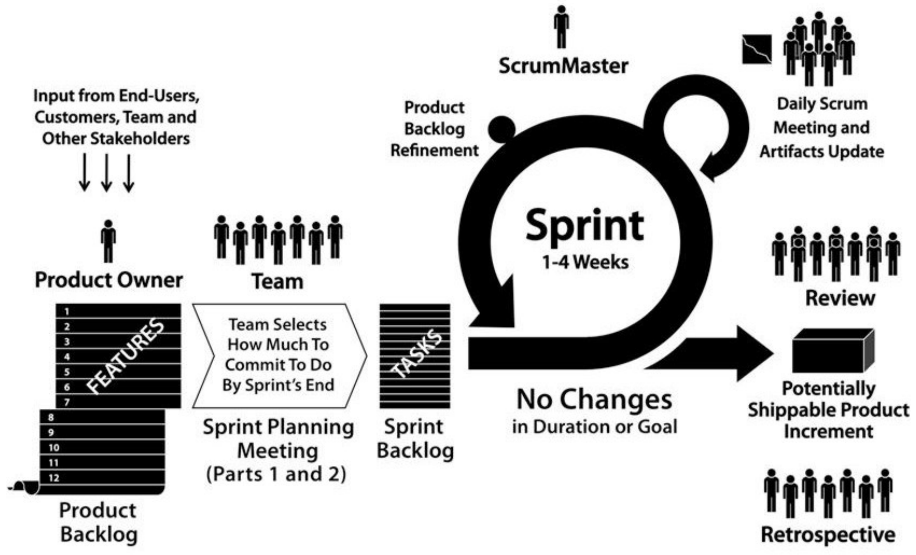

ScrumMaster
Input from End-Users, Customers, Team and Other Stakeholders
Product Backlog Refinement
Daily Scrum Meeting and Artifacts Update
Sprint
1-4 Weeks
Product Owner
Team
Review
FEATURES
Team Selects How Much To Commit To Do By Sprint's End
TASKS
No Changes
in Duration or Goal
Potentially Shippable Product Increment
Sprint Planning Meeting (Parts 1 and 2)
Sprint Backlog
Product Backlog
Retrospective

Figure 2.3: Project Flow of Scrum (based on [19])

The team has the authority and autonomy to decide how to fulfill the project goals. It is assisted by the Scrum Master, who guides the team, removes impediments and makes sure the practices of Scrum are being understood and followed. In a short daily meeting called the Daily Scrum, the members of the team tell each other the progress of their work, synchronize their plans of action and identify possible blocks. Each Sprint ends with a Sprint review, where the team shows their progress to stakeholders and discusses future directions. The Product Backlog is updated and forms the basis for the next Sprint. Figure 2.3 provides an outline of the lifecycle of a typical Scrum project.

2.7.3 Practices and Values

The practices of Scrum mainly consist of the roles, artifacts and management activities already mentioned in the description of the lifecycle. The following section illustrates them in greater detail.

Scrum Master

The Scrum Master, as introduced and defined in [51], is a key role to the success of Scrum. Main responsibilities of the Scrum Master are to make sure Scrum practices and values are understood and followed by anyone involved in the process, and to enable the team to work unimpeded and with maximum productivity. The Scrum Master should possess the authority and determination to protect the team members from outside influences and to remove blocks. Rather than assigning tasks, the Scrum Master is encouraged to facilitate self-organization of the team. She also serves as initiator and moderator, and communicates between management, team and the Product Owner.

Team

The second important role defined by the method is the Scrum Team. The team includes all the people working on the realization of the project. It should ideally be cross-functional and be formed of, more or less, seven persons. In larger projects, several teams can be formed, which are then each coordinated by their own Scrum Master. Within the team there are no special roles or titles, instead everyone is asked to put their talents to the best use of the project. The team commits to do a certain portion of work for every Sprint and chooses for themselves how to get it done. For best efficiency, teams should be stable, dedicated to one project and work together in a common and open workspace to facilitate communication.

Product Owner

The role of Product Owner is ideally held by a single person with authority, who is responsible for the value of the product. The Product Owner lists desired features, arranges them based on their business value, risk and cost, and maintains this list as Product Backlog during the course of the project. Compared to the traditional role of a product manager, the Product Owner interacts more closely with the development team, reviews their success and offers them the possibility to choose items from the top of the Product Backlog to implement in their Sprints.

Product Backlog

Originating from the product vision and influenced by stakeholder interests and feedback of the team, the Product Backlog is an ever-evolving list of all features, functions and technology goals to possibly make it into the final product. Relative estimates of effort, value and risk are the basis for a prioritization of items on a list, with high-value and high-risk items being tackled first. Lower priority features can be sketched roughly as they arise and are refined in more detail as their implementation draws nearer.

Sprint Planning Meeting

Before each Sprint begins, the Product Owner and the team come together in a Sprint Planning Meeting to map out the work for the next development iteration. The meeting starts with a review of top priority items on the Product Backlog to gain a common view of upcoming tasks. Based on this understanding and considering the available workforce, the team chooses a manageable amount of items from the top of the list to implement in the upcoming Sprint. Necessary design decisions are made, the chosen backlog items are decomposed into fine-grained individual tasks, and are then collected into a Sprint Backlog. Finally, the team members split the tasks between each other.

Sprint

A Sprint in Scrum is a time-boxed iteration of incremental development. The length of a Sprint is agreed on in advance, should never be extended and typically lasts between two and four weeks. The foundation for the Sprint is the backlog. As soon as those goals have been set, they cannot be altered externally. Any additions or changes are delayed to the next Sprint to allow the team to focus on their chosen tasks. Progress on the tasks is measured daily by recording the estimated time remaining on a Sprint Burndown Chart and by the Daily Scrum Meeting. If it becomes evident that the backlog will not be cleared by the end of the Sprint, scope or functionality can be reduced by the team. Unimplemented features are then added to the Product Backlog in the next Sprint Review Meeting.

Daily Scrum Meeting

The Daily Scrum, which is an essential element of the method, is a short daily stand-up meeting to synchronize and facilitate work. It is moderated by the Scrum Master, who asks all team members to report in a group setting what they did since the last meeting, what they are planning for the next one, and what impedes their progress. The Scrum Master has the responsibility to help the team removing those blocks.

Sprint Review

Every Sprint ends with a Sprint Review, where the team comes together with the Product Owner, the management, customers and other stakeholders to discuss the status of the project and the results of the last Sprint. The team presents the newly completed functionality with a live demonstration, while everyone else is encouraged to provide feedback or ask questions. This conversation increases the transparency of the project and provides input for the next Sprint, which starts immediately after a new Sprint Planning Meeting. This series of Sprints is repeated until the product is ready for release.

The values described in Scrum are *Commitment*, *Focus*, *Openness*, *Respect* and *Courage*. They are the foundation for the practices and serve as a basis of decision-making. As the team commits to a common goal and is assisted by the Scrum Master to be able to focus on its completion without distractions. Cooperation inside and outside the team should be based on mutual respect, while responsibility is taken by the group, not the individual. Information about status and course of the project is open and visible to all participants. Finally, the method asks for the courage to uphold its values and principles.

2.8 Extreme Programming

2.8.1 Overview

Another well known and widely adopted agile method is Extreme Programming (XP), created by Kent Beck [6]. It provides a body of software development practices based on a set of *agile* values to achieve fast and sustainable development in environments open to change, while reducing the overhead of detailed documentation and rigid processes. Its philosophy is to combine proven concepts like code testing and review, customer involvement, early feedback and frequent integration and take them to, in Beck's words, "extreme levels". The practices are intended to complement each other. It is argued, and shown in empirical studies like [29], that the effect of practices is amplified when they are used in combination.

In the first edition of his method description, Beck defined XP as “a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements” [5]. Later, he broadened this definition to say that with some alterations, larger teams and projects can possibly be realized. Cao et al. describe such a case of a large-scale project in [14], while Boehm and Turner point out in [9] that such alterations are likely to introduce elements of plan-driven processes into XP.

This section continues with an overview of the method’s lifecycle, before the practices of XP and its underlying values are illustrated.

2.8.2 Lifecycle

XP aims to be highly iterative in order to gain early feedback and to produce new value rapidly and continuously. Customers write short Story Cards for desired features, which are then roughly estimated by developers. After date and stories for the next release are agreed upon, development starts. For every time-boxed iteration, which is preferably as short as one week, customers choose a set of stories for implementation, based on priority and available time.

The development team breaks up the stories into short tasks and implements them using practices like test-driven development and pair programming. During all phases of development the customer collaborates closely with the development team, ideally working on-site to be able to clarify requirement and design issues as soon as possible.

2.8.3 Practices and Values

Underlying the practices of XP is a set of general values, intended to give them direction and purpose. Beck stresses that without values to guide them, practices are in danger of becoming meaningless routine, a “thing to do to check off a box” [6]. The values of XP are listed as *Communication*, *Simplicity*, *Feedback*, *Courage* and *Respect*. Communication, especially in the form of personal conversations, is valued highly in XP as a way to strengthen teamwork, to convey the needs of the customer and to overcome recurring

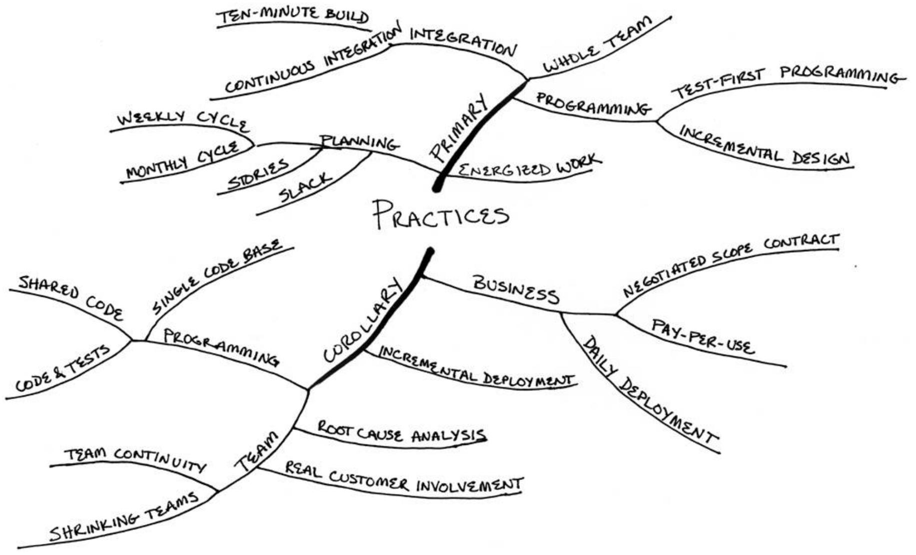


Figure 2.4: Primary and Corollary Practices of XP [6]

problems. The value of simplicity encourages looking for the most basic solutions, tools and designs that are promising to work in a given context and to avoid complexities that may or may not arise in the future. Quick and detailed feedback is highly valued in the environment of rapid change to judge a project's status and to adjust the direction of its development. Furthermore, XP advocates courage to act according to these values even in situations of fear and stress and asks to respect the personality and contributions of the persons involved in the process. The mutual interdependence of values and practices is confirmed in observational studies like [45].

Complementary to these rather abstract values, XP describes a collection of principles, which are used to map those values to the specific practices of software development. It emphasizes humanity and human interactions, diversity of teams and shared responsibility. Also, it encourages a rapid iteration of small steps of improvement, instead of major changes. Problems and occasional failure are accepted if they are used as a way to gain knowledge. Because available time and money are usually fixed, a project is steered mostly by a continuous adaptation of scope.

The practices provided by XP are specific and situation-dependent working instructions for developing software. A distinction is made between primary and corollary practices. Figure 2.4 shows a sketch of all practices contained in XP. The thirteen primary practices represent the core of XP, and are general enough to be followed in most circumstances. Those practices are sketched out individually in the next section.

Sit together

The method encourages establishing an open, shared workspace for all members of the team. Additional workstations can be provided outside this common area to fulfill the developers' need for privacy. Personal communication and co-location are considered to be important aspects of XP.

Whole team

A team in XP should be cross-functional and the developers working in it should possess all needed skills and the necessary know-how. Tasks and responsibilities are taken over as a team. These practices originally also included the requirement of customers working on-site with the developers. However, with the second edition of [6], this was extended to be an independent (corollary) practice.

Informative workspace

Making information visible is seen as an important measure to enhance feedback about a project's course. This practice encourages putting up big, handwritten charts and story cards, sorted by state of completion, to display up-to-date status information directly at the team's workspace.

Energized work

XP discourages developers working overtime, as its creator doubts that it is sustainable and effective over time. Beck argues that working while being tired or sick not only slows progress of development, but poses the threat of removing value from a project through an increased rate of errors. A 40-hour week is seen as ideal, and countermeasures (like reducing the scope) are used in order to realize this.

Pair programming

A core practice of XP is to write all production code in pairs, with two persons working simultaneously on one shared workstation. Although this seems wasteful at first, authors like Cockburn (in [15]) have argued that higher code quality (due to instant code review) and increased trust make up for this perceived loss of productivity. To maximize the gain of mutual learning and the feeling of shared code ownership, pairs are rotated regularly, every few hours.

Stories

In the Planning Game, desired functionality and features are gathered by the customer and written down on index cards as *stories*. To be able to quickly prioritize, developers should estimate the stories as soon as they are written. It is encouraged to use actual cardboard cards in order to keep the descriptions short and to create tangible items for visualizing progress.

Weekly cycle

XP encourages developing in short iterations of roughly one week. Each phase begins with the selection of stories to be implemented. The stories are split into tasks, which the team members then distribute among themselves. Work starts with writing automated tests and ends, when stories are functionally implemented. After every cycle it should be possible to deploy working software.

Quarterly cycle

In addition to the weekly iteration, XP defines another cycle every quarter of a year. This should be used to plan at a larger scale and to make sure that the project still follows it initially defined vision. Also, every such cycle should provide room for reflection on the team, the project and the used methods, in order to detect and overcome bottlenecks and to make long-term decisions.

Slack

XP encourages clear communication and realistic schedules to avoid overcommitment. Broken promises are to be avoided because of their long-term negative impact on customer-relations. Slack also suggests to add additional, minor tasks to plans, which can then be omitted if necessary.

Ten-minute build

In XP, the process of building and testing the system should be fully automated, to facilitate maintenance and make the process repeatable. Additionally, it should be continuously optimized to take less than ten minutes, because longer builds tend to not be used regularly. If it is still not possible to test the whole system in time, automatic testing can be limited to the most error-prone parts.

Continuous integration

To reinforce short feedback cycles, XP employs continuous integration. Changes to the code are to be integrated in the main build as soon as possible, preferably after each session of pair-programming. Before starting a new task, the integrated system should be built and tested. It is argued that correcting bugs is easier if it is done directly after introducing them into the code.

Test-first programming

Before any code is implemented, a short unit-test is written and coded against, until it passes correctly. Following this practice ensures that testing is not neglected, helps focusing on the intended result of the task at hand and encourages a loose coupling of components. The working tests are then added to the automated build.

Incremental design

Instead of creating an elaborated design prior to implementation, XP encourages to start with a simple design and enhancing it whenever necessary. There should be continuous working on the design to keep it in proportion with the current system. Large changes are applied in small steps, to profit from feedback sooner. Refactoring aims at eliminating duplication of code in order to keep the design more flexible.

CHAPTER 3

Open Source Development

"Like many things about the Internet era, open source software is an odd mix of overblown hype and profound innovation." - *Steven Weber [63]*

Open source is a complex phenomenon. It is a social movement, an ideology, a set of legal definitions and a development process. Reflecting briefly on the whole approach and giving an overview of the underlying concepts and principles provides the necessary foundation to tackle more specific process-related questions later on.

The general outline begins by summarizing popular definitions and explains the roots and further development of the movement in a short historical overview. It is followed by a survey of prominent research topics and their respective relevance for the topic at hand. Common elements found in open source projects and the corresponding literature are then used as a basis to present a development cycle and to showcase the practices typically used in the process. Finally, the thereby established process model is split up into basic elements to be compared individually to the approaches described in the previous chapter. This theoretical comparison provides preliminary answers for the categorization of open source process elements and builds the foundation for the following case study.

3.1 Definition

The general concept of open source has become well-established and its central premise, to provide free access to a software's source code to everyone, is widely known. In order to delve deeper into the details of open source development later in this chapter, it is, however, necessary to provide a closer perspective on its exact definitions.

Due to the different roots and motivations that led to the movement's origin, several names are in common use. The terms Free Software and Libre Software (the latter to avoid ambiguities around the word *free* in the English language) predate *open source*, and a part of the movement prefer them still (see [56], for example, for a longer treatise on the topic). This leads to the emergence of umbrella words such as F/OSS, F/OSSD or FLOSS, which are used to cover the most popular variants. However, the differences in denomination are mainly due to ideological issues and differences of opinion outside the main scope of this research. In the context of the development process, the term open source is predominant and is therefore primarily used throughout this thesis.

The license under which software has to be published to be considered open source (or free software) has to fulfill a set of characteristics, formulated in the Free Software Definition by the Free Software Foundation [23] and the Open Source Definition by the Open Source Initiative [28]. Although these two definitions differ in their rhetoric and in some detail, they widely agree on content, and the most commonly used licenses are in accordance with both, as is explained further by Grassmuck in [26]. The following breakdown gives an overview of the key ideas covered by the two definitions:

Access

Providing free access to a readable version of the source code is a primary requirement and a necessary precondition for other rights. It covers not only the right to study the code passively, but also to modify it. Everyone has the same rights to access the code, and the license can not be used to hinder individual persons, groups or whole professional fields from using the software or contributing to its development.

Distribution

Another basic right allows everyone to redistribute copies of the software, including modified versions. Although distributors are allowed to charge fees for the effort of distribution, those fees can not be required by the license, and free ways to obtain the software have to exist. The rights given by the license automatically extend to all copies.

Constraints

The definitions contain several additional constraints and characteristics. The license may require modified versions to be redistributed under a different name to protect the authenticity of the original version's author. Other issues prescribe that the license is not coupled to a specific technology, and that no restrictions can be placed on software just for being distributed together with open source software. It can, however, be extended to software with which it interacts, a fact that is used in the GPL and attributes to its viral character.

Licenses conforming to the aforementioned points differ profoundly from proprietary contracts and licensing schemes, and introduce a diametrically different concept of ownership. Still, taken on its own, this does not require fundamental differences in the process of development, and a number of projects with an opened source code are nonetheless produced with conventional methods by an in-house development team. In many major open source projects, however, a common process has evolved, which draws heavily on the possibilities of distributed cooperation and voluntary contribution.

Before this process is described in detail and common grounds and differences are elaborated, a short history informs about the different roots and mile stones in the development of the movement.

3.2 History of Open Source

In the first decades of computing history, software was only a by-product to keep the large and expensive mainframes running, and was of small commercial value in comparison. Code was often shared and collaboratively written by developers of different companies to create complex machine-code instructions, a behavior encouraged by manufacturers to raise the value of their hardware. This cooperation was limited however, as

Karl Fogel notes in [21], by technical obstacles like serious hardware incompatibilities and the absence of a global network.

When in the seventies computers spread to universities, these places became major places for innovation, two of which being especially important for the history of open source: the University of California at Berkeley and the Massachusetts Institute of Technology (MIT). With increased hardware standardization and the introduction of high level programming languages, the value of software rose, however, and expensive proprietary licenses restricted sharing and joint improvement of code.

Steven Weber has researched in depth the role Berkeley played in contributing to the beginnings of open source in [63]. They had collaborated with AT&T on the advancement of the UNIX operating system, releasing patches and packages of tools as Berkeley Software Distribution (BSD). When AT&T massively raised UNIX licensing fees during the eighties, the developers at Berkeley began to replace proprietary parts by re-engineering them, assisted in this effort by voluntary contributions over the Internet and resulting finally in the free operating system 386/BSD, distributed under generous licensing terms.

The Artificial Intelligence Lab at the MIT was another central hub for software development, and had developed a distinct culture of openness and code sharing. However, this community did not last through the changes brought to the industry by new systems that came without sources and asked for confidentiality and non-disclosure. Richard Stallman, one of the developers who was unwilling to leave the old ideals behind, initiated a counter-movement and founded the Free Software Foundation with the goal of developing a completely free and open operating system. While similar efforts at Berkeley followed rather pragmatic intentions, Stallman's GNU project had strong moral and ideological undertones about the freedom of information. In the next years the project grew and resulted in an extensive suite of utility tools, published under the GPL, a special license Stallman had devised, using copyright to ensure sustained freedom of the software and its derivatives, as Grassmuck notes [26].

In 1991 GNU still lacked a kernel though, the most important part for any independent operating system. At this time the Finnish student Linus Torvalds created a simple UNIX-based kernel, mostly for his own use, and posted it on an Internet newsgroup to be used and modified freely; but although it was simple, it was working, attracted attention and with the help of many volunteers and combined with the available GNU tools it became the first major free operating system, GNU/Linux. The way Torvalds coordinated the project and its contributors, gave direction to a new process of developing software, and Linux eventually evolved to become one of the most prominent success stories of open source. The same time, however, marked the beginnings of other highly successful projects of a similar nature, like the Apache web server, led by a committee of distributed developers, who began their work in 1995 and continued to develop Apache to become the standard choice for HTTP servers ever since [35].

Still, however, the community lacked formalism, common grounds and a name everyone could identify with. Several crucial events, like the publication of Eric Raymond's influential text "The Cathedral & the Bazaar" [43], sparked a discussion about identity. In 1998 the name open source was coined in an effort to re-brand the movement to appeal to companies, to move away from Stallman's controversial ideology and to propagate the development model for pragmatic reasons of quality and cost. The marketing move worked, and when several large international companies decided to create Linux ports for their systems to lessen their dependency on Microsoft's operating system Windows, it led to increased public awareness and credibility for open source software in general. Competition with Microsoft was also a direct cause for another project to join the OS palette when Mozilla decided to release the source code for their browser software in a tactical maneuver to gain ground against the Internet Explorer.

Although the public hype surrounding open source abated slightly, the last decade saw a further maturing of the process and its projects, increased theoretical research and a large number of newcomers. Seen as a viable alternative to proprietary systems in many areas, it also gained a foothold in commercial settings, and is increasingly supported and used by companies of all sizes.

3.3 Areas of Research

A growing body of research is dedicated to understanding the concepts constituting open source development. In this study of the field a number of key areas of interest have emerged, and the primary goal of this section is to give a structured overview of these research topics. A list resulting from this categorization will furthermore serve as a basis to evaluate each topic's contribution to the process and its relevance to the core issues at hand.

Several undertakings have been made by different research groups to lay out the existing works on open source and summarize their results, some examples being [61], [50] and [18]. The latter uses an input-mediator-output-input (IMOI) model to structure the data, where mediators influence the transition from inputs to outputs and are made up of processes and emergent states. Outputs contribute to the inputs of future iterations. The following overview of topics and the findings linked to them build upon this model, but leave out the process, which will be dealt with more thoroughly in a subsequent section.

3.3.1 Inputs

There are three relevant groups of inputs to open source, according to Crowston et al. [18]. The first concerns member characteristics, and deals with the geographical location and distribution of contributors as well as their motivations on individual level and from a company perspective. Of further importance are the characteristics of the project, mainly the influence of different license types, and the use of technology as a means of coordination.

Findings

The question of motivation, why people contribute their time and energy to work on an open source project, has traditionally been a major issue of empirical and theoretical research. Personal needs to improve the software have been shown to often be the decisive factor for initial participation, and many contributors leave after these initial needs have been fulfilled. For those who stay past this phase and increase their participation, the initial reasons to take part are superseded by a mix of intrinsic and extrinsic motivations (see, for example, [41]). Examples for the former include having fun at coding, taking an active part in a community and gaining opportunities to gather and pass on knowledge. The latter comprise of gaining reputation and status, raising one's worth on the market and, when being employed by a firm, getting paid. Companies are in turn motivated to participate by the opportunity to gain large-scale external support and a corresponding increase in quality. Comprehensive studies empirically researching motivation in open source projects, such as [52], identified several groups of developers, such as need-driven participants and hobbyists, contributing differently because of highly diverse motivations.

Open source contributors are globally distributed, with English being the common language in a majority of projects. The type of the license plays an important role in the categorization of a project and has influence on a number of factors like governance and relationships to commercial companies. Another important input is the used technology, as it enables most communication in an open source project and is the main means of creating and sharing knowledge.

Relevance

Motivation is a topic important to understand when considering open source, but not directly relevant to many issues of the development process. It is important, though, for issues of collaboration and conflict management, to keep the diversity of people in terms of culture, motivation, origin and language in mind. Also, the use of technology for communication plays an important role in the process.

3.3.2 Emergent States

Like processes, emergent states are mediators controlling the transition from project inputs to desired outputs. They describe both social characteristics like trust inside the team and task-related concepts like roles, commitment levels and shared mental models.

Findings

Mutual trust between contributors has significant influence on the effectiveness of a team. This trust can be strengthened by close collaboration, a mediating authority and shared ideology. Communities with low levels of trust can still effectively produce results, but have an increased need of control and review mechanisms.

The roles people hold in open source projects are usually unspecified and flexible, and most developers of the software are at the same time users of the systems they are working on. Many communities grant members of merit special committing rights, and those *committers* also gain extended voting rights in democratic decisions. Further definite roles are granted to maintainers of packages and in different levels of management (for example, the Eclipse Development Process defines a number of roles, together with their tasks [24]).

As contributors with different levels of dedication work together on a project, labor is naturally distributed unequally. Although the ratio differs for different projects, the majority of work is usually done by a small core of developers, with a large number of people contributing casually or just once. This supposed imbalance has been observed many times in studies like [39] and [30]. The core groups are able to collaborate closely due to a shared vision of the projects and a thorough understanding of the code base.

Relevance

The nature and composition of roles has an influence on the process, as do the control mechanisms that are adopted to coordinate growing or loosely collaborating groups. Different levels of dedication have to be attributed in the social structures and in the distribution of knowledge.

3.3.3 Outputs

This research area deals with the outcome and consequences of development and how to use them for future improvements. It includes means of measuring success and identifying the variables contributing to it, looks into the application areas and the context in which open source is applied and examines the progressive development over time of both software itself and the community creating it.

Findings

A number of studies have been dedicated to determine how to best measure the success of an open source project. As the field lacks the more direct measurement of commercial success, several other standards have been proposed. A commonly used criterion is code quality, although the number of possible ways to measure success suggests multidimensional qualities that should be taken into consideration as well. Those aspects of success have been shown to depend on several variables like size, ideology, type of license, openness in governance practices and other organizational characteristics, many of which influence the success of the project (see [16]).

On multiple occasions open source software has been shown to grow unusually fast in certain stages of its development due to the self-reinforcing effects of rising popularity, as demonstrated by [31]. Different code sections tend to evolve at variable rates, but structures stay stable and code duplication is naturally low. The communities of growing projects often see a conversion from a single hub to the formation of a core team and a surrounding periphery, and leadership shifts occur from single persons to committees and hierarchies.

Relevance

To understand the process of open source development it is not of immediate importance how success is measured, but it is relevant to notice which variables play a contributing role in its feasibility. The development of projects over time allows conclusions about long-term effects of the process.

3.4 Open Source Development Process

3.4.1 Overview

Creating open source software differs considerably from other forms of software development, and due to a shared culture and a constant exchange of ideas and people, enough common elements have formed that it has become common to talk about an open source development process. Although such a process did not emerge from a theoretical basis, and significant differences appear between various projects, the similarities that exist do warrant a common denomination and description.

In many ways, open source projects do not follow the ideas of traditional project management. Nevertheless, if the process of their development is examined more closely, it becomes clear that the practices employed in these projects fulfill similar functions and are used to cover many areas of project management and system development. This fact justifies a closer comparison of these seemingly disjunct approaches. Following the structure of the process overviews in the previous chapter, this section is subdivided into a description of the development cycle and a list of commonly found practices.

3.4.2 Lifecycle

Due to the open-ended nature of many open source projects, it is difficult to describe a definite lifecycle covering the whole project duration. It has even been argued that the term lifecycle might be misleading. Massey sees the open source development process conceptualized as "steady-state affair" [37]. Furthermore, it is quite common that a project's source code is only opened to the public after an initial version of the product has already been developed. Although, by definition, a project is not yet open source in this early stage of development, this phase can be seen as part of the overall open source development process if it is carried out with the intention of opening it later on. Several precautions can be undertaken at that time to facilitate this transition, as Fogel describes in [21]. Whether an open source project starts with an executable prototype, a fleshed-out product or a mere idea may have consequences for its initial positioning, but has little influence on the mechanisms carried out during its development.

In any case, a release cycle can typically be identified, which defines steps to be iterated through to contribute changes and fix problems. The following list is a sample, based on the early Apache process described in [39] and [54]. Similar variations, however, can be found in several other project descriptions.

Discover Problem

The development cycle usually starts when someone notices the occurrence of a problem or the need for new functionality. As a first step the issue has to be reported in one of the available communication tools of the project like a mailing list or a bug tracker. If the suggested change is considered important by the community, its development cycle is started.

Find Volunteers

Self-assignment for tasks is one of the most universal concepts of open source, and the continuation of the cycle depends on finding a volunteer first. However, a segmentation of the code with at least informally known maintainers exists in many cases, and if no one else volunteers for an important change, those experts in the field can try to delegate the problem or take up the work themselves.

Identify Solution

Suggested solutions for the problem are then posted back to the mailing list, where the community discusses their value and feasibility, and decides on the best route to follow if multiple possibilities have been proposed. When an agreement has been found, the developer goes to work, informing the rest of the community about it to avoid duplication of work.

Develop and Test Code Locally

Changes are generally implemented on local copies of the system and should be tested and documented by the developer before the code is sent back as a patch to the community. For simple fixes, the prior steps can be skipped and the finished patch is presented to the team for scrutiny.

Review Code Change

Although review mechanisms work differently in each project, in most cases some form of peer review happens before the code is committed back to the global repository, especially for changes in stable releases extensive review is common. In development branches it is also possible for code to be reviewed after its deployment.

Commit and Document

If the change has been approved, someone with the right to alter the code base, preferably the maintainer who was responsible for the issue, commits the change back to the repository. Methodical documentation is especially important in open source development due to the lack of personal communication, and automatic commit logs and the archived thread in the mailing list serve as additional artifacts.

Release Management

Although code can be accessed and downloaded at any given time, special release management is necessary to produce the stable versions preferred by most users. For these the project's release manager declares a code-stop, at least for a special branch designated for the release. A phase of repeated testing and bug-fixing is then initiated until enough issues have been resolved for the release to be considered stable. Frequency and timing of new releases differ considerably among specific projects.

3.4.3 Practices

Although the release cycle seems simple in its execution, in order to work effectively it requires a number of social structures and practices to be in place. The following overall view of concepts is structured to follow general areas of software engineering and system development.

Governance and Decision Making

In many cases, open source projects neither possess a central corporate authority, and do not conduct traditional project management. Still, some kind of governance is necessary for efficient decision making. The two most commonly found forms of leadership are that of a *Benevolent Dictator* or a consensus-based democracy, led by a committee, as detailed in [21]. Those two are not as different as they seem at first. In both cases control-mechanisms are in order to limit authority and control the actions of the leaders, as [40] explains. They are either set up by the project or implicitly derived from the license. The possibility for unsatisfied developers to fork the code at any given time, for example, makes rough consensus desirable in order to conserve the project as a whole. Projects often start with a single decision-maker and share the responsibilities in a committee when they grow larger, though exceptions exist where a single leader remains in place.

Like in proprietary environments, managers have to provide a vision and a common sense of direction, and coordinate efforts. The means of coordinating differ, though, as a considerable part of the available work-force consists of volunteers, who cannot be ordered, but have to be persuaded. The governing body or leader also relays authority to package maintainers and other managerial roles, often through informal appointment.

Planning and Requirements

Most projects do not conduct formal planning or detailed requirement analysis ahead of time. Requirements that arise from mailing list discussions or entries in the bug tracking software are noted in public lists and serve as a basis for development. However, the participation of commercial firms has a definitive influence on the way planning and requirements are carried out and the priority they receive.

Architecture and Design

A lot of the communication in mailing lists is dedicated to finding common grounds about the design and ways to improve it. A major concern of open source architecture is its modular structure, as noted in [36]. This division of the code into easily manageable loosely coupled parts allows newcomers to quickly get familiar with the code segment they want to work on, and keeps the complexity of the overall system low. Extensive refactoring, like it happened in the Mozilla project, can be justified and considerably boost efficiency if it helps to increase the modularity and reduce unwanted dependencies.

Testing and Review

Although testing processes differ considerably between different projects, many lack a dedicated testing team and formal testing routines. Instead, several technical and social mechanisms are in place to ensure high quality of contributed code. It is encouraged to test one's own code before committing. Also, automatic testing is in wide use, and the publicly visible nature of the release cycle allows many people to participate in a peer reviewing process. This relies heavily on the developers' commitment to the project.

Maintenance and Quality

Due to the nature of open source projects and the lack of a delivery date, projects are in a constant state of maintenance and improvement. Continuous cycles of bugfixing and new features improve the project quality gradually over time. Quality management is rarely executed in a formal way, and although the system works if the number of contributors and their participation is high enough, it can be seen as inefficient from an outside perspective, and some projects do provide commercial support instead.

Coordination and Collaboration

Coordinating a heterogeneous and voluntary group of people with a strong sense of freedom is a challenging task and requires a number of mechanisms to allow for efficient collaboration. Often, code and areas of responsibility are split into parts. This helps to keep the complexity of both code and social structures down to a manageable level. There is also a form of "microcompetition" [10] in place, when several developers simultaneously work on a bugfix. Controlling the number of people with committing rights also helps with coordination issues. Standardized procedures, general policies and guidelines help building the common vision and focus necessary for self-organizing teams. The open access to the community and its inherent diversity entail the need for effective strategies for conflict management to be devised. This requires social skills from leaders and relies heavily on the social intuitions of the community.

Knowledge Management and Documentation

The distributed and knowledge intensive nature of open source projects complicates not only coordination mechanisms, but also the management of knowledge. Because communication mostly happens online, most information is written. To organize and keep track of an ever growing body of knowledge, technological tools play an important role, as do community guidelines and a shared awareness of the importance of documentation. Nearly all communication is open and persistently available for future inquiries.

3.5 Classification

3.5.1 Approach

To evaluate how open source development relates to agile methods on the one hand and to defined processes on the other, a dual approach is pursued. It will first compare the theoretical grounds built up in the previous parts of the thesis, and in a second step relate those results to the empirical findings of the case study.

Open source presents itself as a distinct method with distinctive characteristics, deployed in a peculiar environment. When viewed in its entirety, it is neither agile nor structured. In order to attempt a meaningful comparison, it is therefore divided into a number of process elements, which are then compared individually. These disciplines are based on the characteristics of the examined definitions of process models. They are defined to cover all relevant areas of project development, yet focus on the areas in which the differences between the approaches are most apparent.

The fourteen process categories originating from this itemization are the focus of the following section. For each element a short outline is given, along with a description of how it is handled in both defined processes and agile methods. This serves as a basis to put open source development in direct relation to the other approaches, according to the classification scheme in Figure 3.1. The scale distinguishes whether there is a tendency toward either of the management styles in the particular discipline, whether open source is balanced between them, or whether it uses an approach that is significantly different from both.

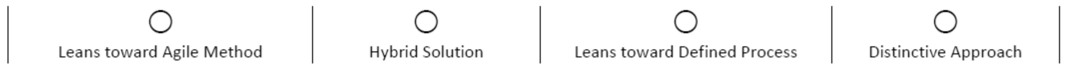


Figure 3.1: Classification Scheme

3.5.2 Comparison by Process Element

Governance

Governance involves all tasks carried out by the leading authority to direct the project and to make decisions, and the structures and hierarchies behind them. Agile projects depend on self-organization and teamwork, and their managerial roles guide the development by facilitating communication, endorsing the method and removing blocks in its execution. The hierarchy is flat, and decisions, like prioritizing and distributing tasks, are based on informed discussions that include developers and customers. Defined processes utilize a tighter organizational structure, in which managers, in addition to planning and controlling the workflow, are responsible for allocating tasks to be carried out by specialist workers. The clear separation of functions, combined with a rigid process, allows for a more direct style of leadership. Lacking those structures, open source governance relies on indirect means of control, and despite differing environments, its project leaders share many tasks and methods with agile management. Decision making, however, depends even more on the opinions and influence of the developer base, and leadership is constantly counterbalanced by democratic (or meritocratic) forms of control.

●	○	○	○
Leans toward Agile Method	Hybrid Solution	Leans toward Defined Process	Distinctive Approach

Figure 3.2: Classification of Governance

Planning

Planning describes the steps undertaken in advance to steer the project and to develop the appropriate course of action. Agile development starts with a simplified outline of the whole project and a rough estimate of important parts, fleshing out features and components more accurately only when they are ready to be implemented in an upcoming iteration. Structured processes define more elaborate plans on different levels before starting each phase and use them as a basis for dependent actions and artifacts, although they can be updated if changes occur or more information becomes available. Open source projects start with a general idea and a vision for further development, but the project's course can change considerably during development, depending on the ideas of developers and a growing contributor base. Although the influence of leaders and companies can steer the project into a direction, open source solutions cannot be planned in detail because of the freedoms guaranteed by the license and the unpredictability of volunteer contributors.

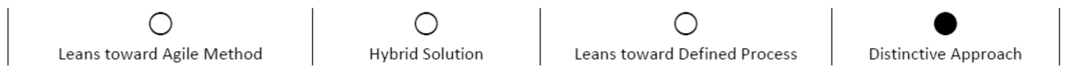


Figure 3.3: Classification of Planning

Requirements

Requirements define a project's scope and focus; finding out what they are and organizing them in a manageable pattern constitutes the discipline of requirement analysis. The initial phases of well-defined processes pay special attention to the search for requirements, they are determined in workshops and brainstorming sessions with customers and build an important foundation for the project's schedule, costs and general outline. Agile methods start with a less elaborate definition of requirements, considering scope as the most flexible control variable, which is adjusted over the project's course in accordance with the customer to reflect changing priorities, updated information and available time. Open source software also has adjustable scope, and because of less planning beforehand and fewer restrictions on schedule and costs it has an even looser conception of requirements. An ongoing discussion about what should be included in the product is held between users and developers, and feature priorities are updated continuously.

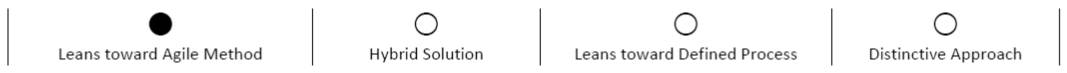


Figure 3.4: Classification of Requirements

Architecture

The architecture of a software system builds the structural skeleton for the code and incorporates design decisions about the layout and behavior of its components. In defined processes that specifically address software development, like the RUP, high priority is given to designing the architecture. It is elaborated in the early phases of development, contributing to the idea of tackling complex and risky concepts at the beginning, and serves as a stable core to which functionality can be added in a modular way. Agile methods initially start with a simple architecture and gradually scale it to keep up with what is required, constantly refining and refactoring it in small steps whenever new information about its desired form becomes available. Open source moves between those approaches, aiming for an executable version with a working modular structure early on, and improving the architecture by refactoring when the need arises.

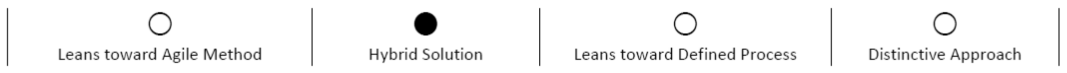


Figure 3.5: Classification of Architecture

Risk

Risk management concerns possible future events with adverse effects on project success and deals with how to avoid them or mitigate their impact. Defined processes define risk management as a separate discipline, where possible risks are collected in the early phases of the project. Then they are rated according to the probability of their occurrence, and on how grave their impact would be in such a case. The RUP is risk-driven, in that the riskiest parts of development are tackled first. Agile methods use implicit ways to address risks throughout the project lifecycle, mainly made possible by rapid feedback times and adaptable approaches. In open source communities, risks differ significantly in nature and in the threats they pose, due to different goals and definitions of project success. Therefore, these communities have also come up with different ways to absorb and avert risks, ingrained in license terms and social conventions.

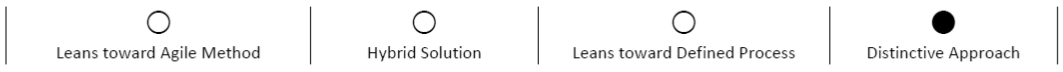


Figure 3.6: Classification of Risk

Roles

Roles give information about the skills and responsibilities of persons working on the project, as they define and categorize people's function in the team. Defined processes specify a comprehensive number of roles in all positions of project development and have people with similar roles work together on specialized tasks. In this system, people are hired and trained specifically to fulfill particular functions over extended periods of time. Agile methods use dedicated roles mainly for managerial positions, and although workers can have specialized backgrounds and focus on different areas, teams are cross-functional and tasks are chosen rather than distributed. A similar situation presents itself in open source projects, where roles are used primarily to designate administrative functions - and even there they are mostly informal and highly flexible. Team members can earn influence in specialized areas based on their expertise and commitment, and may be given authority as maintainer or coordinator.

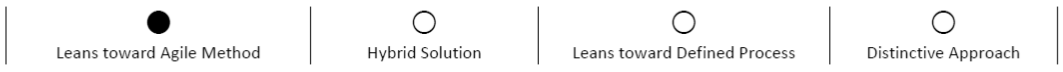


Figure 3.7: Classification of Roles

Integration

Integration describes the task of putting together the various components forming a system, as well as managing, creating and merging differing versions and code bases. Here, the approaches do not vary much, as there is a common tendency to integrate often and to keep divergences to a minimum, but variations exist because of the restrictions of different environments. Agile methods benefit the most from their small size, close cooperation and low organizational overhead. This allows for continuous or even synchronous integration, where versions are kept uniform in a single code base. Modern structured processes aspire toward the same direction, but due to a slower iterative cycle and a larger number of people involved in it, different versions and components are kept up and integration exists as a definitive step in the lifecycle. A similar situation can presumably be found in open source projects. Different versions exist and survive at any given time at the developers' workplaces, and although changes are incorporated into the main project frequently, a number of code bases, like stable and nightly builds, are managed simultaneously in an ongoing integration process.

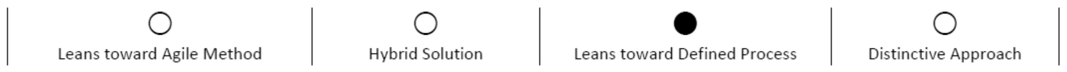


Figure 3.8: Classification of Integration

Costs

Cost management deals with money as a resource and covers the steps necessary to prepare a budget, to estimate tasks and to control spending in order to be finally able to adhere to its limits. Agile methods consider the budget as a given, which is fixed outside the project, and use it to align the scope to the available funds. Estimates concerning the time necessary for task completion are sketched roughly at the beginning and are refined within the team throughout the course of the project. Although defined processes specify more concrete means of estimating, and deploy them more rigorously in the planning phase, the general principles are similar, in that cost management mainly deals with estimating tasks and increasing their precision. Approaches in an open source environment differ profoundly from both; because of a mostly voluntary workforce and no direct income from product sales, money is less significant to the development process. Money can be used to offer an incentive to do unpopular work, as a way for a company to buy influence or to pay for marketing and server upkeep.

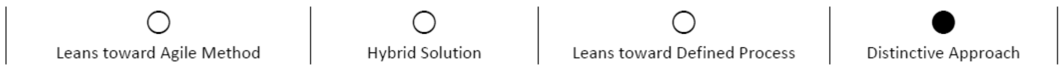


Figure 3.9: Classification of Costs

Testing

Testing consists of a wide array of practices carried out to ensure and increase the quality of written code, making sure the product fulfills its intended functionality reliably and with low rates of defect. Agile development places testing close to coding, since it is carried out both at nearly the same time and by the same group of people. It employs frequent and automatic testing, often utilizing test-driven development, to increase the chance of identifying and fixing defects early on. In defined processes, testing is a self-contained discipline carried out by a dedicated team of specialized testers, who follow a formal testing plan and review finished code to ensure a high coverage of possible areas of defect. After the release of the first beta version, direct user input drives further tests. Open source uses a mixed approach for testing, combining different approaches and taking advantage of the large number of possible contributors. Submitted code is ideally subjected to both peer review and automatic testing, and projects are deemed in a state of beta testing during their whole lifecycle, relying on user feedback to locate and fix problems.

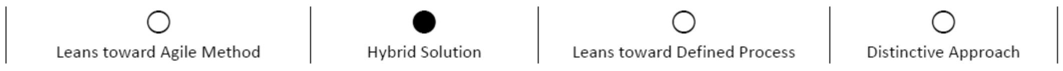


Figure 3.10: Classification of Testing

Knowledge

Knowledge management concerns the ways information in the project is collected, processed and distributed among stakeholders. In agile development, knowledge is implicitly shared among management, team members and customers. The need for elaborate documentation is reduced due to personal communication and close collaboration in a shared environment. The formal process utilized by well-structured approaches defines a set of artifacts, which are gradually refined using the information at hand. They serve as a baseline for the workflow and provide stable and visible data for everyone it concerns. Open source projects have similar needs to make information transparent and available for a large group of persons. Although their documents tend to be less formal, they too provide a reliable common knowledge base for stakeholders and are used extensively to synchronize and coordinate development efforts.

○	○	●	○
Leans toward Agile Method	Hybrid Solution	Leans toward Defined Process	Distinctive Approach

Figure 3.11: Classification of Knowledge

Cooperation

Cooperation affects all parts of development, dealing with how people collaborate and what is done from a management perspective to help coordinate effective teamwork. A prominent characteristic of agile development is the close collaboration of a small team in a co-located environment, using face-to-face communication to increase efficiency. Practices like XP's pair programming, the Daily Scrum and shared ownership over a single code base all further these means. Defined processes, on the other hand, pursue division of labor to parallelize activities in an efficient way. Code is owned by individuals, and specialized teams are responsible for each part of the development. The distributed nature of open source projects presents an environment which is different from both and has originated its own mechanisms. It uses extensive communication, aided by technology and a pronounced sense of shared identity, to coordinate a large group of relative strangers and establishes trust to work on common goals.

○	○	○	●
Leans toward Agile Method	Hybrid Solution	Leans toward Defined Process	Distinctive Approach

Figure 3.12: Classification of Cooperation

Tools

Tool support is an auxiliary factor with influence on all stages of development, and the use of technology can profoundly alter project flow and communications. Agile development is designed for speed and efficiency at close quarters, and while it utilizes tools for automation purposes, it prefers “low-tech/high-touch” [19] techniques like story cards on post-its and whiteboard diagrams to visualize and organize project flow. The processes used in structured approaches, however, use a full spectrum of tools to support development in a synchronized and accountable way. The RUP as a whole can be seen and is delivered as a software product providing tools for all its disciplines. Open source projects, due to their highly distributed character, also rely heavily on technology for communication and structure, and utilize a wide and diversified palette of tools in similar ways to synchronize and aid the effort of their developers throughout their lifecycle.

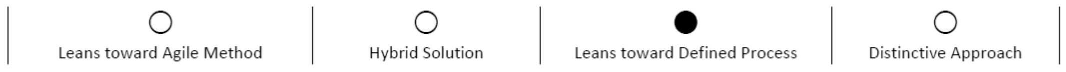


Figure 3.13: Classification of Tools

Release

Release management controls the flow of development, defining how often new versions of the software reach the users and how the project’s lifecycle is structured. Agile development methods encourage that a runnable version is created early, and that new releases follow frequently thereafter, with XP aiming for daily deployment to minimize differences between the versions used by developers and customers. Defined processes allow more time between releases and usually do not deploy until the project has reached a beta-stage with almost full functionality. From there it is improved through bugfixes and performance-tuning until it gets accepted by the customer as a final version. Current development versions of open source software can typically be downloaded and can be run at any given time. In addition, though, those projects often follow some kind of a release process to create stable versions for general use.

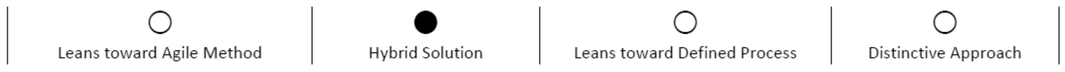


Figure 3.14: Classification of Release

Maintenance

Maintenance is the ongoing attendance of a system in use, providing a way to interact with users to supply them with information and support. It allows fixing problems surfacing in the running software. Agile methods know no clear demarcation between maintenance and development, as they directly involve customers in the project from the start. Problems with operating the program are faced whenever they arise. In defined processes maintenance is a separated discipline with clear specifications, mainly carried out during the phases of beta testing and thereafter and ended by achieving acceptance with the customer. Open source projects utilize user feedback as its driving force and exist in a constant state of maintenance, where every user is a potential beta tester to improve system functions and spot its weaknesses. User support is either achieved by peers helping in forums or outsourced as professional paid training.

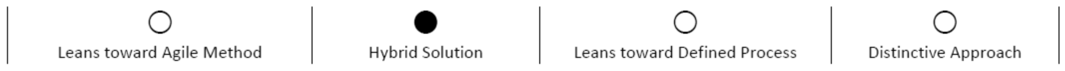
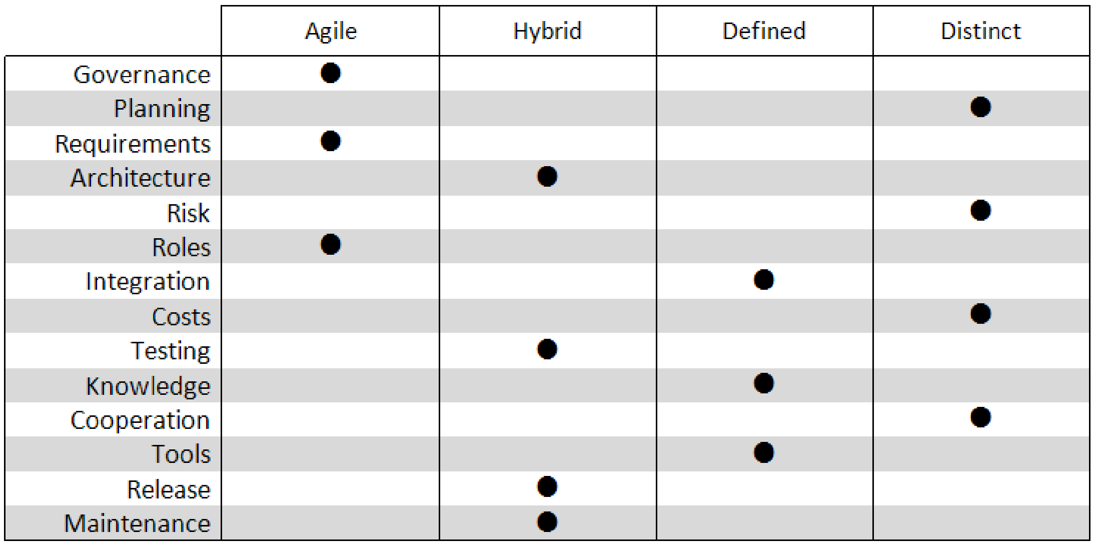


Figure 3.15: Classification of Maintenance

3.5.3 Results

Based on process descriptions, research on open source projects and the defining characteristics of their shared environment, the theoretical comparison has brought out varied results. Figure 3.16 provides an overview of the reviewed elements and their classification. It shows an even and widespread distribution of tendencies, which confirms the previous assumption that open source shares similarities with the compared approaches in some aspects, while differing from both in other areas. However, these results are preliminary and will be refined and updated with the information gained from modern projects through the means of a case study, which is described in detail in the following chapter.



	Agile	Hybrid	Defined	Distinct
Governance	●			
Planning				●
Requirements	●			
Architecture		●		
Risk				●
Roles	●			
Integration			●	
Costs				●
Testing		●		
Knowledge			●	
Cooperation				●
Tools			●	
Release		●		
Maintenance		●		

Figure 3.16: Overview of Classification Results

CHAPTER 4

Case Study

"Many people think that open source projects are sort of chaotic and anarchistic. They think that developers randomly throw code at the code base and see what sticks." - Winifred Mitchell Baker

The second half of the previous chapter provided a general overview of the elements typically found in open source development, based on the available literature, popular early projects, and the natural constraints and possibilities of the open source environment. Within the case study, which was carried out in the course of this research, this data was complemented with updated knowledge gained from several contemporary cases.

What the case study is trying to achieve, and how it addresses its goals, is the main focus of the methodology which constitutes the beginning of this chapter. The next section describes the criteria for process selection and gives a report of how the three projects of the case study were selected. Each of these projects is then addressed in detail in the main part of the case study, which includes an overview of the projects in general, a report of the specifics of the case study approach, and a thorough examination of the development process as attained by developer interviews and the study of the project documentation.

4.1 Methodology

4.1.1 Aims

Before the case study and its methodology are described in detail, it will first be established what exactly the research aims to achieve. There are a number of different aspects in which the review of secondary data is insufficient to answer the research questions, or in which further conclusions can be drawn from direct contact with open source developers. In general this case study serves as a reality check. It looks at complete instances of the process as it is found in the examined projects and helps forming a well-rounded picture of the organizational culture of open source environments.

Actuality

Open source development, like most sectors of information technology, is a fast moving and ever-changing environment. This leads to a noticeable gap between secondary data from published literature and the reality of development in action. In particular, the available literature hints at several trends and changes in the process. One of the aims of the case study is to pick up these trends and to observe if they can be found in modern examples, and how they have influenced the nature of open source projects in general. The data gained from the case study then presents an updated view of the development process that takes modern structures and practices into consideration and shows a snapshot of present-day open source development.

Variation

A large part of the research that contributed to initially formalizing the open source process was based on a small sample of very specific projects. While early and popular agents of free software like Linux and Apache have played a crucial role in the rise of open source, the methods used in their creation may not, at least in some aspects, be representative for the majority of contemporary projects. The relevant question in this case is, however, not only how the process changed over time, but also how and in which parts of the development these modern projects differ from each other. The case study aims to explore where variations can be found, and which practices exist in a common form even under diverging conditions. It also tries to find out how much the development process depends on the type of the project and its environment.

Details and Dependencies

The general process descriptions, which constitute the foundation for the theoretical analysis in the previous chapter, cover different aspects of development to a varying degree. To be able to reliably categorize open source development processes and to compare them with those of the different proprietary approaches, a closer examination of several parts of the process and structure is necessary. It is not enough, however, to look at those process elements in isolation. As it is likely that the different tasks of development are dependent on each other as well as on the environment they are employed in, it is important to view the projects as integrated systems and to focus on how the elements of their processes interact.

Motive and Perspective

An important goal of the case study is to collect inside information, based on first-hand experience and showing the perspective of those developing the projects. Additional insights concerning the background and motives for the process design can be gained by supplementing the theoretical data with direct input obtained from committed members of open source communities. Also, by determining the influence of the project's culture on the chosen process models and practices, these choices can be rationalized more clearly. Personal assessment by experts in the field can therefore contribute to a well-balanced view on the open source process and is decidedly one of the specific aims of the case study.

Relevance and Practical Realization

Theory is often based on ideal cases and matters of principle. Observing projects in action and leading conversations with those most invested in them can shed light on those circumstances where the workflows in practice differ from the ones propagated by abstract definitions of processes. This is especially important and becomes a matter of investigation when the constraints of reality alter the execution of the process permanently, leading to deliberate changes and special tailoring. One final purpose of the case study is therefore to examine the influence of practical project characteristics and execution on the adaptation of processes, and how relevant those remain to be in everyday development.

4.1.2 Case Study Instruments

In order to gain a thorough and well-rounded perspective of the projects and the processes they employ, the case study utilizes a two-sided approach, which combines the study of available project documentation with interviews led with knowledgeable insiders working on the projects. How those two methods are planned and carried out is the focus of this section, while the reasons behind the choice for this particular setup in favor of possible alternatives are explained thereafter in a short chapter describing the case study rationale.

Project Documentation

Two of the characteristics of many open source projects are a tendency to provide elaborate documentation and a desire for transparent and open processes. Where these principles are combined and applied diligently, this documentation presents a rich source of information about many aspects that are relevant for answering the research questions at hand. The background knowledge that is gained through the study of the process as it is described on the projects' websites can on its own provide sufficient material to understand parts of the way development works in the project, while it simultaneously serves as a sound foundation for interviewing the developers. In the latter function the precursory research supports the subsequent interviews both in the preparation phase and in the analysis of the results.

Although it would have been possible to compile uniform questions for all case study objects by basing them solely on the general knowledge of process models and the open source environment, tailoring the interviews around the internal process records found in the projects' documentation resulted in several advantages, which are further detailed in the chapter on the method's rationale. Because not every project provides process information at the required level of thoroughness, the availability of detailed and up-to-date descriptions of the project's development methods has been chosen as a criterion for project selection in the present case study. In most cases, however, a larger body of written knowledge comes with the cost of an increasingly complex documentation structure, along with a concurrent increase of information not or only marginally relevant to the research topics. Therefore the bulk of possibly relevant process documentation had to be sighted, filtered and processed before the interviews were conducted.

Qualitative Interviews

Although examining the project documentation was an important part of the case study, the main source of information were the interviews with developers. To ensure that their results would be of high quality, they had to be carefully planned and carried out. The interviewing process was influenced by the approach utilized by Herbert and Irene Rubin in [47], who included responsive elements into their qualitative research in order to be able to adapt the interviews to fit specific fields of application and to react to the answers of interview partners. Following these considerations, the method was designed to incorporate the following attributes:

Qualitative

The first and most important step in planning the case study was the decision to base the collection of primary data on qualitative interviews. These were targeted at committed developers and major contributors of open source projects, and were designed with the intention to provide a conclusive inside view into the process, and to shed light onto the practices employed and how those practices interact with each other. Personal experience and expertise of the interview partners helped in the understanding of complex correlations and peculiarities of the process, and their familiarity with the specific culture facilitated putting those process elements into the right perspective.

In-Depth

In the matter of balancing the case study's scope and width it was decided to examine each targeted project comprehensively and in depth. Although this entailed a long and detailed preparation phase, it was deemed important to do so in order to thoroughly understand the process of open source development, and especially to grasp the way this process is deployed in modern real-world applications. However, this meant that a trade-off had to be made by limiting the size of the sample in order to stay within the time-constraints of the study. While this imposes some limitations on the universality of results (cf. Rationale & Limitations), a thorough and well-balanced insight into the researched projects is crucial for the purpose of the study and outweighs these limitations.

Semistructured

An important point of consideration concerns the structure of the interviews, specifically how detailed it is planned upfront, how flexible it is and how it is used to help focus the conversation on the research questions. In this case study a semistructured approach has been chosen, in which guidelines for the questions and the topics they cover exist, but no specific wording or sequence is prescribed. The researcher used a checklist of topics and corresponding potential questions, which was based on the process areas mapped out by the previous chapter and incorporated specific adaptations to fit the particular projects. Several broad exploratory main-questions were devised, which were complemented by more specific follow-up questions to fill the gaps.

Responsive

Another measure that was undertaken to get the most out of each interview was to follow a responsive approach. Entailed by this style of questioning is an ongoing adaptation of follow-up questions to previous answers of the conversational partner, with the goal of covering the proposed set of topics in a targeted and efficient way. This method works well in combination with the semistructured format described earlier, and is further assisted by providing the interviewee with an outline of topics in advance. The responsive character of the case study was somewhat limited though, as only a single session was held with each interview partner and the time frame for each of the conversations was limited.

Remote

As most established open source development communities are distributed globally and the Internet is their primary means of interaction, it came naturally to use internet-based forms of communication for the interviews. Although this method has some drawbacks compared with face-to-face conversation, it meant that the search for potential candidates was not confined on a specific regional area, but could build on a global pool of contributors. It was left to the choice of the interview partners if they preferred a spoken conversation or a written format like chat or email.

Realization

After the preparation phase and in addition to making the mentioned design decisions, several practical steps had to be taken in order to actually realize the case study. In a first step, a list of potential candidate projects had to be made and narrowed down, a process which is further described in the section on Process Selection. The developers of those projects which met the required criteria were then contacted in a standardized email, asking for their participation in the case study. Furthermore the email both provided details about the study and its research topics and informed about the format and practical procedures of the interview. In those instances in which the initial contact elicited feedback, an appointment was made and the interview was carried out. A recording was made with the compliance of the interviewee to facilitate the following analysis and ensure a faithful account of their position.

Analysis

A first preparatory step in analyzing the results was to transcribe the recorded conversations, and compile and sort the parts of the email-based interview into a more easily readable form. These written records can be found in the appendices A, B and C of the thesis. For the purpose of the research they were then analyzed and coded using the process categories established in Chapter 3.5. Wherever the data provided by the interview lacked detail or needed clarification, the project documentation was consulted for additional information. This resulted in a structured collection of data for each project of the case study, which was then included in the thesis in two ways. Examined separately, the data of each case was used to describe the development process of the particular project to form the process reports at the end of this chapter. Compared by process element, and incorporating the theoretical data of the two previous chapters, the data constituted the foundation for answering the research questions in Chapter 5.

4.1.3 Rationale & Limitations

The following section explains why the present approach has been chosen, and it justifies the choice by giving reasons why, for the particular field of study and the present aims, it is more useful than alternative methods. To present a balanced view and give ideas for future research, it furthermore describes the limitations of the case study in its current form and execution.

The consideration that carried the most weight was the decision to use qualitative interviews as the main element of the case study. It would have been possible, instead, to base the research solely on quantitative data, extracted from one or more of several possible sources. In the meta study conducted by Crowston in [18], more than half of the studies in the sample were “based on archival data retrieved from development repositories.” Another alternative would have been to send out surveys to the projects with standardized questions about the process. Those and similar approaches share the advantage of providing numerical and comparable data, and using them facilitates collecting and analyzing a large sample more efficiently. Nevertheless these alternative methods were finally rejected, because the type of information gained by applying those methods would have been insufficient for the research topics at hand.

The aims of the case study required a thorough view on the sample projects in order to understand their development processes well enough to compare them with other process models. In order to obtain this view it was necessary to gain insight into culture, background and conventions of the examined open source communities. To achieve this it was decided to go directly to the source and conduct interviews with people who are committed in the development of these open source projects. Furthermore, dedicated insiders would give depth to the process data by providing examples from their personal experience, expert assessments and information about the development of the project over time.

Using available project documentation as a foundation for these interviews followed from several considerations. First, it helped phrasing specific questions to be able to proceed beyond discussing common knowledge in the limited time frame of the interviews. Also, in the subsequent analysis of the conversations the additional information could be used to prevent misunderstandings, to put the given answers into broader perspective and limit personal bias. Some parts of the processes could be reconstructed almost entirely from available documents, mailing list archives and blog entries, leaving more time and space to focus on those aspects that were less documented or more complex.

Along similar lines of reasoning followed the decision to cover the observed projects in depth, forgoing a larger number of participants in favor of detailed analysis of each item of the sample. Spending less time on preparation or analysis, or limiting the interviews to a narrower scope would have compromised the quality and explanatory power of the results, even when considering a larger sample as a balancing factor. Separately examining several elements of the process increased the complexity of the research, but allowed deeper insights into the development lifecycle.

The in-depth analysis and the focus on a smaller sample led to one of the more obvious limitations of the case study. It does not lie within the possibilities or the intentions of the research to deliver a representative view of how all open source projects are developed nor does it provide a numerical comparison of how common and how pervasive certain development practices are in the overall ecosystem of different approaches. It rather showcases possible scenarios of how the process can be employed and successfully be in effect in a selection of model projects. Within these constraints, however, the study was designed to give insight into a range of environments showing varying solutions depending on the type of the software and the culture established by the communities. While leading to first-hand information and insights from domain experts, basing the research on developer interviews also put limitations on the method. Data gained in individual conversations is naturally influenced by the personal position, knowledge, interests and opinion of the interview partner. This had to be considered while setting the study up and carrying it out. Therefore the information obtained in interviews was backed up with additional sources wherever they were available.

4.2 Project Selection

4.2.1 Overview

Before the interviews could be carried out, suitable candidates for participating in the case study had to be found. Starting from a comprehensive list of projects, a number of criteria have been applied to limit the list to the most promising cases. An invitation to take part in the research has then been extended to the developer communities of those projects. The following section focuses more closely on several points of the selection setup and execution. It starts by giving a description of the criteria which have been applied to find suitable projects, followed by a report of the actual selection process, and ends with a short overview and factual comparison of those projects which finally constituted the participating set of the case study.

4.2.2 Criteria

Project Information

For several reasons it was important to restrict the case study to projects with well-developed documentation and sufficient information about the project in general, as well as with some specific data describing its development process. Most of these reasons are laid out already in the methodology and are directly concerned with supporting the interviews with complementary knowledge. In addition, the coverage and state of the project website was used as an indicator for the overall maturity and transparency of the process. Although basic information and user documentation were present in most cases, the availability of comprehensive and well-organized data aimed at developers was used as a meaningful selection criterion.

Possibility for Participation

Another detail that was specifically looked for on the websites of prospective cases was instructive information about how to contribute to the projects' development. An area which presents various ways of contribution and is easily accessible to newcomers to the project was considered a necessary requirement for inclusion in the case study. This was justified by the consideration that the possibility to participate in the process is a major indication of the project not only being open source in a purely technical, legal sense, but that it was developed by an open community using an inclusive process, which in turn was required for staying inside the scope of the research. It is important to emphasize this point, as roughly one half of surveyed projects did not present this information in a sufficient way.

Active Development

In addition to examining the static information displayed on the project's website the selection process also entailed looking for signs of active, ongoing development of the project. Points that were taken into consideration when determining the status and progression of code development included the patch-history, updates in the repository and, if present, roadmaps for future releases. Measuring the activity in the project's mailing lists and public forums was used as an indication of general activity of the community, as was the existence and actuality of developer blogs and newsfeeds. A further selection factor was ensuring that the project was well-established and has reached a certain degree of maturity, as indicated by the release history, project age and available information about the project's past and present development.

Size and Popularity

Determining size and popularity of an open source project is no trivial task, as both depend on a multitude of different factors that are often hard to establish and can be largely independent of each other. Using available information on the official websites and on *ohloh.net* [55], which features detailed statistics of many open source projects, rough estimates about the size of the communities and the length and complexity of the code itself were made. Many projects were found to be based mainly on the work and dedication of individuals and very small teams, and were excluded for their obvious lack of necessary cooperation and governance. On the other hand, highly popular and much-researched projects, like the many distributions and derivatives of Linux, were also excluded from the sample, since information about their development already builds the foundation of the theoretical knowledge of open source processes.

Diversity

The criteria which were listed so far only consider the characteristics of individual projects. It was, however, also an issue how the chosen projects would relate to each other in various aspects. While all participants in the case study would have to fulfill the given criteria, they should also show a certain degree of variation in terms of structure, business model, product type and size. Contrary to the other criteria, which could be used to filter the list of projects before attempting to contact them, the criterion of diversity depended on the state of the case study, and had to be taken into consideration throughout the process to adjust the search in order to look for projects that would complement the research in the best possible way.

4.2.3 Selection Process

An online list found on the website of the magazine t3n [67] formed the starting point for the selection process. Being comprised of roughly 350 popular open source projects and giving basic information and links, the list provided an adequate basis for further selection. Since the purpose of this initial screening was to find a small number of fitting projects for the qualitative case study, it was of no particular concern if the starting list was complete, or if its selection was representative. By visiting the websites of the listed projects and applying the criteria laid out in the above description, the number of prospective projects was reduced to a sample of approximately 60 cases. In a subsequent step, the online statistics of the remaining projects (as found on ohloh.net) were used to select a smaller subset of 15 entries by comparing the available information on commits and project size over time.

After projects with the necessary characteristics had been selected, they were contacted one at a time per email, on appropriate channels like the developer mailing lists. Based on the needs for diversity, the list of potential projects was meanwhile gradually adjusted in terms of number and order of candidates. Emails were written to about ten projects, until enough project developers agreed to participate in the research. This led to the final selection of the three projects constituting the case study.

4.2.4 Project Outline and Comparison

The case study is comprised of the projects *Django*, *Drupal* and *XWiki*. They are each described in further detail in the following sections, along with a report of the approach used for the respective interviews, and an aggregation of the facts gained about the development processes applied in the projects. Some basic facts about the projects are collected in the factsheet in figure 4.1. All three fulfill the established criteria. On their websites they provide ample information and developer documentation, and describe different ways to join the development as a contributor. Also, they are actively developed and have lively communities and regular patches. In order to compare the size of the projects, several factors were taken into consideration, such as lines of code in respect to programming language, number of core developers and contributors, user base and the traffic on their preferred online communication channels. Still, this provides only rough estimates which place Django and XWiki quite similarly as medium-sized projects, and Drupal in a larger category, mainly due to its massive collection of user-contributed modules and the people maintaining them. Although they are similar in many aspects, the projects show significant differences in their social structures and design decisions, as will be demonstrated in the following process descriptions.

	Django	Drupal	XWiki
Product Type	Web Framework	CMS	Wiki
Founded In	2005	2001	2003
License	BSD	GPL	LGPL
Language	Python	PHP	Java
Examined Version	1.4	7.14	4.0
Project Website	djangoproject.com	drupal.org	xwiki.org
Size Estimate	●●●○○	●●●●○	●●●○○

Figure 4.1: Comparative Factsheet

4.3 Django

4.3.1 Project Overview

Django is an open source web-framework, which is based on Python and facilitates the development of dynamic Internet applications. It originated as an in-house project at the “Lawrence Journal-World”, a local newspaper company situated in Kansas. As the programmers describe their situation in [22] and [27], working in this “fast-paced newsroom” environment required the construction of new web-projects under a tight schedule and “extreme deadlines”. To manage the workload and automate the production in an efficient way, developers Adrian Holovaty and Simon Willison started developing reusable components for their projects, switching from PHP to Python in 2003. Over time they created a framework and used it to power most of the company’s web-applications. Together with Jacob Kaplan-Moss, who had joined the development team, Holovaty then decided to release their work as Django, opening the source code to the public under a BSD license in the summer of 2005.

Following the publication of the code, the project grew in popularity and managed to attract a large circle of users and contributors, gradually increasing in size and maturing to a versatile product. Holovaty and Kaplan-Moss stayed on the project, guiding its progress as Benevolent Dictators For Life, but the development effort spread to a larger team of volunteers, and is now centered around a core team of twenty to thirty developers with committing rights. The target audience consists mainly of web-developers with a background in programming with Python, leading to a significant overlap between users and contributors. Since 2008 Django is maintained by the Django Software Foundation, a non-profit organization set up to promote the project and support it financially and legally.

The framework consists primarily of a collection of libraries that provide the functionality to rapidly develop web-applications in Python. Although the terminology in the project is different, Django roughly follows an MVC (Model-View-Controller) architectural pattern, splitting development into models, templates and views, with loose coupling between those parts. The stack furthermore features a database wrapper, providing object-relational mapping and a basic CRUD-interface, as well as a URL dispatcher and an optional self-generating administration interface with automated validation. Other characteristics include internationalization assistance, a caching system and authentication support. At the time of writing the project is in version 1.4, with a new minor version being released roughly every nine months. Besides powering the websites of many major and minor newspapers, Django is used in a wide variety of web-applications, ranging from websites at NASA to high-profile startups like Pinterest and Instagram.

4.3.2 Case Study Approach

The conversation about the development process of Django was the first interview in the case study. It was led in the form of a Skype call, lasted about 35 minutes, and is available as a transcript in appendix of the thesis. The interview partner was Dr. Russell Keith-Magee, an Australian developer who has been in the core development team of Django for the last six years. He maintained the release process for the project's 1.2 and 1.3 release cycles, was involved both in several internal refactorings and the creation of the test system, and is also the President of the Django Software Foundation. During the conversation, the interviewee provided a detailed overview of the development process and the structures behind it, elaborating his account with examples and personal assessments. The researcher acted mainly as an impulse-giver to ensure an equal coverage of the relevant topics.

4.3.3 Process Description

The development of Django is structured in several release cycles of different scale. Major Releases bring ground-shaking changes but happen very rarely, So far only one Major Version (Version 1) has been released. About every nine months, a new Minor Version is scheduled to bring new features and major improvements to existing ones. The time frame between Minor Releases is roughly split into three phases. In the first phase proposals for new features are accepted and decisions are made on which of those features to include in the release. The second phase is dedicated to the development, and the last is reserved for necessary bugfixes. Between every two Minor Releases one or more Micro Releases can be carried out to repair important bugs or fix security issues. There is a strong concern for backwards compatibility, so code is designed to keep working between new releases, and if a feature is flagged for deprecation, it is still supported for two full Minor Release cycles, giving users 18 months to move to newer functions. Because the framework exists as a library of different components, there is an inherent modularity in the architecture, facilitating the overhaul or replacement of single features and, when following the deprecation process, the transition to new solutions. According to Keith-Magee there have been some major reworks in the early stages of development, but the core architecture has been "almost completely stable" in the last four years. This ongoing stability is seen as a source of the popularity Django has earned among large organizations.

Bugfixes are not only implemented in the latest version, but also backported to the previous release. In addition, important security patches are retrofitted to the previous two versions. This leads to the existence of three concurrently active branches in the main repository, although the majority of the development effort effectively happens in the trunk, where the most recent version is located. Further branches exist for large, officially mandated projects like the contributions made for the Google Summer of Code, in which Django regularly participates. Generally, though, branched development plays a secondary role, as most other features tend to be developed outside the official branches of the project and are later handed in as patches, once they have been advanced sufficiently. Anyone can develop a patch and propose its inclusion in the main project. In order to be accepted, however, it has to fulfill several criteria. Notable among these criteria is that the code has to be sufficiently documented, and all significant areas have to covered by tests.

Testing plays a major role in the project, and according to Russell Keith-Magee the framework was very heavily tested from the start. One important aspect of the testing process is the reliance on automatic tests. At the time of the interview the test suite had about 4,500 tests on different levels, including unit-tests, view-level tests, integration-level tests and browser-tests. These tests are run against actual databases, and the duration of executing the tests depends heavily on the type of the database. Before any code is accepted into the main repository, it has to be reviewed by at least one additional developer, and more reviewers are required for larger or more complex features. Because only core developers have the necessary rights to check-in changes, they are considered to be “the last line of review” and are responsible for getting the problem fixed if the new code breaks the system or causes tests in the test-suite to fail. The intended purpose of these measures is to keep the trunk essentially stable at all times. Although it is usually not recommended, checking out the most recent build of the code should work at any given time. The commit process in general is elaborate, but informal and depends on the judgment of the developers, especially that of the core committers.

Different tools are used throughout the process to automate workflows, to facilitate the communication between the people involved in the project and to allow the coordination of their efforts. Mailing lists and an IRC-channel are used to exchange knowledge, discuss multiple approaches and to socialize. A central hub of cooperation is *trac*, which is used as an issue- and bugtracker. The development team compares the use of the bug tracking system to a “community garden”, where every developer is allowed to create and modify tickets and can do their part in assessing and processing issues. People work together to identify priorities and try to agree on the best course of action. Discussions of a larger scale are moved to the mailing lists in order to find a consensus. For their written code to be included into the project, contributors have to convince the other developers, especially the core team, of its significance to the project as well as of the fact that their chosen approach works best. No individual or group in the project has the power to force others to implement a feature, which means that developers have to either implement their ideas on their own or need to try convincing others in the community of the importance of the feature and the benefits resulting in its implementation.

As personal initiative and dedication are main factors for earning responsibility, the project is based on a mostly informal role system. Some formal roles with specific functions do, however, exist in the project, like the Release Manager, who is officially tasked with the necessary steps to cut a new release, and the BDFLs, who are the founders of the project and whose opinion on the course of the project has special weight in the community. This fact is represented and formalized by, among other things, special decision rights and the final say in conflicts of opinion. A major distinction that has some far-reaching implications is typified by the degree to which developers have write-access to the main repository. There are a number of domain experts who have full access to their special area of expertise but no additional political rights or responsibilities. Members of the core team on the other hand, who have committing-rights throughout the whole project, are responsible for the code that is checked in, and have voting rights that influence the project flow. New committers are therefore selected carefully by existing core developers to represent the ideals and "design ethics" of the project, which is one of the reasons why the core team stayed relatively small, with about a dozen or less active committers at any given time. Having code internals shared by too few developers can pose a threat to the development of the project though, as those individuals could leave the project or become inactive. Further potential risks include fundamental security problems or the concurrent departure of several important decision makers. The fact that the project does not have a formal disaster recovery plan was explained by Russell Keith-Magee in his interview. He emphasized the resilience Django, and open source software in general, have against catastrophes:

> "If the worst happened - if every single member of the core team got hit by a bus, and the Django website went down, and all the worst things that could possibly happen to the code happened to the code - you would still have the code. And it would be sitting in thousands of repositories around the world, and someone would be able to find a way to get the most recent revision up somewhere, and I'm sure there would be enough of the community around to start developing the next round of patches, the next round of updates, and so on."

The above quote also provides some indication of how knowledge is shared in the project and in which form it presents itself. In essence, there are two sources from which detailed information about how the code works can be gained; one is the direct contact with the community, the other one is the code itself and its surrounding documentation. By constantly communicating via transparent and public channels, information is spread among many members of the community. According to the interviewee it would, however, also be possible to reconstruct the essentials of the project from written sources alone, due to extensive documentation and good readability of the Python code.

Because of the voluntary character of most contributions to the project it is not possible for anyone to give orders to others, which causes what Russell Keith-Magee calls "perhaps the biggest culture shift" compared to other forms of developing software. One implication of this is that there are no detailed plans for the long-term development of Django, instead it is guided by a general idea of where the project should be heading. This idea is defined by commonly established rules and a culture shared by the developers. Whenever decisions have to be made, it is usually first attempted to find a consensus by informally discussing a respective issue in the community. In some cases, or if no agreement can be found, the core committers cast a vote using Apache style voting rules. Should this indicate dissent among the core team members, it is up to the BDFLs to intervene and make a final decision. Legal representation, fundraising and promotion of the project are in the hands of the Django Software Foundation. Budgets and monetary issues, however, are of little significance to volunteer-driven development, therefore the DSF's concerns are widely separated from the development process itself. The interviewee pointed out that it is an important requirement for the management of open source projects to keep its developers happy and interested in the tasks at hand.

4.4 Drupal

4.4.1 Project Overview

Drupal is an open source content management system (CMS) and a platform to create dynamic websites. It features a large collection of prebuilt components and provides tools to create custom solutions. In the beginning, however, it started as a small project among students, who created a simple message board and news site to communicate with each other in their dorms. The founder of the project, Dries Buytaert, kept the site up after his graduation and opened it to the Internet. There it attracted a larger group of people, who used the platform for discussing their ideas concerning new web-technologies. The community began implementing these ideas, using the project as an "experimentation environment". In 2001 the software that backed the website was released under a GNU license and also adopted its final name, Drupal.

This publication led to further growth and diversification of both code and community, which resulted in an organizational split between Drupal Core, the central component that provides basic functionality, and a large universe of custom modules. The core is developed under the supervision of a small team of committers surrounding Buytaert, who keeps on leading the development of the project as its BDFL. The modules are maintained by members of the community, and although those modules share the main project's infrastructure, they are autonomous in several aspects and can be added to an installation of the system individually or in pre-made distributions. With more than 18,000 developer-accounts and 840,000 signed-up users (as listed on the project homepage [12]), Drupal builds on a broad and closely connected community. It is furthermore backed by the Drupal Association, an educational non-profit organisation, and a large network of profit-oriented companies providing support and additional services.

The targeted audience for the project consists of web-developers and designers with different levels of expertise. Because of the graphical user interface, no special programming skills are necessary for installation, administration and the creation of basic websites. By installing Drupal Core, users get essential CMS functionality like user administration, news aggregation and a publishing workflow, and they can use it for discussion and commenting purposes. This function range can be extended by adding modules from the "Contributed Space", or by writing or changing extensions to adjust the solution to fit specific needs, using PHP. Drupal is used for a wide variety of different web-projects and powers about six percent of CMS sites and two percent of all websites worldwide, according to Buytaert in his keynote for the DrupalCon 2012 in Denver [13]. Notable operators include *ubuntu.com*, entertainment nodes like *mtv.co.uk* and prominent government offices, including *whitehouse.gov*.

4.4.2 Case Study Approach

Like the previous interview, the case study on Drupal is based on a 35-minute Skype call with a dedicated contributor. The transcript of the conversation can be found in appendix B. Klaus Purer, who agreed to take part in the research, works as a web-developer for a company specializing on Drupal-based solutions, and has been an active member in the Drupal community since 2008. In this time he helped maintaining several modules, provided patches for Drupal Core, and acted as a mentor for new developers. He has also been actively involved in the project reviewing process. Contrary to the other interviews, the conversation was held in German, as this is the native language of both researcher and interviewee. The intention behind this decision was to allow for a more fluent and natural conversation. In comparison to the first interview there was a more active involvement of the researcher, leading to a dialog-like exchange through several phases of the interview. The conversation started with a general overview of the process and the organizational structure of Drupal and moved further from there to cover more specific details and examples of the development and the methods applied in it.

4.4.3 Process Description

At the current rate of development it takes about two to three years for a new Major Version of Drupal Core to be released. The development cycle is, at the time of writing, about halfway between the release of Drupal 7 and that of Drupal 8, which is scheduled to be finished in 2013. A new release starts with an initial phase of turmoil and fixing bugs, until the current version and its API stabilize. Meanwhile, new features are developed predominantly for the future release. To allow for progress between such versions, changes in the architecture and reworkings of existing functionality is possible. Therefore, although it is also part of Drupal's philosophy concerning backwards compatibility that the users' data is protected when updating to a new major version, the code of the website probably requires to be readjusted. To allow for longer phases of transition, the project also keeps providing maintenance for one previous Major Release, that way effectively granting site owners a time frame of about six years to move their code to a more recent version. Support is available in a variety of forms and from different sources, and ranges from peer support in online forums over books and documentation to paid customer service and training courses provided by affiliated companies. The large network of custom components surrounding Drupal Core is characteristic and a result of the distinctively modular architecture of the project. These modules build upon the core, which acts as a hub and provides common ground and scaffolding for the components. Also, published modules are stored in a central repository. They can be globally searched and can all be addressed with a shared issue tracker. They are, however, individually maintained and differ from each other in how they are developed, tested and released.

Contributions to either core or modules are in most cases developed locally and then proposed to be added as patches, which are then integrated into the project once they are sufficiently advanced. According to Klaus Purer, the modules are where most of the innovation happens in the development. If they are of high enough quality and serve a purpose that benefits most users, they may be considered for inclusion in the project's core. For many components there is ongoing discussion whether these should be included in Drupal Core or keep on existing as optional extensions. Lately there has been an increased promotion of using Distributions, special compilations of modules that provide the functionality for a variety of common use cases.

Contributions to Drupal undergo various stages of testing and quality control before they are accepted into the project. A rather recent addition to the testing process is the automatic test suite covering the functionality of Drupal Core, which includes both unit and integration tests. It is triggered whenever a patch is posted to the issue tracker, and if the contribution causes the tests to fail, it automatically sets the issue to an according status. Before being added to the core, every piece of code also has to be manually tested and reviewed. Only if the community approves of a patch it is considered to be ready for inclusion. In the modules, the rigidity of the testing process varies and depends on the individual maintainers' judgment, but tends to be more lenient to allow for a more rapid development of new functionality. The degree to which automatic testing is implemented also varies, but the larger modules are reported to be generally well-covered.

A major channel of communication and a source of cooperation in the community is the issue-queue in the bug tracker. It allows for posting ideas for new features as well as discussing existing ones. By setting the status of an issue and commenting on its progress, developers try to find an agreement over the progress of a component and its corresponding patches, as well as in which area there is need for improvement. Generally it can be said that a working community with closely connected developers is a highly held ideal in Drupal, and the project features an array of events and mechanisms for facilitating cooperation and integrating new people into the community. These include an IRC-channel with weekly Core Office Hours, participation in the Google Summer of Code, and several possibilities for real-world meet-ups between developers in the form of regular talks, workshops and the major convention DrupalCon, which is held twice a year. While these occasions strengthen the team spirit and help spreading knowledge, most of the everyday business happens on the Internet and is assisted by tools like the issue tracker, the chat system and the discussion capabilities provided by Drupal itself. Tool support is therefore especially important for communication purposes, in addition to test automation and version control of the code.

A second source of gaining information about the project internals other than direct contact within the community, is written documentation. According to the interviewee, Drupal Core and most of the modules are "excessively" covered with comments in the code, and although a collection of general and introductory documentation material can be found on the project portal, he asserted that the comments are in most cases a better reference than the documentation pages on drupal.org. He considered a lack of ability to visualize architectural processes a major weakness in Drupal's documentation system. Also, because there are less strict rules for documentation than for coding style, the quality of documentation varies in different parts of the system.

The governance structure of Drupal relies on a small group of people being responsible for developing the core and builds on a large and flat hierarchy of contributors who supply patches and develop and maintain modules. Dries Buytaert, as the founder of Drupal and its Benevolent Dictator, has the right to lead core development and to make final decisions concerning the future of the project. In supervising Drupal Core he is supported by a small team of co-maintainers, usually not more than three people at a time, who have committing rights throughout the core. For smaller tasks, and particularly for work in the modules, it applies that whatever someone wants to implement and finds a way to realize it usually finds its way into the project, a concept that is often described as *Do-ocracy*. Apart from a possible intervention of Dries Buytaert, Drupal does not have formal decision making rules or mechanisms to resolve conflict. Members of the community are expected to follow the social norms set out in the project's Code of Conduct, and the process relies on people discussing their issues and subsequently reaching consensus. Although this works in most situations, the interviewee reported cases in which two groups could not agree on a contested topic, and the Benevolent Dictator had to solve the dispute by making a final call. Klaus Purer attributed the difficulties of the consensus-based system to the growing size of the project and the diversity of its community. Apart from individual developers and the inner circle of core developers another factor contributing to the development of Drupal is the influence of companies whose developers are paid for doing work on Drupal. However, this influence is indirect in nature, as developers still speak and act as individuals. The Drupal Association, although it handles the project's finances and supports it legally, has little impact on its governance, and it does not pay for development efforts.

Long-term planning of the project, deciding which functionality should be added to the core and how to do this has long been a soft spot in the process; the interviewee described the situation in the first seven years of development as "very chaotic". However, recent development cycles have brought changes to improve the predictability of the core development, and for Drupal 8 so called *Core initiatives* have been installed. These are special task forces with designated leaders that are assigned to tackle challenging, risky or promising areas of development, like implementing support for mobile browsers of HTML5, in a preemptive and coordinated way. Concerning the perception of software management in Drupal, Klaus Purer described what he sees as "cultural clash", emphasizing that although sophisticated development processes were in place in many areas of the project, due to their pronounced *hacker ethos* many members of the community would be reluctant to openly acknowledge them.

4.5 XWiki

4.5.1 Project Overview

XWiki is an open source wiki software written in Java. It is described by its developers as a "second-generation wiki" [66], because it combines the functions of a standard wiki with enterprise features and allows users to extend the wiki with scripts and macros to create data-based applications. The project's development began in 2003, when its founder, the French developer Ludovic Dubost, created the first version and released it to the public, originally under a GPL-license. Shortly afterwards he founded a company to sustain the project and provide an organizational framework for its developers. Initially called XPertNet, it was later renamed and restructured into XWiki SAS and is at the time of writing fully owned by 12 of its employees. In 2006 the license of the open source project was changed to the less restrictive LGPL in order to facilitate embedding of code and make it easier to manage code contributions. The project showed steady growth, and in 2007 XWiki 1.0 was released.

The software targets a wide range of customers, from individuals and non-profit organizations to commercial companies. Besides serving as a basic wiki, it is used for collaborative authoring, as blog or simple CMS, to create mash-ups and feeds of external sources, and for similar tasks that are too small to justify stand-alone development. At the basis of the system is the XWiki Platform, which provides APIs and common UI elements. Those are necessary to power the main implementation, XWiki Enterprise, but can also be utilized in the extensions, the project's library of contributed modules, and in individual custom components. The project is governed by its committers, who each have equal voting rights in discussions. Although the majority of the core committers are employed at XWiki SAS, there is a strong separation of concerns between the company and the open source project.

Up to version 4.0, the latest Major Release, the project has accumulated a number of features for creating wikis and wiki-based applications. In addition to authentication and rights management, skinning and a WYSIWYG editor with inline editing, it also allows for the inclusion of user-created data structures, based on Hibernate. Furthermore, it has strong import and export capabilities and a form and scripting engine to extend its functionality and serves as collaborative development environment. It is used both by individual users and many customers in the enterprise sector, and has formed a special partnership with the e-learning platform *curriki.org*.

4.5.2 Case Study Approach

Following the suggestion of one of the core committers of XWiki, a form of communication different from the one used for the other projects was chosen to gain information about the project's development process. It was argued that by posting and discussing the interview questions openly on the project's mailing list, more developers would be able to take part in the conversation, and that the public could profit directly from the answers. The resulting exchange can be read in its entirety in the mailing list archives [65], and is also included in appendix , slightly edited for a more linear reading experience. Because of the chosen approach the interview took a significantly longer time to yield results. Yet, in return it led to valuable and diverse responses from several members of the development team. Contributions were made by Vincent Massol, Guillaume Lerouge, Ecaterina Moraru and Sorin Burjan. The interview was split into three sets of questions, leaving a adequate amount of time between posing each to allow for responses from the community. To account for the format of the interview, and because the project's documentation already laid out the basic structure of the process, the questions had to be both more specific and phrased in greater detail. The answers given differed in length, but were mostly to the point, included many additional links and granted insight into different viewpoints inside the core team.

4.5.3 Process Description

The release cycle for Major Versions of XWiki lasts approximately one year, with five or six Minor Releases scheduled in the course of this period. Before the development on each Minor Release starts, the community decides about its content and records the necessary tasks in the project's *roadmap*. This collaborative exploration and decision process happens on several levels. In the open source community, committers and contributors start by stating what they individually would like to work on for the release. Furthermore, there is a Roadmap Meeting at XWiki SAS, where stakeholders come together to discuss priorities and to find and assign developers as owners of issues and features. The proposed roadmap is further discussed and refined on the mailing lists, and committers have the chance to vote on changes concerning both content and schedule. The releases themselves are timeboxed, and structured further to contain several milestone releases and one or two release candidates, which are published before the final version comes out and serve as a means to collect early feedback from the community. As Guillaume Lerouge stated in the interview, about 40-50 percent of the core developers' time is preserved for maintenance and bugfixing tasks. XWiki developers pledge support for the latest stable version and the one under development, leading to two active branches being maintained at any given time. At a community level, user support happens in the mailing lists and the project's IRC channel, as well as through online documentation. The documentation is continually updated to provide information and instructions for the most common user requests, while developers and users try to answer specific questions individually but in public. In addition to said methods, professional support is given by affiliated companies, mainly XWiki SAS. Their paid services include hosting, user training, consulting and custom development.

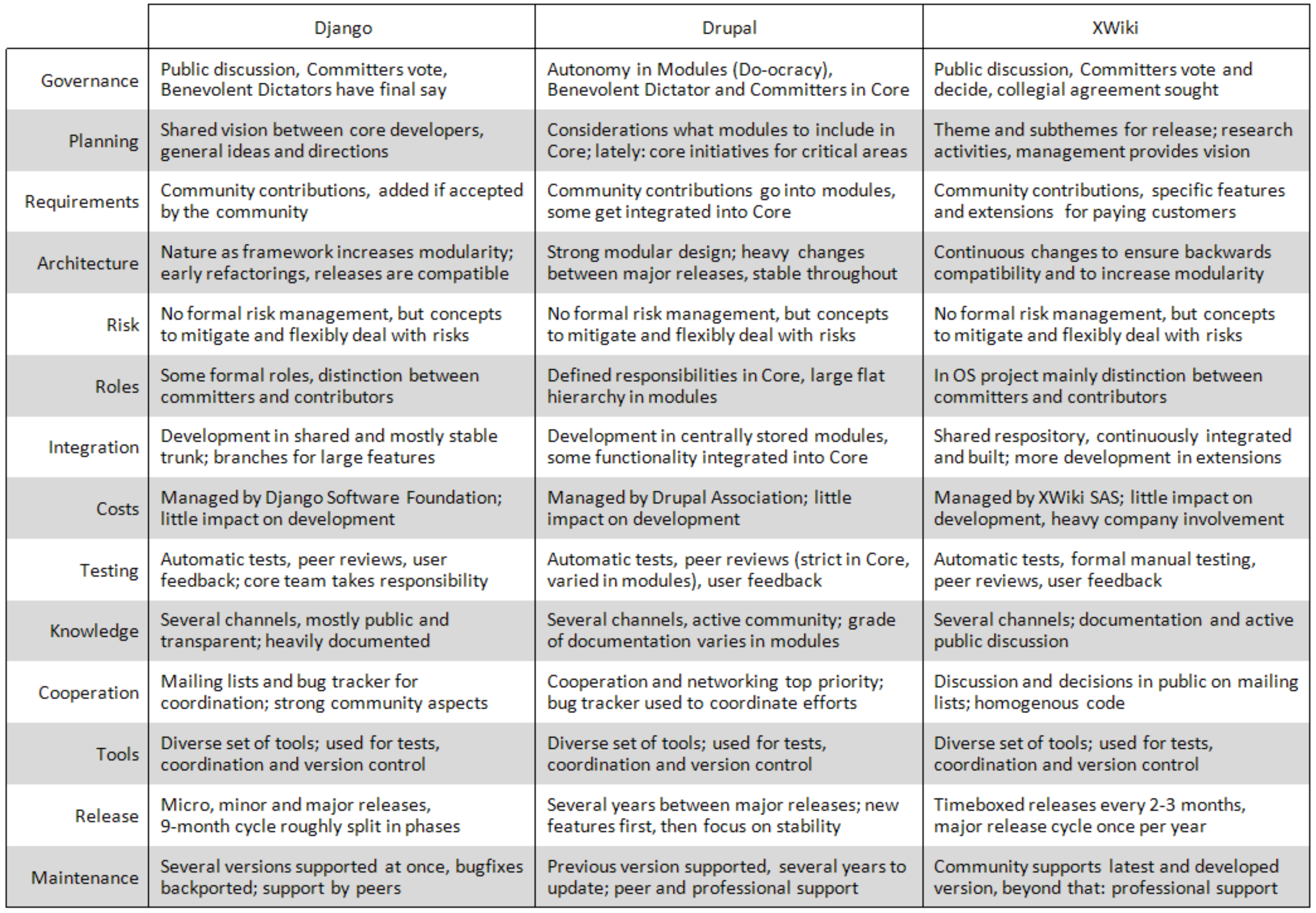
Keeping up backward compatibility has a high priority in the project to accommodate the needs of enterprise users. This has led to several design decisions, one of which being to embrace "evolution rather than revolution", as Vincent Massol put it in the interview, and to further the project through slow but constant changes. A major undertaking was the gradual move from a monolithic code to a module-based system in order to allow for easier customization and extendability. These modules can be added or removed from a runtime, and are managed with an integrated extension manager. Deprecated features are moved into legacy modules, but are kept in the repository in case they are needed anyway. From the perspective of the XWiki Platform user-contributed extensions are handled similarly to top level projects, accessing the same APIs and extending their functionality, and all code is collectively stored and versioned in a global GitHub repository. Whenever code gets checked in, a build is compiled by Jenkins, which is used as the project's continuous integration tool. Also triggered with every commit is the execution of the test-suite, which consists of unit tests carried out in isolation on mock objects, and functional tests, which in turn include UI tests and code validations. If the build fails, the responsibility to fix it or find someone who can lies with the Build Manager, a role for which the active committers take weekly turns. In the end, though, all developers are responsible for the quality of their own code, and should both manually test the patches they submit and cover them with automatic tests. For the manual testing process the project follows a combined approach. In part it relies on the community of users and developers to find bugs, especially those occurring in exotic browser or database environments or during special use cases. In addition there is a dedicated QA engineer employed at XWiki SAS, who follows a formal Manual Test Plan and specifically screens for certain problems. In both cases, issues are reported in the bug tracker JIRA, where they can be picked up and fixed by one of the developers.

As can be seen in the above description, many parts of the process, like test automation, code versioning and integration, issue tracking and decision making rely on a set of tools to coordinate the different groups and members of the community. Online communication platforms play an important part, and although the developers have meetings and discussions outside the observable channels, special emphasis is laid on making all relevant proposals and decisions in public on the project's mailing list. This is a mechanism partly in place to allow external observers to follow the development and to lower the entry barrier for joining the community. Also, a common coding style is encouraged to enhance the readability of the code, especially with multiple persons collaborating on the files, and the documentation specifically tries to address the problems encountered by many. The multiple active rules for development can, on the other hand, make it harder for new people to join the community. Still, everyone is encouraged to participate according to their abilities. In the open source project, the only official differentiation of roles is between contributors and committers. The former group is comprised of all people who participate in the project and add to its development by submitting bug reports, patches or translations, by writing extensions, or by just providing helpful feedback in discussions. Contributors who send in enough good patches and show a long-term dedication to the project can be voted in to join the development as committers. Through this meritocratic process developers gain full write-access to the source repository and the right to vote. They are also responsible for reviewing patches, and applying them when they meet the necessary requirements. Many of the currently active committers have also been recruited into XWiki SAS, which has a more hierarchical structure and a diverse role system, to work on the project from a company environment. However, the XWiki SAS title or affiliation with any other company plays no part when participating in the open source community, where all interactions are kept decidedly between individuals only.

The governance of XWiki lies in the hands of its committers. To make decisions, the project follows the voting rules laid out by Apache. These votes are held whenever important changes to the code, the process or the team have to be made. For an issue to be successful it requires three positive votes, and the fact that no-one makes use of their veto within 72 hours. Votes are accompanied by a discussion in the mailing lists, but although the expertise and the past actions of the developer influence the weight of their opinion, every committer has the same rights in the voting process. If no consensus can be found, the proposal is discussed and amended until everyone agrees on the course of action. While the current development team includes about 15 active committers, an interviewee expressed confidence that the system would be able to scale to 50 or more people. Although the long-term progress of the project is not planned ahead in detail, a shared vision is in place about the general direction of its development. Every year, before the work on a new major release starts, the team agrees on a theme and a sub-theme for the release. Also, the project maintains several research activities to stay up-to-date with modern technology. Ludovic Dubost stated in his blog series about the future development of XWiki [20] that the vision of the project is defined by the upper management, but that it is refined by discussions with its employees. Asked directly about the common ground of the process to other forms of software development, Vincent Massol expressed his personal preference for agile methods and listed several similarities between practices used in XWiki and Extreme Programming.

4.6 Summary of Project Findings

In order to provide a quick overview and a point of reference, the table in Figure 4.2 aggregates the case study results of the three examined projects according to the categories defined previously in the theoretical part of this thesis. It has to be noted, however, that the illustration shows only a shortened and simplified view of the data to fit the chosen format and should therefore be only considered in combination with the written descriptions made in this chapter, which provide a more thorough representation of the interview findings, and the detailed analysis following in Chapter 5.



	Django	Drupal	XWiki
Governance	Public discussion, Committers vote, Benevolent Dictators have final say	Autonomy in Modules (Do-ocracy), Benevolent Dictator and Committers in Core	Public discussion, Committers vote and decide, collegial agreement sought
Planning	Shared vision between core developers, general ideas and directions	Considerations what modules to include in Core; lately: core initiatives for critical areas	Theme and subthemes for release; research activities, management provides vision
Requirements	Community contributions, added if accepted by the community	Community contributions go into modules, some get integrated into Core	Community contributions, specific features and extensions for paying customers
Architecture	Nature as framework increases modularity; early refactorings, releases are compatible	Strong modular design; heavy changes between major releases, stable throughout	Continuous changes to ensure backwards compatibility and to increase modularity
Risk	No formal risk management, but concepts to mitigate and flexibly deal with risks	No formal risk management, but concepts to mitigate and flexibly deal with risks	No formal risk management, but concepts to mitigate and flexibly deal with risks
Roles	Some formal roles, distinction between committers and contributors	Defined responsibilities in Core, large flat hierarchy in modules	In OS project mainly distinction between committers and contributors
Integration	Development in shared and mostly stable trunk; branches for large features	Development in centrally stored modules, some functionality integrated into Core	Shared respository, continuously integrated and built; more development in extensions
Costs	Managed by Django Software Foundation; little impact on development	Managed by Drupal Association; little impact on development	Managed by XWiki SAS; little impact on development, heavy company involvement
Testing	Automatic tests, peer reviews, user feedback; core team takes responsibility	Automatic tests, peer reviews (strict in Core, varied in modules), user feedback	Automatic tests, formal manual testing, peer reviews, user feedback
Knowledge	Several channels, mostly public and transparent; heavily documented	Several channels, active community; grade of documentation varies in modules	Several channels; documentation and active public discussion
Cooperation	Mailing lists and bug tracker for coordination; strong community aspects	Cooperation and networking top priority; bug tracker used to coordinate efforts	Discussion and decisions in public on mailing lists; homogenous code
Tools	Diverse set of tools; used for tests, coordination and version control	Diverse set of tools; used for tests, coordination and version control	Diverse set of tools; used for tests, coordination and version control
Release	Micro, minor and major releases, 9-month cycle roughly split in phases	Several years between major releases; new features first, then focus on stability	Timeboxed releases every 2-3 months, major release cycle once per year
Maintenance	Several versions supported at once, bugfixes backported; support by peers	Previous version supported, several years to update; peer and professional support	Community supports latest and developed version, beyond that: professional support

Figure 4.2: Summary of Case Study Findings by Project and Category

CHAPTER 5

Analysis of Case Study Findings

"How do you eat an elephant? One bite at a time!" - Philippe Kruchten [33]

The findings of the case study, in combination with information gained from previous chapters, form the basis for the following analysis. Its aim is to split the results according to several criteria, re-integrate them into more broadly applicable conclusions and finally to set them into a wider perspective.

In a first step, the process categories established in Chapter 3.5 are revisited and updated with the newly gained primary data. The second part of the analysis looks at the process models, which constituted the core of Chapter 2, and tries to find points of intersection between them and the elements of open source development. With the final section of the analysis, the research questions, which were set up in the introduction, are assessed and responded to, reflecting the knowledge gained throughout the thesis.

5.1 Concepts by Process Category

5.1.1 Governance

A direct control of open source projects is very difficult, as there is no reliable way of giving binding orders to volunteer contributors. In the social context of a project, however, developers earn influence within the community, and the core development team does have control over the main repository. It has the ability to decide which contributions are integrated into the commonly distributed version of the software. Similar to a democratic system, this procedure works as long as the community members put their trust into the decisions of the group or individual holding control over the code, because unhappy developers can fork the source code at any given time, or will simply stop contributing. Regardless of the form of governance, a project, therefore, always needs the acceptance of the community. Who makes decisions in the projects, and how these decisions are formally made, differs in the inspected cases. In XWiki the committers can cast votes on contested issues, and discuss their opinions until they find consensus. Django has a similar system, which is, like XWiki's, derived from the voting system introduced by the Apache community, but in the case of ongoing dissent the two founders of the project have the final say in their function as Benevolent Dictators. Drupal does not have special voting rights for committers. If there is a conflict that blocks development, it lies in the hands of the project's Benevolent Dictator to settle it. In the external modules developers have the liberty to implement what they want, if they are able to realize it.

All examined projects also have clearly defined rules of cooperation and codes of conduct in order to establish a productive atmosphere of communication. Although in other projects different forms of governance are possible, they are necessarily limited to a certain range of parameters due to the nature of open source licenses and the restrictions of power those entail. Management tasks and responsibilities do resemble those found in agile development, but the open source environment presents unique challenges and opportunities which ask for distinct forms of governance.

5.1.2 Planning

In each of the projects of the case study, development follows a general vision shared by the community. Making concrete and reliable plans for the future is not possible, because a project's progress ultimately depends on the interests and actual contributions of its developers. There are, however, indirect means of control through which the course of development can be influenced. In order to achieve this, the examined projects pursue different approaches. While in Drupal everyone is free to add their own external modules, the Benevolent Dictator decides which contributions are added into the project's core, and thus retains control over its progress. Recent past also saw the introduction of Core Initiatives, in which the development of certain key features is actively pursued with a long-term plan. Django's progression depends on the views its core committers have of the direction of the project's growth. New committers are carefully recruited to reflect the general ideas of the existing community. XWiki starts every new release with a meeting and subsequent online discussions to agree on a roadmap and a prioritization of tasks. On a larger timescale the team agrees on a theme and sub-themes of development.

The nature of open source inherently makes detailed long-term planning hard to realize. Influencing the course of development requires trust, gradual changes, and the establishment of a shared vision within the community. Due to its constraints, planning plays a distinct role in open source development. In order to steer the project at all, agile ideas of management promise to be more successful than those of traditional software engineering.

5.1.3 Requirements

The examined projects employ a very similar approach in respect to their requirements. In all cases there is no formal requirement analysis, and there are barely any restrictions on the projects' scope. What gets added to the code depends primarily on what people are willing and able to implement. Developers with committing-rights can however influence which contributions are accepted into the main project, and what will be available in external modules or in private repositories only. The communities regularly assess the priorities of tasks, and discuss which features should be included in upcoming releases. Stakeholders outside a project have less influence on its requirements. Users can ask for a feature to be developed and try to convince the developers of its necessity, but unless they can implement it themselves there is no guarantee that it will be done. Even if everyone can pay developers to implement needed functionality, in order for the changes to be included into the main project they still have to make a case for the usefulness of the changes to the community as a whole.

Because of the nature of open source licenses nobody can be prevented from modifying the code, as long as the rules of the specific license are being upheld. But what is developed depends on the programmers doing the work, and what gets added to the distributed main version depends on the committers of the community maintaining it. While the agile approach has an ongoing discussion about scope between the project owners and the customers, in open source projects this discussion takes place inside the community.

5.1.4 Architecture

What the projects of the case study primarily have in common in terms of architecture is their strong modular character. This is most obvious in Drupal, which is strictly separated into a comparatively small core, which provides basic functionality, and a large number of optional modules. XWiki is made of several interacting parts, including a platform for common API, XWiki Enterprise as a working implementation, and a library of extensions which can be added and combined at will. Django is inherently modular because of its function as a framework. In all three project the code and its architecture have changed considerably over time. Django has been through major refactorings particularly during its early years, and gradually changed since then. In XWiki changes in the code have been continuous and cautious, but have slowly transformed the system into its aforementioned modular structure. Although Drupal keeps its code stable during the time frame of a major release, it encourages radical changes in its structure with each transition to a following version. Each project has deprecation routines and facilitates upgrading to new versions through different methods.

Modular architecture allows simultaneous work on different areas of the software by many people at once and appears to be a major factor for scalability. It also makes it easier to replace parts without compromising the integrity of the whole system. Open source projects use various approaches to change their architecture over time while making sure that users can reliably keep on using the software. By starting the architecture simple, but moving it gradually toward a modular structure, different approaches are combined.

5.1.5 Risk

The reviewed projects do not employ formal risk management as a separate process discipline. One explanation for this is that the possible risks of open source software are inherently different from those threatening the development of proprietary projects. Because their code is distributed and changes are revertible, and because their development does not directly rely on money or schedules being kept, open source projects are not easily stopped completely as long as enough people are interested in continuing to develop them. Nevertheless, certain events can hinder progress or have negative effects on the community. These are discussed as soon as they can be fully anticipated, and countermeasures are planned when necessary. Also, Drupal introduced Core Initiatives lately to preemptively deal with important technical challenges, and XWiki keeps up research activities to stay up to date with recent technology.

The influence of risks is mitigated by several aspects of the development process, like rapid feedback cycles, openly available code and knowledge, and the modular architecture. Although the specific measures and practices may vary between different projects, the overall observations concerning risk hold for a majority of open source projects. Open source projects face distinct risks with distinct dangers, but deal with them in a reactive and agile way.

5.1.6 Roles

In all participating projects, special rights and responsibilities are closely tied to the commit-access to the main repository. The role of committer has therefore an important function in each of the processes. Both XWiki and Django have an elected team of committers who, among other things, have the right to vote and the final say in accepting patches into the project. The committing rights for Drupal Core lie in the hands of its founder, who acts as the project's Benevolent Dictator, and a handful of different aides for each release. Django has two Benevolent Dictators complementing the core commit team. Additionally, the projects all define some specific roles and titles for individuals with specific representative or administrative responsibilities, like the release or build manager, or maintainers of project parts. Although people contribute differently according to their abilities, the projects do not distinguish formally between developers based on their field of activity. Each community has a basis of contributors, developers who add their contributions to the project but do not fulfill specific administrative functions in its organization.

This partition into a dedicated core team and a base of casual contributors, with specific additional roles for administrative tasks, is commonly found in open source projects. Although role attribution is at least as flexible as it is in agile methods, many of the roles and their distribution and tasks represent concepts distinct in open source development.

5.1.7 Integration

All three examined projects have a central repository, where different versions of the source code are kept. New additions to the code are applied to a dedicated development branch, or *trunk*, which should be in a working condition at all times. New builds are created whenever the code is changed. Major new features and bugfixes are developed locally and applied to the code base as patches, if they pass the respective project's review process. If a patch breaks the system anyway, it can be rolled back easily. Besides the development branch, the projects maintain stable versions of the software, which are created and released in regular intervals, and can later receive important bugfixes, but no new features. Major features in Drupal are often implemented as a separate module. In some cases they are integrated into the core at a later time, given that they are stable and considered to be useful for a majority of users. A similar strategy has been pursued lately with the extension-system of XWiki. Django has separate branches for the development of some large features, but most patches are developed in user-owned repositories before they are applied to the trunk.

In general the integration procedure appears to be facilitated by a modular architecture and by encapsulating changes in the form of patches. Development branches are likely to be updated and integrated continuously, in a way that is similar to the various other forms of modern incremental development.

5.1.8 Costs

Although the reviewed projects differ in their organizational background, money plays a similar role in each of their processes. This role is fundamentally different from its significance in the creation of proprietary software. The main discerning factor is that the actual task of developing the software does not cause costs for the open source projects as entities. This does not mean, however, that all contributions have to come from developers working for free. Django is indeed mainly a volunteer effort, but many of Drupal's contributors are paid by affiliated companies, and most of the core developers of XWiki are employed to work on the project. Because people still interact with the projects as individuals, the source of contributions is largely irrelevant for management purposes. Money is still needed for other issues, like legal expenses, promotion and server costs, but these factors are not directly tied to the development of the software.

Because the budgets of the projects do not include the development efforts, they are to a large degree independent of the development processes. Additional cost management can become necessary whenever projects are more directly involved in the payment for development tasks. All in all, however, the role of money in open source development is profoundly different from the one it plays in other environments.

5.1.9 Testing

The three projects of the case study each focus on the quality of the software in multiple ways and in various parts of the process. An important step of testing is to prevent that bugs get into the code base in the first place, by using peer review and executing automatic tests before or while patches are applied. All projects have an automatic test suite covering at least the most critical parts of the code. These tests are either written by the person contributing the code, or added at a later time. Which type of tests the test suites consist of depends on the nature of the code, but commonly they include at least unit and integration tests. Before code is added to the main repository, there is always at least one review by another developer. Django and Drupal require multiple reviewers for complex or critical patches. In any case, the final review is made by the member of the core team who applies the changes. Besides these measures, the projects rely mostly on the participation of the community in finding and reporting bugs that occur. The bug tracking systems in use provide a platform for organizing reported bugs and the information known about them, and they document the status of efforts made to correct them. In addition to relying on the community to find bugs, XWiki employs a dedicated tester, sponsored by the project's supporting company, who tests the code manually following a formal testing plan.

It is difficult to say which parts of the quality assurance and testing process have greater influence on the resulting quality of the code, but projects tend to pursue a broad mixture of approaches. Apart from the heavy involvement of the users in finding and fixing bugs, which is characteristic for this type of development, the elements of testing used in open source projects are applied similarly both in agile and structured approaches.

5.1.10 Knowledge

Common to the three projects is that they feature extensive documentation of the code and its development. General instructive knowledge about the projects can be gained from their online documentation pages, and specific information about the code is available in the form of code comments. There is also a considerable amount of interpersonal communication, but as these conversations are mostly held in a persistently written and publicly observable form, they are later available as archive and become part of the documentation and the common knowledge. In Django it is a necessary criterion for patches to be sufficiently documented. Drupal also emphasizes the importance of documentation directly at the code level, especially in critical code areas such as the project core. Although many of its core developers share an office, XWiki makes it a point to hold all important discussions and decisions publicly in archivable mailing lists.

Knowledge in open source projects seems to be predominantly distributed in written form, via persistent channels that are open to the public. Although it has to be noted that the projects of the case study were partly chosen for the quality of their documentation, a sighting of the market and considerations from the environment infer that these assumptions generally hold. The management of knowledge is less formal than it is in structured processes, and its exchange happens primarily between peers.

5.1.11 Cooperation

The projects of the case study are being developed in distributed communities, which influences the way how developers communicate. It also means that certain methods for coordinating people's work are necessary. One of these methods, which is common to the examined projects, is the extended use of a bug-tracker. Not only does it store bug reports and feature requests, it also documents the status of patches and fixes, as well as corresponding discussion among the developers. Contributions to the code have to go through a review-process, in which at least one other developer checks the code's function and quality. For larger features, many people work together to write the code. Although individuals are sometimes responsible for maintaining certain parts of the software, there is no code ownership for specific files, and every developer can make changes throughout the code base. This enforces a common code-style and good documentation to enhance the readability of the source code. Public discussions in chats and mailing lists, together with comments in the bug-tracker and the version control system, are used to discuss best courses of action. Although these communication channels and coordination methods are used in different degrees in the specific projects, the principles of cooperation are very similar. To give an example, in XWiki, where many core team members develop from a common workplace, important decisions are still made online to include the whole community.

The principles of using public, tool-supported communication channels and collaborating closely both in large-scale planning and in specific feature-development hold for most open source projects. Even if development happens distributed over the Internet, cooperation resembles that of agile methods in style and underlying concepts.

5.1.12 Tools

Both the types of used tools and the functions they are employed for in the process are quite similar among the inspected projects. Communication tools serve as a way to enable distributed and open conversations, to build social communities, and to support knowledge exchange. Version control systems are used to help synchronize different versions, to integrate them into a shared repository and to allow rolling back code if necessary. Bug and issue tracking software helps coordinating the contributions people make to the project, and automatic test and build systems allow faster feedback cycles and more efficient work. The projects of the case study differ in the specific software tools used and in the degree those tools are employed to fulfill the described tasks.

A common characteristic of the technology found in open source development is that it is mainly employed by the community in a decentralized, bottom-up way, in contrast to, for example, the use of planning and scheduling software in the RUP. This means that how tools are used is in essence a mix between their employment in agile and in structured development.

5.1.13 Release

Although a working copy of the current code can be downloaded at any given time, the development of each of the reviewed projects is structured into nested release cycles of different length. Larger releases allow for significant changes over a long period of time, while smaller versions released in between might only bring fixes for known bugs or security issues. Each major release itself is more or less strictly separated into phases. Apart from these common grounds, the specific structure and content of the releases differ between the examined projects. Drupal Core goes through release cycles of several years of time. Each cycle begins with a period of large-scale refactorings and the introduction of new features and then gradually stabilizes. XWiki mainly utilizes a shorter rhythm of time-boxed releases, which last between two and three months. Each starts with the community agreeing on a roadmap, and results in a new stable release. In Django a new major version is released roughly every nine months, and the time is split into periods to cover feature proposal, development and fixing bugs.

Different ways of release management are conceivable, but many open source projects seem to have long-lasting release cycles in addition to them continuously improving the current development branch. This allows the implementation of large-scale changes without sacrificing the continuous availability of a running version of the code. The resulting approach combines elements from both agile and structured types of release management.

5.1.14 Maintenance

As the reviewed projects are continuously developed and improved, they are also in a constant state of maintenance. The input that tells the developers where improvement is necessary and which bugs need fixing comes from the user-community. Issue tracker software is used to store and organize this information. Bugs that pose a problem for many users have a higher priority of being fixed right away. In addition to maintaining the current development version, the projects provide support for one or more previous releases, and certain patches are *backported* to older versions of the software. User training happens in various ways. An important part in all observed cases is peer support by the community, usually in persistent form (like online documentation, books, FAQs and searchable mailing lists) to allow many people to profit from answers given once. Affiliated companies or individual developers provide additional professional support if needed. The projects differ in how long previous versions are supported. While Drupal supports several older releases to allow users an extended period of time to migrate, XWiki only guarantees community support for the latest stable version and encourages users to update frequently.

The fact of projects being in a constant state of development and maintenance is an important general aspect of open source software. The form and scope of support might vary in different projects, although peer support is a key factor in most cases. As a whole, maintenance and support in open source development builds on several concepts that are not equally significant in either of the other approaches.

5.2 Process Model Comparisons

5.2.1 PMBoK

The PMBoK covers all areas of traditional project management and describes them from a manager's perspective. It represents a well-proven but generic approach, which does not give special attention to the possibilities and constraints of developing software. For most processes to function successfully, a manager using PMBoK requires a high degree of control over the project. As far as it has been observed, however, open source projects and communities are not and cannot be controlled in such a way. When trying to apply the lifecycle of the process model and its phases on a larger scale, this becomes obvious. As work on the project starts, its scope and the means of development are not yet decided on. This takes away the foundation of both initiation and planning process groups. Because there is no planned duration and end of development, the flow of execution is fundamentally different, and the project is not formally closed. It is possible in theory to apply the phases of the lifecycle to individual releases, but patterns found in shorter iterations of observed projects differed from this structure.

Planning ahead on different levels and in detail is an essential part of PMBoK, but was not, in this degree, seen in open source projects. Rough schedules are used to define releases, but which activities would constitute the respective period is not decided in advance. Incidentally, this also makes the concept of change management obsolete. Although shared visions of the projects typically exist, the specific scope depends on the changing interests of the community and on what developers actually choose to implement. Teams are not systematically assembled, but form naturally from the people who are dedicated to the project. Risks are not formally identified up front, but are addressed as soon they become apparent. Ideas from the monitoring and controlling process group are represented in open source projects, but the ways in which they are implemented differ. Controlling is decentralized, and inherently has to be more subtle, depending on interpersonal skills and persuasion. While monitoring is an important part of open source development, it also happens in an informal way. It is done publicly and utilizes widespread tool-support. Several areas in which the PMBoK specializes are hardly significant for the observed type of projects. This includes cost management, which is only marginally relevant for the development aspect of open source, and the procurement of proprietary components, which is problematic for both cost- and license-related reasons.

Overall, the many differences in form and content show that the core elements of generic project management, as they are proposed by the PMBoK, either do not work or are not needed in the specific environment of open source projects.

5.2.2 RUP

The Rational Unified Process provides a structured framework and adaptable guidelines to manage the development of software. It can be tailored to match specific problem fields and it allows different styles of development. This inherent flexibility complicates attempts to categorize the RUP. Depending on its setup and usage, it can have elements both of agile and of well-defined approaches. Taking this characteristic of the process model into consideration is necessary when comparing it to the ways open source software is developed. The phases of RUP's lifecycle differ in a subtle, but important way from traditional models like the PMBoK. Following an incremental project flow, the process aims to build a working prototype early on and develops it further through several iterations of the various phases until all requirements are met. The iterations used in the observed open source projects cannot be easily mapped to similar phases. Lacking predefined requirements, the projects as a whole did not progress through formal stages of development in order to reach certain levels of acceptance. A similar concept on a smaller scale could, however, be deduced from applying the lifecycle to shorter releases, which are often developed through various stages of growing function and stability, with milestones in between signaling progress.

Formal structures, static workflows and a fixed role-system, on which the process is partially based, take no significant part in the examined projects. The dynamic process components however, which are used to control development in an ongoing way, show a number of similarities. One common element is the flow of iterative, timeboxed development. Also, continuous verification and improvement of quality, which is supported by automatic testing, goes along the same line in both approaches. Building a component-based architecture is a major concept of the RUP. Although they arrived there differently, in a more gradual way, all examined open source projects have a similar structure in place. Even though both ways of development are heavily supported by tools, it differs how and in which areas they are applied. Features of open source projects are presented in a way that is similar to the use-cases endorsed by the RUP, but are less formally defined. All of this shows that the approaches often have similar ideas, but differ in the way they are carried out. Other areas, however, show differences in their core foundations. In contrast to the RUP, open source projects do not follow a predefined specification. In the bottom line, this removes the need for change management and analysis of requirements. Also, because those projects have different conceptions of risk and value, these have less significance as guiding principles.

All in all, the RUP is meant to be used in another environment, and is organized quite differently. Nevertheless, interesting parallels can be seen, especially when examining the dynamic methods used to tame the complexities of the project flow.

5.2.3 XP

Extreme Programming gives detailed rules about how to organize work in agile software development teams. It manages many aspects of collaboration by describing both a general set of principles and particular routines to follow in a project. Many of its practices are specifically designed to function in shared working spaces, which makes a direct comparison of some of its practices difficult. The lifecycle, however, is mostly free of such constraints. With its iterations lasting from one to several weeks, Extreme Programming creates new releases faster than the observed open source projects are able to. It can be argued though, that in addition to their official release cycles, all of those projects make new and often stable versions available to their users after every committed change. When considering individual development activities, a simple common workflow is found in both approaches. A developer chooses a task to work on, implements it, and after it is peer reviewed, the code is integrated into the main repository. In XP, ongoing close cooperation with the customer ensures that the tasks fulfill their requirements. In open source projects, developers are often users themselves, and implement whatever functionality they require of the product.

Some of the practices included in the description of Extreme Programming are related to the general flow of the project and provide guidelines for how to build the code. Many of the ideas entailed by these practices, whether they concern continuous integration, incremental design, quick and frequent builds, or fast release cycles, are reflected in a similar way in the examined open source projects. By encouraging testing before writing code, XP goes a step further in the aim of automating the testing process. Programming in pairs, sitting together and using the workspace as a means to convey information about progress cannot be transferred literally to the distributed way of creating open source projects. The underlying ideas, however, are present in the way their developers use the Internet to cooperate, share information and review each other's work. Those practices that give advice on how to structure working time of employees are not significant in open source environments, where work is either contributed by volunteers or managed from outside the scope of the project.

XP is a collection of guidelines which cover only a subset of project management areas. Also it is meant to be used in project surroundings which are quite different from those of typical open source communities. Still, the method's workflow and many of its practices resemble at their core the rules which guide the examined projects, and can be transferred to their environment with only minor adjustments.

5.2.4 Scrum

Scrum provides a particular lifecycle that can be employed to manage agile projects, and it explains the concepts, roles and meetings that are necessary to make it work. The method is specially tailored to fit its environment, and all of its practices are tightly interwoven with the project flow it describes. A crucial difference between Scrum Sprints and a typical lifecycle of an open source project is that the former starts from existing specifications and is goal-oriented towards fulfilling its objectives efficiently, while the latter is used to generally keep on improving the product. Many activities in Scrum focus around choosing, prioritizing and distributing tasks. Although feature logs exist in a similar form in the observed open source projects, they are less binding and defining for the overall flow than the Product and Sprint Backlogs are for a Scrum project. While knowledge exchange and feedback in Scrum happen primarily in personal group meetings, in open source communities this purpose is fulfilled by Internet discussions, tools and public documents. This form of asymmetric communication allows a larger and distributed group of people to participate in the project, but otherwise follows similar principles. Instead of Scrum's preference for low-tech and high-touch methods, these communities employ technology in a decentralized and informal way. The general flow of development shows similarities between the two approaches, like timeboxed releases, incremental development and constant measurement of progress. However, the lifecycle of Scrum is strictly controlled in several aspects, and requires developers to focus more clearly on the activities they agreed to take on at the beginning of a Sprint.

The method definition of Scrum explicitly describes two distinct management roles together with their common tasks and responsibilities. Although in open source projects those functions are usually not assigned formally to a single person, dedicated community members carry out tasks which are similar to those of the Scrum Master in order to keep the project running. These include initiation, moderation and conflict management, and enabling people to work unhindered on the project. It is possible that leaders of open source communities create and maintain a collection of desired features and prospective research areas, but in contrast to the backlog created by the Product Owner these have only recommendatory character. While Scrum works best in small teams of seven to ten people, many more people are usually involved in the creation of open source projects. Also, larger communities often distinguish developers depending on their commitment and involvement in the project. Apart from these obvious differences, teams in the two approaches share many characteristics, like being cross-functional, self-organized and being able to choose the tasks they would like to work on for themselves.

To sum up, the lifecycle of Scrum is structured to help small and agile teams efficiently reach the goals set for each iteration. While this differs from open source development as observed in the case study, several general characteristics are shared. Moreover, analogies can be drawn when comparing how teams are organized and how they collaborate.

5.3 Assessment of Research Questions

5.3.1 Shared Aspects of Open Source Development

From both theoretical research and analysis of the market - the latter including a close examination of the projects of the case study - it became apparent that projects in an open source environment, even if they differ in other aspects, share a considerable part of their ideas and methods of development. These similarities justify talking about an *open source development process*. Furthermore, projects show signs of a shared culture, commonly held values and a global identity. This is noteworthy as this process (and in a broader sense the culture) is not based on a single standard or one definite description, but instead grew out of several different communities, ideologies and pioneering projects. In part these common grounds stem from sociological reasons, like the constant networking between major project groups and the writing of various important texts which helped building this identity. But it also seems that developing in an open source environment alone prescribes certain characteristics to its process because of underlying shared aspects in the nature of open source projects.

The characteristics of open source development have been discussed on several occasions throughout this thesis. By extracting information from the available literature, a rough sketch of the process was built up in Chapter 3.4, which was then split into separate project categories in 3.5 to be compared to other forms of development. These categories were then revisited in further detail in Chapter 5.1, where the results of the case study were used as an additional data point to provide a more thorough assessment of the individual elements. But in addition, some general insights about the process can be gained or reinforced from these observations:

- License schemes and voluntary contributions restrict direct control over the process and make subtler, collaborative forms of governance necessary
- Development itself does not cause costs to the project, a fact that moves budget management outside of the project's central scope
- Because binding specifications cannot be defined in advance, the process is less goal-oriented and focuses on maintaining steady progress

- Using the Internet as a shared workspace and employing decentralized, asymmetric communication allows large, globally distributed teams to cooperate

It is not an easy task to measure the success of open source projects, as many different metrics exist, none of which is definitive on its own. The continuing existence of a community, which keeps up its interest in maintaining and improving the software, is the single most important factor for the survival of an open source project.

5.3.2 Agile Methods and Structured Processes in Open Source Projects

The core of this research question, examining the elements of agile methods and structured processes in open source development, has been a common theme throughout this thesis, and it has been a starting point for several detailed comparisons in this analysis and in the previous chapters. Besides providing a short summary of the findings so far, the primary focus of this section is to extract a number of general conclusions about the relationship between open source development and each of the other approaches, and to state the most important reasons for major incompatibilities.

Agile

One of the main conflicting issues when comparing open source development to agile methods is that the two approaches are optimized to work in specific environments, which differ from each other in a profound way. This allows various agile practices only to be applied indirectly to open source projects, if at all. On the other hand, many similarities can be seen in the approaches' general culture, problem solving strategies, and value systems. Together they share a common style of governance and collaboration, and prefer gradual adaptation over planning ahead. Communicating over the Internet, open source communities cannot act quite as flexible as agile teams, but are able to scale more easily.

Structured

Several main elements of structured approaches, like detailed plans at an early stage of the project, and separating the project flow into formal phases and processes, cannot normally be applied to an open source project due to inherent characteristics of its environment. Nevertheless, process models like the RUP, which is iterative and specifically adapted to the field of software engineering, clearly show similarities, especially in mechanisms which are employed to keep the project flow running over time and to tame rising complexity. Examples of common ground are found in architectural design, release and knowledge management, and partly in the way tools are employed in the process.

Open source development handles no major part of the process in the same way as either of the other approaches does. In many cases, though, it applies either hybrid solutions with elements from both, or an independent solution, which might have tendencies toward one or the other. A significant distinction, which sets open source projects apart from both agile and structured approaches, is that they usually do not start from specifications that should be eventually met, but instead evolve gradually, while always depending on the interests and capabilities of the community.

5.3.3 Variable Factors and Practical Constraints

Because open source development is not based on a single authoritative theory, it is not possible to assess projects with regards to the extent they differ from such a standard. Instead, it always has to be observed how individual projects relate to each other. Although single projects in an open source environment tend to be similar if compared to other forms of development, differences can still be found in project execution throughout all relevant areas of the process. On the basis of variations found in the projects of the case study and other popular examples of open source development, variable factors and ideas of corresponding causes can be derived:

- A system's architecture has to reflect the size of the project, with larger projects typically possessing some form of modular structure

- Projects have developed different ways of upgrading their product without deterring its users; forms of backward compatibility and the support of deprecated versions differ depending on targeted user groups and product type

- Although organizational forms and decision making structures differ between projects, they usually do so within a certain set of parameters, which are rooted in a global culture

- Depending on the project's business orientation, there can be different involvement of affiliated companies, which can provide additional services and professional support

- In most cases different kinds of testing and a variety of communication channels exist in a single project; however, their specific composition differs and depends on mixed aspects of the community culture

- In order to keep up with current external events and developments, different types of research activities can be carried out in a project, if the community considers them to be worthwhile

- The level of collaboration between individuals also differs between project communities, and is influenced to a certain degree by the complexity of the code

While it is not exhaustive, the above collection of factors can at least be indicative of some general trends and can provide a starting point for further considerations. In spite of the diversity found in multiple aspects of the process, the parameters move within certain limits which are specific to the environment and set open source processes apart from other forms of software development. Also, the most differences can be seen in those parts of the process where a broad spectrum of approaches can be successfully employed at once. In centrally applied concepts, like the project's governance or architecture, a smaller range of variety is found.

5.3.4 Transferability of Results

Many aspects of open source development have origins in other forms of project development and have been adapted to fit its particular environment and needs. But regardless of whether they stem from consistent adaptation, or if they are based on genuinely new ideas, the development concepts emerging from open source projects have their own merit and are aptly suited to guide the creation of software in their specific field of application. However, whether any part of this process can be applied to other environmental backgrounds remains a question to be answered.

In order to find transferable strengths of open source development, a good starting point for a search are those areas of the process which have the potential to pose serious problems, and therefore have undergone certain steps of innovation. An example for these are the mechanisms of coordinating the work of large groups of independent-minded individuals without detailed plans or imposed hierarchies. Many structures in open source projects, if applied together, specifically function to facilitate this kind of collaboration, like the extended use of issue tracker systems and other tools, the widespread practice of peer reviews, and the corresponding architectural design. Deserving of special note is the way people communicate in the projects. Communication is mostly public and persistent, and yet at the same time decentralized and largely informal. This grants large communities a fast and comprehensive way of exchanging knowledge.

These are mainly concepts which could be used in a similar way in order to allow Agile Methods to scale more easily, or to grant Structured Processes added flexibility. It can be argued that the self-organized forms of close cooperation found in open source communities require a certain mindset and a dedication to the project which may not always be found in traditional company environments. But this leads directly to one of the core strengths of said communities. The strong sense of motivation and social cohesion is a major driving force of open source software, and is based on various community-building efforts, a common sense of accomplishment and the possibility of dedicated members to participate on all levels of the project in a meaningful way. Finding ways to transfer these concepts and values to other structures of software development exceeds the scope of this thesis, but if successfully integrated into an approach they could prove to be an effective way to increase the cohesion of a team, and thus strengthen the project as a whole.

5.4 Summary

Analyzing the case study findings and relating them to previously obtained information revealed some general trends and further insights into the development of open source projects. A first glance at the literature had already suggested that no part of the process is carried out exactly as it is in either agile or structured development. However, tendencies toward one of the approaches were frequently observed. The second and more thorough look at the process categories, which took the results of the case study into consideration, calls for a validation and an update of these tendencies:

First, it should be noted that the observations of the theoretical process assessment still hold true in principle. However, when examining the process categories in greater detail, it became clear that most elements of the process have to be revalidated. When observed in the projects of the case study, mechanisms that seemed closely related to those found in agile or structured approaches, revealed peculiar underlying characteristics unique to open source development. Also, several of its parts, which resembled well-defined processes in the first place, have been shown to be more decentralized, informal, or otherwise show a tendency toward hybrid approaches. The style of communication and management, although different in many ways, often embodies agile principles. Changes from the original model and an approximation of observed tendencies are illustrated in Figure 5.1.

Because their development has no roots in a definitive theoretical standard, it was expected that different projects would vary wildly in their development approach. However, although variation among the projects and a certain spectrum of approaches was observed, strong common tendencies could be noticed as well. The same concepts, denominations, and solutions were encountered repeatedly throughout each participant of the case study. Both the special, distributed environment of open source projects and the shared conventions are factors contributing to a distinct culture and to a fundamental distinction of the open source development process from other approaches.

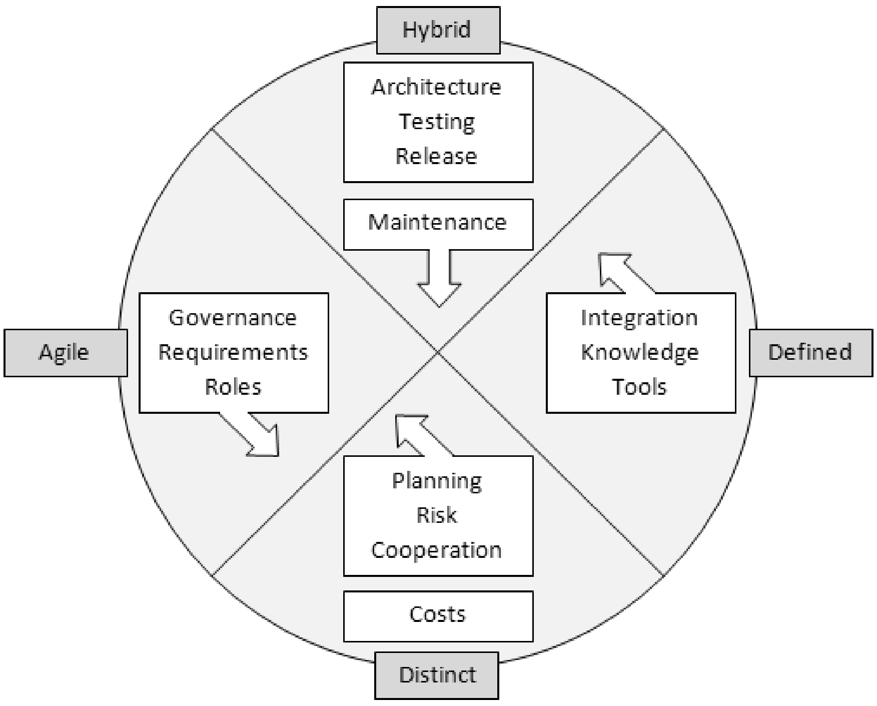


Figure 5.1: Tendency Shift Observed in Process Categories

Furthermore, a direct comparison to the four other examined process models showed different extents of similarity: With traditional, predictive forms of project management, as represented by the PMBoK, the least accordance could be found. In comparison with the RUP, some similarity with the process model's more iterative and dynamic features could be seen. Both of the agile methods showed resemblance of basic principles. However, most of their specific practices differ in the specific environment they are targeted at.

Some of the ideas found in open source projects can be transferred to proprietary software development. In particular, it is noteworthy how projects and teams are able to scale with little formal hierarchy, and even less signs of centralized control. This is made possible by a number of mechanisms, tools and design decisions. Successful open source communities give developers the chance to meaningfully participate in a common project and co-decide its course.

CHAPTER 6

Conclusion

"In software development, 'perfect' is a verb, not an adjective. There is no perfect process. There is no perfect design. There are no perfect stories. You can, however, perfect your process, your design, and your stories." - Kent Beck [6]

The final chapter looks back on the thesis and, after concisely revisiting the findings so far, goes on to explore the implications of this study and takes a glance into the future.

In a first step, the aims and the approach of the thesis are restated and the results of the case study are summarized and explained. The conclusion continues by placing this work in the context of related studies. Furthermore, suggestions for additional research topics are given, which could be pursued to expand the study beyond its inherent limitations, are given. Practical recommendations for different groups of developers and final thoughts complete the chapter.

6.1 Summary of Results

6.1.1 Thesis Aims and Approach

This thesis set out to investigate the development of open source projects and sought to demystify its profile, its characteristics, and how it relates to other forms of creating software. In a first step, concepts and practices prominently used in open source development were reconstructed from previous academic literature on the topic. From these, a process model was derived, which was then put in relation to a number of popular software development methods in order to find similarities and identify areas of distinction. In order to cross-check the results gained in this comparison with the reality of contemporary open source development, knowledgeable members of three active projects were consulted using in-depth qualitative interviews. This first-hand information, embedded in a background of theoretical research, served to deepen the understanding of how software is created in an open source environment. Further investigation showed how the approach differs from other types of development, and how it varies in different open source projects. Understanding how open source works leads to knowledge that can be used to improve software development processes used in open source projects, as well as those optimized for other environments.

6.1.2 Findings Explained

Open source development is a unique and self-contained approach for creating software. It is tightly interwoven with the open source and free software movements, and relies on the usage of an open licensing scheme and the participation of an active community. Although the process uses concepts shared with other forms of development, it stands on its own by employing distinct and specifically adapted solutions in most of its parts. It is iterative and open-ended, and heavily utilizes the Internet to coordinate distributed development efforts. Developers take over a significant role in defining the content and outline of the project, both by choosing how to contribute and by participating directly in the process of decision making. Projects are in a constant state of maintenance, and draw upon their user-base for suggestions and feedback. Knowledge is distributed on public and mostly persistent channels, through which the community discusses issues openly and in a decentralized way. These are some of the most common aspects shared by the majority of open source projects.

Although essential points of this information could already be derived from the theoretical analysis, the case study added further details, explained the rationale behind the data, and gave insight into the interrelation of practices, and what their successful use depends on: Necessarily, the specific environment of open source development influences the shape of its process. On one hand, it puts constraints on the possibilities of managing projects, because the degree of control that can possibly be exerted on the project and its developers is inherently limited. Put simply, people cannot directly be told what to do by anyone in the project, a fact that hinders attempts of detailed long-term planning. Scope depends on the interests and capabilities of the community; successfully steering the course of the development requires persuasion and skills in community-building. On the other hand, the open source environment changes the structure of the process by altering the importance of its elements. Several areas of traditional project management are not relevant in an open source project, and therefore play no role in the process. Others require special treatment and profoundly different practices because of their specific nature. In a similar way, the lifecycle of open source projects differs from that of conventionally managed projects, particularly noticeable in both initiating and closing phases. Whenever methods are compared directly, this shift in process coverage, balance and scope has to be taken into careful consideration.

Another noteworthy observation is that open source projects share many characteristics that go beyond what is necessarily defined by their environment. Those similarities in project structure, roles and tool sets, decision making procedures, and very specific terms and vocabulary cannot be explained by the necessity of project surroundings, but are instead signs of a shared development culture. This culture originated from common roots of the open source movement, the emulation of successful pioneering projects, and ideas absorbed from widely read vision papers. In an early criticism on the open source methodology, Steve McConnell once remarked that it was primarily based on “informal legend, myth and lore” [38]. Although the rationale behind the development process of open source software was already more complex back then, and has evolved significantly ever since, it should be noted that there is indeed not a definitive single theory, but tradition and loose conventions to be found at its ground. As a consequence, the process is naturally harder to understand, to define, and to expand upon.

However, this also means that many variations of the open source development process exist. In fact, there are in all likelihood no two projects that employ the exact same process. Still, the differences between projects are limited by the boundaries of the open source environment, the shared culture, and interdependencies between practices. Possible reasons for variety are diverse and include the project's business model, team composition, size and product type. Projects examined throughout this thesis showcased independent solutions in several process areas and have adjusted their approach as they grew and matured.

All in all, the developer interviews gave insight into the development of open source projects and provided working examples of a multifaceted and elaborate development process. It is in its main parts independent of other approaches, although parallels do exist in a number of practices and shared values. Variations exist within constraints of environment and culture.

6.2 Comparison With Related Work

A large quantity of studies has already been carried out dealing with the different aspects of creating open source software. While a considerable part of them focuses on a specific aspecct of development, only few examine the process itself and the actual use of practices in contemporary projects. One of the most comprehensive overviews of the research on open source and its findings is provided by the meta-study of Crowston et al. in [18]. Another useful compilation of studies is [61] by von Krogh and von Hippel. These collected findings have been the main foundation for describing the open source process in Chapter 3. However, the case study showed a number of practices being employed by the observed projects that were not found as such in the examined research, and major shifts were noticed in the emphasis of the development process.

One possible reason for these differences is the previously noted variation among open source projects. As all examined projects differ from the body of research in a similar way, it is likely though that contemporary open source development has evolved from its early roots and picked up trends not yet visible in the majority of academic research. Indeed, many studies rely on secondary data, particularly on early texts that observed and defined the open source process at its beginnings. Those texts include works like Eric Raymond's [43], the comparison of Apache and Mozilla by Mockus et al. [39], early works of Walt Scacchi [48] [49], and the influential essay [60] by Paul Vixie. While those studies and essays all are important documents for understanding the beginnings of open source and contributed significantly to the building of an open source culture and its shared identity, in many cases their practical observations no longer reflect the reality of open source development. Because of the continually evolving nature of the process model it is critical for effective research to incorporate contemporary project data. For this reason, the thesis included primary data from an in-depth case study.

A wide range of approaches can be employed to study the development of open source projects. Two books demonstrating the bandwidth of in-depth research are "Producing Open Source Software" by Karl Fogel [21] and "The Success of Open Source" by Steven Weber [63], who represent widely differing vantage points on the same topic. Being an open source developer himself, Fogel concentrates on the practical aspects of building a project and provides guidelines for successfully managing its execution. On the other side, the political scientist Weber observes open source culture from the outside and provides an anthropological view on the workings and rationale of open source communities and the processes used by them. Both view the process as a whole and emphasize complex relationships between project development and its surrounding environment. The two contrary approaches highlight the importance of different perspectives in order to fully understand a complex topic such as open source development. By placing qualitative process data into a theoretical framework, this thesis adds another perspective to this picture.

Comparisons of open source practices with various other process models has a longstanding history in academical research on open source. On one hand, traditional software engineering was mostly shown to differ from open source development in various aspects, for instance by Massey in [37]. On the other hand, a number of studies, including [30] and [62], noted analogies between the values and practices of agile methods and those of open source development. On the basis of similar considerations, several attempts of hybridizing the two approaches have been made, an example of which being [3]. The impact of these theoretical studies on the actuality of open source developing is, however, unclear. Quite recently, researchers have started focusing on the complementary field of *Inner Source*, which signifies the usage of open source development practices in a corporate environment. Stol et al. discovered several interesting challenges and implications concerning the topic in their qualitative study published in [57]. By analyzing analogies and differences, comparative studies have contributed a lot to identifying the special characteristics of each development approach. By using both agile and structured process models as points of reference, this thesis helped defining the open source process more clearly.

Sound and in-depth qualitative research, which incorporates both the position of developers and theoretical background, offers great promises in extending the knowledge on the complex interactions found in open source projects. Besides [57], another good example for a study based on extensive developer interviews is Sonali Shah's [52], in which the researcher analyzed the manifold motivations of people contributing to open source projects. However, most studies rely on secondary data and quantitative measures, while only about 10% are based on interviews, as Crowston et al. observe in [18]. For building accurate and usable theory, Shah and Corley note in [53], qualitative and multi-method studies are necessary. By combining theoretical process information, project documentation and data from qualitative interviews, this thesis provided such an in-depth and varied view into open source development.

All in all, this comparison with related work showed that with its practical examination of contemporary project development - embedded in a theoretical background of process models - the thesis contributed in multiple ways to an improvement of modern theories for developing open source projects.

6.3 Open Issues and Future Research

The thesis as it stands produced useful results, gave insight into the development of open source projects, and fulfilled its initial goals as defined by the research questions. However, the case study was limited to a relatively small scope, examining the general development processes of a carefully selected sample of projects. The most promising ways of adding to the research are therefore to modify sample size and composition, to give a special focus or greater depth to each of the interviews, or to ground the case study on a different theoretical framework:

Sample Set

- An obvious choice of action is to expand the sample set to include a significantly larger number of similar projects. This is laborious, but improves the variance and reliability of results. It helps identifying common tendencies between projects and has the chance of pointing out individual solutions.
- Another possibility is to specifically choose projects that differ in one of the relevant project properties like product type, size of community, age, or business model, but otherwise show similar characteristics, in order to isolate a respective factor and investigate the influence it has on the development process.

Observation Focus

- The interviews of the case study provided an overview of the whole cycle of development. A different, but equally viable strategy is to concentrate on a specific part of the process and examine it in detail throughout various projects. Each of the observed process categories can reasonably be treated as the main focus of a conversation.
- A further interesting prospect is to observe a chosen project over time, carrying out interviews at various stages of its growth. Although this requires time and dedication to carry out, the study setup would in turn be able to show and explain the changes a project goes through in its development from an immediate and uniquely traceable perspective.
- An approach that was partly realized in this study is the consultation of multiple developers per project, either in an open conversation or individually. This helps illuminating the process from different points of view, and improves results by

triangulation. In large projects, this setup may become necessary if no single interview partner has sufficient insight into all examined parts of the process.

Process Comparison

- The comparison with four different process models was used to define the structure of the interviews and helped putting the findings into context. By comparing open source projects with other development methods than the ones used in this study, and thus applying different measurement, interesting new results could be gained.
- In the case study, only stand-alone projects and processes were examined. Although most open source projects are developed independently, there are several organizations like Apache, Eclipse or Mozilla, which develop groups of projects in sequence or concurrently. In these cases it is worthy of research how process improvements compare to those measured by organizational frameworks like the CMMI.
- The current research compared theoretical standards and methods on one hand with actual processes found in open source projects on the other. However, an interesting and perhaps more balanced approach might be to compare realistic project data on both sides, by carrying out interviews with developers from each of the compared process models.

The above list of suggestions covered some of the areas in which future research could reach additional conclusions using a study setup similar to the one employed in this thesis. Along the same lines, further related topics could easily be thought of, and researched using analog methods.

6.4 Implications and Recommendations for Practice

It has been argued before, and demonstrated by many successful projects, that open source development does work. Research like this one shows why, how, and in what circumstances it works. To know these factors is important for another question with significant implications: How can software development be improved to be more successful, and to be successful more often? The introduction already emphasized the frequent failure of projects, and this problem is not limited to commercial and proprietary environments alone. Karl Fogel began his book on producing open source software [21] by decidedly saying: “Most free software projects fail.” He argued that failure of open source projects was merely less visible than that of their commercial counterparts, because they do not fail in a single dramatic event, but rather through decline of interest and lacking contributions. Yet, the main motivation for studying and comparing processes does not even have to be a threat of failure. It is enough to know that most projects could do better in what they are doing to justify research on development and searching for chances of improvement.

Recommendations for Open Source Developers

The fact that open source projects are not based on a common theory has contributed to the movement’s diversity and led to many creative solutions. On the downside, this means that the same mistakes are found to be repeated throughout every consecutive generations of projects. A systematic investigation of contemporary open source processes and modern research is crucial for anyone starting a new project, but seems to be often neglected. By considering state-of-the-art instead of re-evolving from the roots of the movement, reliance on trial and error can be reduced significantly. New governance and business models, useful and highly-modular design ideas, integrated tool-support and modern means of communication have been introduced in many communities individually, without being adopted by a majority of projects. A lot of innovation is therefore wasted or considerably delayed.

Projects are well-advised not to stop innovating. Even if a part of the process works as intended, possible improvements should always be considered. For example, email-based communication used in many projects has changed little over the last decade, while modern technology allows faster, more convenient and better integrated means of communication and connectivity. Social and technological structures are repeatedly inherited from projects like Linux or Apache without any major innovation or adaptation.

In place of the radical (and often crazy) ideas that fueled the movement's initial growth, in many areas today's projects exhibit conformance to cultural standards and settle for what *works*.

Also, parts of the movement still consider proprietary development as "the enemy" [56]. While the ideological implications of free software are a source of motivation for many and a unique part of the movement's culture, this should not lead to an isolation of the method. A familiarity with approaches used in proprietary development is necessary for understanding and, in consequence, effectively improving open source processes.

Recommendations for Other Developers

For other creators of software, open source processes can serve as a working example for a unique way of development which is in many ways different from commonly used process models. In its analysis, this thesis briefly discussed several areas in which it could be useful to transfer ideas found in open source projects to other forms of development. Noteworthy and innovative elements include distributed, tool-supported cooperation, decentralized management with active developer participation, the strong focus on user-feedback, and open, persistent and decentralized knowledge exchange.

How these concepts could be modified to be successfully applied to a different environment exceeds the scope of this research. Here they are mentioned to serve as a reminder of the potential open source development methods have for introducing new elements into the overall area of software development. For the last decade, project management was mostly stuck in drawn out debates between Structured Processes and Agile Methods. Experimenting with the inclusion of open source concepts could be a way to find novel solutions for old problems and to perhaps move away slightly from this dichotomy.

Final Thoughts

The elements of a process are interrelated and complement each other. Effective improvement often depends on making several careful steps and courageous jumps, all at once. In order to do this successfully it is necessary to understand the motivation behind practices and rules, to know the parts of the process and to see correlations and causations.

Two strategies can lead to better processes. The first one is to improve existing approaches gradually and carefully. Every method, even if employed correctly, can always be pushed a little bit further. The second strategy is to keep on searching for radically new ways of developing software. Now called *traditional* software engineering was once radically new; as was agile development, and open source.

In the end, it probably does not matter if legendary silver bullets are ever to be found - it has always been the *search* for better solutions that continues to drive constant improvement and occasionally inspires profound innovation.

Acknowledgements

A number of different people were involved in the making of this thesis, without whom this work would not have been possible.

My first and foremost thanks go to my parents, Alois and Ulrike, not only for their constant financial and personal support, but also for their ongoing care and the occasional gentle nudge in the right direction.

Furthermore, I wish to thank the people at the INSO institute and the AMMA group for their supervision and guidance, in particular my advisor Thomas Grechenig as well as Martin Pazderka and Brigitte Brem for their dedicated support.

The chosen approach would have failed without the active assistance given by the open source communities and individuals involved in this work's case study. My special thanks go to Russell Keith-Magee from Django, Klaus Purer from Drupal, and Vincent Massol, Guillaume Lerouge, Ecaterina Moraru and Sorin Burjan from XWiki.

And lastly, my dearest Evelyn, I would like to thank you for proof-reading and honest advice, for moral and practical support and, perhaps most importantly, for believing in me throughout this whole time.

Martin Schönberger

APPENDIX A

Interview: Django

Martin Schönberger: Thanks a lot for agreeing to talk to me. Can you just tell me for a start a bit about yourself and your role in the development of Django?

Russell Keith-Magee: Okay. My full name is Dr. Russell Keith-Magee, I am a core developer of Django and I joined the project in January of 2006, which was about six or seven months after the Django project went public. I don't know how much you know about the background, but Django originally started as an in-house project at the Lawrence Journal-World which is a local newspaper in Kansas. The two co-founders of the project, Jacob Kaplan-Moss and Adrian Holovaty, went to PyCON in 2005 and gave a little informal presentation about this thing they were working on and were sort of overwhelmed with interest from people who said: "Hey, can you open source that so we can help out?". They open-sourced in June-July 2005. I discovered the project in about November 2005, made a few contributions and was added as a core contributor in January 2006. Since then I've been fairly active contributor to the project, the early days was a rapid development time so there was lots of new features I could add in, so the fact that Django is a testing framework is mostly my doing. After that point Django sort of stabilized and became a more mature framework so a lot of it has become smaller, incremental changes and managing the introduction of larger features developed by other people. So for example I mentored for the Google Summer of Code, Alex Gaynor who is now a core contributor but at the time wasn't, adding multiple database-support to Django. So yeah, I am a core contributor, I have been a core contributor - over the last twelve months I've been a little less active than I have been in terms of actual code contributions, but I am still active in the mailing lists in terms of helping users with support and having discussions - where the project should go, what features we should add, what properties those features should have and so on. Not as strictly related to the software management sort of things, but I'm also the president of the Django Software Foundation, which is the not-for-profit fundraising of the Django project. I took that role over just short of two years ago. That's got very little to do with the development of Django itself, other than being the body that holds the RP, is the legal owner of the source code from a copyright-perspective, of a trademark-perspective and is the body that you can give money to if you want to give money to Django.

MS: So that money, how is it ... ?

RKM: Well, that's the interesting thing. Because it's an open source project, because it's a volunteer project, the requirement for money has very little to do with the development of the soft ware itself. It has a lot more to do with things like: We have a trademark dispute because someone has decided to start using the Django name in a way that the project is not comfortable with. When we need to hire a lawyer. That's expensive. To actually make the trademark application for Django.That's expensive, and involves a lawyer. It's more things like that. And then occasionally we have been able to give funds for people which gather at [] conferences or to give keynotes or things like that. So it's not strictly to do with development of Django or paying for the development of Django itself, but it is related activities to support the project as a whole. Did you want any more about me personally or my background or .. ?

MS: Well, you told me you were the release manager of 1.2 and 1.3?

RKM: Formally not the release manager. We have a release manager, that's the person who is responsible for actually cutting the release. Inside that - one of the things that I'm sure will become apparent over the course of this conversation is that a lot of the roles in Django are very ad-hoc. Anyone who steps up to do something is the person who ends up doing it. When we get close to the point where a release is necessary there is usually in the community who says: “Hi, it's about time we do the 1.2 release” and then that person steps up and really drives the process to make sure: “Okay here are all the bugs, I know where all the bugs are, I know what status they're in, I know who's working on them” and just mainstays on top of that process until that thing is out the door. I managed that process for 1.2 and 1.3. 1.4 was handled by Aymeric Augustin who is a relatively new contributor. But it's not a formal role in that sense. We have a formal release manager who has the keys and knows all the email addresses - of the Red Hat package manager who needs to be notified when there's a security update - and that sort of thing. That wasn't me. I was just the person who managed a lot of the work, managed the release process.

MS: I see. The release process seems kind of complex, with the branches and things - I think it might be a bit complex to get those differing versions together, the feature branches, and the bugfix releases. Is there development at the same time in different versions of the software? How does it usually work?

RKM: Yes there are branches, but generally speaking they don't actually play that big of a role in our development process. We generally work on a principle that the trunk should always be stable. So, although in practice you probably don't actually want to do this but you should, in theory, be able to take the most recent commit to the version-control system and deploy it into a live site. So the goal is, the trunk should always be stable. Nothing gets committed until it is stable enough that we have confidence that we could use it in []. Essentially there's only ever really three branches in effect. There's trunk, where all the active development is happening. There's our maintenance release, which is one version back effectively. So right now, we have just released Version 1.4, our maintenance branch is 1.3, where any significant bugfix gets back-ported. By significant bug-fix we basically say anything that would have held up the release of 1.4 if we had known about it before 1.4 came out. So that means: catastrophic data-loss. If a query can accidentally delete half your database. Or a significant error in a new feature. This is a new feature that has been added, and something is fundamentally wrong about the way that it operates. Things that are on that sort of level of significance get backported one release to 1.3 and then we also maintain one version back prior that for security problems. So if someone discovers that there is a hole in a CSRF framework, for example, which allows a malicious attack to pretend that you are a user you are not. Then that gets backported two releases. So it would be in trunk, it would be in 1.3, and it would be backported to 1.2. Now effectively that means that we've only ever got three versions of the code we are working on, and two of them are only ever receiving back-ports of things that go into trunk. We do have other branches, so if you look in it there are other branches that are in the tree. They are usually for large projects that have been officially mandated, or officially sort of blessed by the core team. That in practice usually means GSoC, mostly because GSoC requires us to have a branch in order to get paid. Alex, for example, did his multiple database support. We set up a branch, under the Summer of Code, and every now and then, Alex would push code into that branch which would then be the official GSoC multiple database branch, which eventually got merged into the main release tree. And essentially the day it was merged into the main trunk tree it was ready for production because it had been tested very heavily. When we're developing other major features we tend to just develop them as patches, there is a growing use of things like Github and Bitbucket in the community broadly, and quite often you discover that those features have been developed in a branch in someone's Github repository, but they're not officially branches in Django's core repository.

MS: I see. Can we talk a bit about the testing process, you said you were involved in it? How did it get set up?

RKM: Okay, so.. I can claim responsibility for part of the testing framework. When I joined the project, it had been in serious use, and actually had an off-shoot as a commercial product as well, the journal World. Django was very heavily tested to start with, there was very very good test coverage off its cart. And we essentially said if anything ever breaks in the test-suite the person who commits it has to fix it, and you don't get to commit anything unless it is fully tested or there is a very significant reason why you can't test it - there are some high-level meta processes that are very hard to test and so we accept [] testing of that. What I added back in 2006 was essentially that framework that lets you add easily tests to your Django project, not to Django itself but to the Django project. And all of those tools they get, they're essentially shared between what Django uses to test itself and what you as an end-user can use on your Django project. So we have a test-suite, last time I checked in ran something like 4,500 tests I think. So it has fairly heavy coverage, it covers anything from individual unit-tests to small features to view-level tests, integration-level tests, or just recently we started adding browser-tests, to make sure that Django's admin interface worked the way it should.

MS: How long does it take to run these tests?

RKM: Depends. The controlling factor is mostly the database that you are backing it on to. If you're running under SQLite on a moderately modern machine, you're looking at about 10-15 minutes to run the full test-suite. If you install it on a badly configured version of MySQL, it can take eight hours. Everything in between that is possible. One of the areas that Django doesn't do very well is we don't mock database interfaces for example, so when we run tests we're running them against an actual database, which means you got a lot of 'create-table', destroy-table', 'create-table', destroy-table', .. and on many databases, that's not an [] operation, because it's not something you should be doing very often. So that's the source of the slowdown when there is a slowdown when you test right.

MS: And then there's also then the testing on the user-level, like the bug-triaging system? And I read something about code reviews. Every code is reviewed when it is checked in?

RKM: The way that the commit-process essentially works is that we have a very small, small by some standards anyway, core commit team. So there's only about 28 people I think, who could commit anything to the Django source repository at all. Of those, at any given time there's probably only a dozen or less who are actually actively committing code. Many of them are people who had access and just haven't done anything for three years. So the core team ultimately takes responsibility for checking anything into the tree. And that means that if it breaks, it's their fault, it's their responsibility to fix it, so we are essentially the last line of review that happens before anything gets committed. If you got a core team member who's been developing something on their own, they will usually end up calling on another core committer to say: "Hey can you throw some eyes over that so it's not just my eyeballs that have been reviewing this." Looking at contributions from the broader community, we have a bug-tracker, we have trac. The way our documentation describes it as sort of a community garden. Anyone is welcome to do anything they want to the database, to update any ticket, to modify any ticket, to add any patch they want and we sort of rely upon the fact that aren't going to be actively malicious and that they'll be sort of cautious and not just go through and close everything because they can. So what we ask people to do is essentially: any piece of code that you add to the repository as a patch gets someone else to look at it. And if it's a really big patch, maybe get two or three people to look at it. The core team won't commit it until they're satisfied that is has been reviewed, and generally if it's a big patch, one of the precursors that we are looking for is that there is evidence that three or four people have looked at this and said that it's okay. Or even better, that three or four people that I know have looked and said it's okay. So it's not a formal "You must get this person to look at it, then it must be checked off by this person". It's a very soft process, that is ultimately held down by the small group of core committers to [] of what is acceptable.

MS: Okay, that makes sense. So, when someone proposes a feature .. I would like to know a bit more about the larger picture. When you plan the development of Django over a long period of time, how does the vision of Django change over time? Three years ago, what did you think the project would look now? And I've seen plans of Django 1.5, .6, .7, 2.0, that's about three years in the future. How can you steer the project to that with the contributions of random submitters?

RKM: Right. That is perhaps the biggest culture shift and the biggest fundamental difference between textbook software engineering and what happens in an open source project. Essentially we can't guide the project. I can control what I do, I know what I want in general, and I know, because I talked with other members of the core team, maybe what they're interested in as well. But I can't make anyone else be interested in something. and ultimately, because this is a volunteer project, something only gets done if someone actually volunteers to fix it, which usually means that the features that get added are whatever itch someone has right now. And sometimes someone will say something or do something in a particular forum and that will excite a lot of people at the same time to look at one particular problem. So what we so discovered over time is that it is almost pointless for us trying to plan what is going to be in two years time or even two releases time, it's almost pointless to work out what's going to be in the next release until we are almost ready to cut the []. So the features that get added are essentially whatever anyone in the community can write and essentially get the community and core team to buy in on the idea that their approach is correct. That's a little bit harder than it sounds, it's not completely ad-hoc because the core team is somewhat resistant to change. We have a very firm backwards compatibility policy, we have a very firm [principally led surprise] sort of design aspect. One of the reasons the core team has remained so small for so long is that we are very picky about who we add, and we need to make sure that the people we are adding to the project don't have fundamentally different ideas about where the project is going long term. And the best judge of whether someone has a similar design ethic and feeling about how the project runs is to see them contribute a bunch of code and see that the code they are contributing is something you would have written if only you'd had the spare time. So, essentially we don't have grand plans. We have ideas of things we might like to fix, we have directions we might be vaguely heading, someone to protect [] an idea we might sort of say. We've had some ideas about that, and that kind of goes in the opposite direction of where we thought we were going. But ultimately the features that get added is whatever someone actually volunteers and steps up to write the code for. And manages to get the two or three other people to review. And then manages to find a core team member who is actually willing to do the final review and commit. Now a good example of that in practice, we had just recently an active and sometimes heated discussion about the way the users manage things on Django. Now that is a feature that has been in Django since Version 1, so way back in the days of yore [..] So the user model has been in Django since way back, since before I joined essentially. But it had a couple of assumptions built into it. That a username must have this many characters and must be unique, and that an email-address must fit into this many characters, and there are a number of people who have complained over the last six years that the user model isn't flexible enough. Over those six years people have variously said: "Hey we should just do this." and the core team has said: "No we've got bigger plans we'd like to address here, this is sort of the direction we would like to go." And the person who said "Well I'm just going to do this little thing" they've just gone off and done the little thing and not contribute it back to Django. What has happened just recently is that enough people got together and sort of said: "Hey this sucks enough that we wanna see if we can fix it, how can we fix it? Let's go through a process of working out exactly what we need to do, what we can't do, where our constraints are, build up a full formal documentation of what we could do, and then get the core team to formally buy into this, to say what are we actually gonna do." The core team had a few opinions and said a few things, came up with a few new ideas. And when there's a dispute among the core team essentially we fall back to the original two core developers, Adrian and Jacob - following the naming from Python they're called the Benevolent Dictators for Life - to make the final call. And we said: "Okay, here's seven options: This one has almost no impact, this one has incredibly high impact, this one is gonna at least require changes, and so on, which one are we going to do?" And I said: "We are going to do this one." And now hopefully that's gonna turn into a body of work that will get committed for 1.5. Does that sort of explain how that process works?

MS: Yes absolutely, it does. But what happens with unpopular code, which has to be done but no one really has the time for, or the skills, or just doesn't want to do it at the time. Does that happen at all?

RKM: That is probably the reason why things don't get built when they need to be built. Essentially .. I suppose by unpopular code you mean things like tests and documentation and the support code and things like that?

MS: Not only that, but..

RKM: If you had a specific in mind?

MS: Also like a critical bugfix that has to be done, a security fix that has to be done but no one has the time. Or is the commitment from the core developers then kicking in?

RKM: Essentially there's commitment from the core team. That's part of the reason why people get added to the core team, because they demonstrated that they're going to be active in the project for a while. And at this point, you know, if someone goes out in public and says Django is insecure, my career is kind of on the line for that. Because I have put so much of my effort into Django, and my name is one of a very small number that is on the door as someone who can commit. So, if it's a security patch, we got a bit of personal responsibility from the core team to look after that. If it's a significant bug, generally it falls back to the person who originally committed it. If it's a significant bug that gets discovered in code that has been lingering for very long time essentially the reading of that is that it can't be that significant. So, significant bugs are things you actually going to hit all the time, and there is a difference between a use case that I hit all the time, and a use case that everyone hits every time. So if something looks like it is going to become high profile as in "Here is a bugfix that caused a high profile customer to lose a lot of data", then that kind of again comes back to the security problem where the core team has their reputation on the line to sort of address it. But unless it's actually one of these reputation-modifying things, essentially nothing gets committed to core unless it is ready to be in core. So there shouldn't be any major bugs in the code, by the time it actually gets there. And when you review their code and then you discover there are bugs the person who added the code is the person who is responsible for fixing them. It doesn't get committed unless it's got tests, it doesn't get committed unless it's got documentation, it doesn't get committed unless it looks like the tests actually test everything that need to be there. So we slightly avoid the problem you were describing by essentially saying you don't get into Django unless you've met these basic criteria. And the rest falls back to the core team responsible for the project looking serious and looking professional over time.

MS: Okay, I see. So, what's the worst that could happen to the project? Are there any major risks on your mind, and countermeasures planned for emergency situations, whichever could arise?

RKM: There are areas of code, where we have a very low bus factor, so if - a couple of years ago, if someone had driven over myself and Malcolm Tredinnick in a bus there would have been very few people who would have known the internals of the database code well enough to be able to fix it straight away. That has been largely mitigated by bringing in more core developers who have spent time in the database code, so that situation has been improved. Worst case scenarios would be things like a very high profile Django site being hacked for a reason that wasn't an esoteric problem, it was a fundamental problem in Django's security. We never know what security problems exist until some finds them and exploits them, but we have a fairly good historical reputation with security, and we've been fairly proactive in terms of fixing security problems and forcing security fixes down on people over time. I suppose the biggest issue would be if the active members of the core team stopped being active. The are twenty-odd people who have the commit-bit, but any given time there's probably only a dozen, maybe even down to less than six, who are actually actively committing code, and if those people get bored, or go off somewhere else, or have other commitments, or anything else happens, then development on Django stalls because that is our bottleneck. The core team is our bottleneck.

MS: So is the knowledge on the team held implicitly by the core developers?

RKM: There is a lot more knowledge about Django's internals than is held purely by the core team itself. The metric I'll have to use on that is that I'm in Australia, on the west coast of Australia, I know any number of people who are using Django. Almost none of them are in the commit-file or even on the Django mailing list, but most of them know the internals of some of Django. Because it's an open source project, because it's Python it's very easily to read, very easily to get in to, so there is a lot of knowledge about Django that isn't specifically held in the core team itself. The core team obviously is a group of people who do know the internals very well, but they're not the only people who know the internals very well. I'm trying to think of other things that could be a problem- If both of the BDFLs decided they wanted to disappear completely, that might be a minor problem because we'd had to find a new way of [] resolving ties. If our formal release manager got hit by a bus, then we'd need to find all the people who are on our actual notification list and renegotiate GPG key signing, things like that. At the moment we have one person who has the keys who can formally sign a Django release and knows who to notify when that release happens.

MS: But whatever emergencies could come up you decide when they really arise how to deal with them?

RKM: Yeah, I mean we don't have a formal disaster recovery plan if that's what you're asking. It's mostly because - and this is one of the redeeming characteristics of [] - if the worst happened, if every single member of the core team got hit by a bus, and the Django website went down, and all the worst things that could possible happen to the code happened to the code, you'd still have the code. And it would be sitting in thousands of repositories around the world, and someone would be able to find a way to get the most recent revision up somewhere, and I'm sure there would be enough of the community around to start developing the next round of patches, the next round of updates, and so on.

MS: And would it be possible to retrieve the knowledge about the code from the code and the documentation?

RKM: Almost certainly, yes. One of the things about Django is that does have extraordinarily extensive documentation. If you printed it out, I think at last count it ran to something like 1,200 pages if you would physically print it all out. So it's very well documented. There are some areas the documentation could be better that's for sure, but most of Django is very well documented. And internally it is also very well documented on the parts that are difficult, and it's very well tested in terms of the things that are complex, and a really good branch coverage in terms of the things that are complex. So I've no doubt that if you got someone who is left dependent on actually being able to progress Django and fix Django's core and they didn't have access to the core team, they would be able to fix those problems, it's not a complete black hole.

MS: Okay. I see we almost already ran out of time, so a quick question about the architecture. It says on your bio that there were some major refactorings you were involved in. How much does the architecture change or was the code structure quite stable over time?

RKM: The core structure has been very stable over time. There was one massive rework, which happened within about three months of me joining the project. When I joined the project, we were in the middle of a change called the 'magic removal'.

MS: Oh yes, I read about that.

RKM: Yeah, it was getting rid of a bunch of insider tricks that worked, but weren't very good engineering. So that was a massive rework of all sorts of parts of Django. Since then, there's been one semi major change. We made a big change to the forms-framework, but that was still before version 1. Since version 1 has come out, there's been almost nothing that I would classify as a major change. There have been very major features added, but it has been a very intentional position of the core team that backwards compatibility is core. People developing Django projects expect them to live for a very long time and not have to every minor release go and refactor all of their code because something changed, positional value of an argument changed, something like that. So we have a very firm policy that if you are upgrading from Version 1.2 to Version 1.3 the code must just work, unless there is a fundamental reason why it cannot. And that usually means there is a security problem, or something very very serious that is wrong with the original. And even then we try to make the changes in a way that if you update the code, you'll get warnings telling you what it is you need to update, and consequences will happen if you don't update. So architecturally the core framework, the core structure of Django, has been basically stable, or has almost completely stable for four years, and relatively stable for quite sort of five, almost six. And the intention is it's going to stay that way, we have enough of an installed base, that we cannot make a major change without causing a lot of people to be very pissed of. And one of the reasons that I attribute to why Django has been able to maintain or has been able to gain popularity in a lot of very large organizations is that policy. Some projects say it's perfectly okay to completely change the meaning of something between a minor release. We don't. Which means that large organizations like NASA can say: "Okay, we gonna deploy this now" and know that it's going to be exactly the same for the next four years without any surprises popping up over time.

MS: So is it a modular structure, or how does it allow for additions to be made on the stable architecture?

RKM: Well there's sort of a core architecture which is basically just describing the things that Django is. It is a database model, it is a collection of contributed apps that make building web apps easier, there are some testing systems, there are some form management systems, and they are all just high level modules. And inside that, yes, we made modifications to the form system, and we add a new submodel inside forms that makes that happen. One example: in 1.3 we added a new generic views framework, there's this idea that most web pages are essentially: "Show me a list of stuff, show the details for one thing in that list, allow me to edit for one thing in that list." That generic concept was wrapped up in a series of views. In 1.3 we replaced all of these views with a new version of a more easily expendable class base that could be easily modified and so on. It's now Django 1.4, you can still use the old views. They are in a package, it is labeled, and if you try to import that package now, you get a warning that pops up that says: "Hi, these views have been deprecated you may need to update. If you want to update, you can do this update. Here is the sequence of steps to go through, it's all documented it's got what you need to modify and how you modify them." But if you are building a new project from scratch you can just use the new views from scratch, import them from the new module and on you go. So we do explore the fact that you can have modules and submodules to enable that sort of migration to happen, and then we use this sort of deprecation process to get people away from the old modules and into new modules. And essentially, once we formally decide to deprecate something, you've effectively got about 18 months to get your code off of the old version before we completely cut off for the old version.

MS: That's quite a while. Well, thanks a lot, you answered my questions wonderfully. Do you have any things to add on your own, any agenda, any more points coming up?

RKM: No. I suppose the only thing is, I mean I said it before, but it really does bear repeating, is that there is a fundamental difference between the way commercial projects work, the way commercial software engineering, planning, resource management works, and the way open source projects have to work, by necessity. Now I cannot compel anyone to do anything. I cannot build an oak chart that says that I am gonna tell someone to finish this thing in two weeks. The best I can do is offer them an enticement to say: "If you do this in two weeks, then I will do this for you." And that fundamentally alters the dynamic for how you manage a project. It really does change what you can realistically achieve and the timelines of when you can achieve things. And often it means that you have to say: "No, I just can't do that because I can't force someone to do that for me. I've either gotta do it myself or live with the fact that it doesn't exist."

MS: I've researched quite a bit into agile development methods as one of the things I was comparing in my thesis and there is also a tendency for the methods of project managers to change away from assigning work to enabling people to do what they choose to do. Do you see connections to this type of management?

RKM: Very much so. It's essentially the ultimate expression of that. Because we literally cannot force anyone to do anything, we can only entice them to do something and make it interesting for them. And sometimes that actually means that you've actually got to go out of your way to make something really really bad in the hope that someone will step up and do the hard job. You say: "We are not going to commit a half-assed solution to the usermodel problem, we need a proper fix. So go off and work on a proper fix." So yeah I would completely agree: The modern management is all about making your employees happy - we have no option but to make our employees happy.

MS: The interesting thing is that the agile development methods work in a small and spatially very concentrated team of like ten people in one room constantly communicating with each other while open source development teams are spread all over the world.

RKM: Spread all over the world but never the less communicative. Big communities, we generate a lot of mail on django-dev, we generate a lot of mail on django-users, the IRC-channel is almost always populated, there is a strong social aspect, even outside. There is django-dev, the IRC-channel, but there is also the IRC-channel django-social, where people just go and kind of hang out with other Django people, so there is a very strong community aspect. And whilst they are not necessarily geographically together there is a very strong sense of keeping in touch with each other.

MS: And the technology is enabling this communication?

RKM: Yes, sure.

MS: Well, thanks. Thank you for the conversation.

APPENDIX B

Interview: Drupal

Martin Schönberger: Hallo erstmal und danke dass du teilnimmst an dieser Case Study. Kannst du dich am Anfang kurz vorstellen und erzählen wie deine Arbeit mit Drupal aussieht, in welchen Bereichen du aktiv sind und wo deine Schwerpunkte liegen?

Klaus Purer: Mein Name ist Klaus Purer, ich bin im Drupal-Projekt seit 2008 mehr oder weniger aktiv, habe davor schon Drupal benutzt als User und bin seit 2008 auch Developer. Ich arbeite in einer Wiener Firma die sich hauptsächlich mit Drupal beschäftigt, wir machen Web-Projekte. Unsere Firmenphilosophie ist, dass wir sehr stark mit der Drupal-Community zusammenarbeiten, das heißt Entwicklungen die wir machen geben wir dann wieder als Module frei für die Community und nehmen uns dafür auch andere Module, kombinieren Arbeiten von anderen Leuten, und so schaffen wir es eigentlich recht effizient zu arbeiten und recht schnell unsere Projekte durchzuziehen. Meine Aufgabe bei dem Ganzen ist: einerseits bin ich ein bisschen im Systemadministrationsbereich tätig, das heißt ich schau dass die Seiten laufen, schau dass die Konfiguration passt, alles was da zur Verwaltung dazugehört, und bin auf der anderen Seite Developer, das heißt ich schreibe Module, ich mache Arbeiten im Back-End, PHP-Programmierung ist das eben bei Drupal. Im Vergleich dazu gibt es bei Drupal auch andere Rollen, Themer, Site-Builder, aber ich bin eher so der Hardcore-Developer.

MS: Es gibt hier also eine Aufteilung zwischen den Rollen, unter denen Aufgaben verteilt sind?

KP: Ja. Also Drupal ist im Web-Development-Bereich ein bisschen berüchtigt dafür, sogar eine neue Rolle erfunden zu haben. Das ist bei Drupal der Site-Builder, das heißt es gibt nicht nur Leute die Code schreiben und Designs entwickeln und die in HTML umsetzen sondern es gibt auch eine dritte Rolle die sich eigentlich nur damit beschäftigt, Drupal zu konfigurieren, Sachen im User-Interface zusammenzuklicken und die Seite zu betreuen über das Administrationsinterface.

MS: Verstehe, das ist ja doch eine eigene Aufgabe bei der einiges zu tun ist.. Ich interessiere mich auch sehr für den Entwicklungsprozess, wie die Abläufe sind. Kannst du mir vielleicht so einen Ablauf in der Projektentwicklung beschreiben, wie die typischen Schritte sind von der ersten Idee bis zur fertigen Umsetzung?

KP: Also im Prinzip wird Drupal immer wieder bezeichnet als 'Do-ocracy'. Das heißt wenn jemand etwas machen will und das durchzieht, dann wird es auch meistens bei Drupal aufgenommen und hat Erfolg. Es gibt keine starren Strukturen die irgendwie vorgeben was die Roadmap ist oder fix erledigt gehört, sondern Leute haben Ideen und versuchen die dann umzusetzen. Und wenn es Erfolg hat, dann fließt es in Drupal ein und wird so weitergegeben an das ganze Projekt. Prinzipiell interagieren die Leute ja weltweit miteinander, das heißt es läuft alles in der Issue-Queue ab, das ist der Bugtracker auf drupal.org. Dort werden die Ideen gepostet, dort können die Leute kommentieren, dort wird der Status festgelegt für diese Idee, die kann zum Beispiel 'aktiv' sein, oder es kann schon ein Patch zur Verfügung stehen, also Code der diese Änderung implementiert, oder es kann noch Arbeit brauchen, oder es wird wieder geschlossen weil es nicht relevant ist, oder andere Gründe. Sozusagen die zentrale Anlaufstelle für die Koordination im ganzen Projekt ist drupal.org mit dem Issue Tracking. Ein zweiter sehr wichtiger Kanal bei Drupal ist der IRC-Chat, wo sich die Leute online zum Chatten treffen um schneller miteinander zu interagieren - um sich nicht über lange Kommentare auszutauschen, sondern eben direkt im Chat. Es gibt natürlich auch Skype-Calls zwischen einzelnen Personen, aber es gibt das zentralisierte Chat-System damit sich die Leute schneller austauschen können.

MS: Also eine Kommunikation auf unterschiedlichsten Kanälen. Ist die Arbeit dann aber eher aufgeteilt, und wie eng ist die Zusammenarbeit innerhalb der einzelnen Module oder Bereiche?

KP: Bei Drupal gibt es den Drupal Core, das ist sozusagen was jeder in der Drupal-Welt braucht, das ist das Herzstück das man sich als erstes installiert, und dann gibt es noch ein großes Universum aus Modulen. Der Core wird von einigen wenigen Personen maintaint, die darauf aufpassen dass nur die ausgereiften Features hineinkommen, dass die Bugfixes hineinkommen, ganz normale Wartung eben. Und dann gibt es einen großen Bereich auf drupal.org, die sogenannte Contributed Module sind, das heißt es gibt da wieder eigene Maintainer, die ihre eigenen Module schreiben, die diese dann als Projekt bereitstellen und für das Modul verantwortlich sind. Es gibt also eine hierarchische Aufteilung zum Core wo einige wenige Personen verantwortlich sind, und dann eine große flache Hierarchie von gleichberechtigten Modulen, für die dann wieder andere Einzelpersonen verantwortlich sind.

MS: Okay. Und wie ist so die Kommunikation zwischen dem Core und der Community, wie wird es sich da dazwischen ausgetauscht, inwieweit richtet sich die Community nach dem Core und umgekehrt?

KP: Ich würde sagen das ist so ein Geben und Nehmen. Der Drupal Core muss natürlich sehr gut funktionieren, das heißt es bleibt wenig Spielraum für Experimente. Wenn man also so sagen will: Innovation und wirklich radikal neue Sachen werden sehr oft im Contributed Space, also bei den Modulen, zuerst entwickelt, bevor sie dann später wenn sie gut funktionieren irgendwann im Drupal Core landen - oder auch nicht. Es gibt auch einige Beispiele von Modulen, die explizit nicht in Drupal Core hinein genommen werden, weil sie eben nicht für jeden Use Case geeignet sind. Andererseits muss natürlich Drupal Core auch gewisse Regeln vorgeben und einen gewissen Common Ground schaffen, was jeder in der Community braucht, das heißt es passieren sehr wohl auch neue Entwicklungen in Drupal Core, damit überhaupt die Module rundherum gut funktionieren können. In der früheren Geschichte von Drupal - Drupal ist jetzt zehn Jahre halt - ich würde sagen in den ersten sieben Jahren oder so, war das wirklich sehr chaotisch. Es war nicht sehr vorherbestimmt was in Drupal Core aufgenommen wird, was nicht, und wo eigentlich der Fokus liegt, was hineinkommen soll oder was in Drupal Core neuentwickelt werden soll. Als das Projekt dann gewachsen ist, hat sich das ein bisschen verändert zu sogenannten Core Initiativen. Die werden jetzt vor allem bei der Entwicklung von Drupal 8 eingesetzt, da gibt es so spezielle, ich würde sagen kleine Task Forces, die sich dann um gewisse Bereiche kümmern die zur Zeit in Drupal noch sehr schmerzhaft sind, und wo viele Entwickler darunter stöhnen, und die dann im Core vorantreiben, damit das wieder eine gemeinsame schöne API ist um gute Module darauf aufsetzen zu können. So gibt sich die Innovation und die Neuentwicklung ein bisschen die Hand.

MS: Ich habe mir die Keynote angehört, die Dries Buytaert im März gegeben hat, in der er die Pläne von Drupal beschrieben hat, eben auch von diesen Core Initiativen, und wie sich das Projekt in den nächsten drei Jahren und danach entwickeln soll. Und da habe ich mich ein bisschen gewundert wie sich solche längerfristigen Visionen wirklich umsetzen lassen in der Community?

KP: Generell hat Drupal keine sehr starren Regeln was Entscheidungsstrukturen anbelangt. Es gibt einen Code of Conduct, der ist glaube ich von Ubuntu gestohlen, der beschreibt wie die Community miteinander umgehen soll, dass man nett zueinander sein soll, wenn man Probleme hat soll man andere Leute miteinbeziehen um das Problem zu lösen... Hast du den schon gefunden, den Drupal Code of Conduct?

MS: Nicht bei Drupal, aber ich kenne das prinzipielle System von anderen Projekten.

KP: Es gibt also solche generellen Community-Regeln wie man miteinander umgeht um Entscheidungen herbeizuführen und Probleme zu lösen. Dann gibt es eine ganz nette Blog-Serie von Randy Fey über Drupals Governance, ich schicke dir mal den Link (Link einfügen). Auf dem Blog gibt es einige Artikel darüber, wie es früher war, wie es bei anderen Open Source Projekten ist, und wie es bei Drupal sein könnte. Nachdem Drupal jetzt ziemlich groß geworden ist funktioniert das Konsensprinzip nicht immer. Das heißt: Eine gewisse Gruppe von Leuten ist sich einig in einem Issue, etwas durchzuziehen, und plötzlich kommt eine Person her und redet dagegen, dann war es früher meistens so dass das das ganze Issue aufgehalten hat, weil man eben versucht hat einen Konsens zu bilden in der Community. Jetzt bei einer sehr großen Community wird das natürlich immer schwieriger und muss man sich auch andere Entscheidungsprozesse überlegen wie man das Projekt vorantreibt oder in welche Richtung man geht und so weiter.

MS: Ah ist klar. Kannst du dazu vielleicht ein konkretes Beispiel geben?

KP: Zum Beispiel Support von Internet Explorer 7 war ein heiß diskutiertes Thema. Drupal 8 wird gerade entwickelt und wird wahrscheinlich nächstes Jahr releast werden, oder vielleicht übernächstes Jahr, und da kommt natürlich die Diskussion auf 'Soll Drupal 8 noch eine so alte Version vom Internet Explorer unterstützen, ja oder nein?' Dann kommen natürlich Leute aus Ländern wie zum Beispiel aus dem asiatischen Raum, aus China, wo teilweise Internet Explorer 6 und 7 noch sehr verbreitet sind, und die argumentieren dann natürlich dagegen in dem Issue im Bugtracker. Und so treffen ein bisschen zwei Fronten aufeinander, einerseits natürlich die progressiven Webentwickler die mit dem neuen Standards arbeiten und den alten Code raushauen wollen, und eine größere Usergruppe, die darauf angewiesen ist, dass auch zukünftige Versionen von Drupal noch mit ihren Browsern funktioniert, die sie eben nicht ändern können bei gewissen Firmen, weil diese die verwenden.

MS: Und hat man in diesem Fall eine Entscheidung gefunden, oder einen Kompromiss?

KP: Ja man ist mittlerweile doch in die Richtung gegangen dass Internet Explorer 7 nicht mehr supportet wird in Drupal 8, es haben sich sozusagen die Entwickler, die die meiste Arbeit erledigen durchgesetzt, und das letzte Wort bei solchen Issues hat bei Drupal dann meistens Dries selbst. Das ist ein ähnliches Modell wie bei Ubuntu oder anderen Open Source Projekten, es gibt so einen Benevolent Dictator For Life, jemanden der sozusagen der Chef vom Projekt ist und da gerade bei solchen politischen Entscheidungen, weil es ist eigentlich kein technisches Problem zu lösen jetzt ob Internet Explorer 7 supportet wird oder nicht, sondern es ist eigentlich eine politische Entscheidung, das bewusst zu machen oder bewusst nicht zu machen. Und gerade bei solchen Sachen hat dann Dries das letzte Wort, um dann eine Entscheidung herbeizuführen.

MS: Ja ich verstehe. Eine andere Frage zur Architektur - soweit ich gelesen habe verzichtet Drupal ja absichtlich auf zu viel Rückwärtskompatibilität, man hat starke Veränderungen zwischen den Major Releases, von 6 auf 7, von 7 auf 8... Wenn sich der Core stark verändert, wieweit wird man davon beeinflusst in der Entwicklung?

KP: Drupal hat in den letzten Jahren eine sehr interessante Veränderung durchgemacht. Früher waren Major Releases nicht weit voneinander entfernt, da hat es teilweise einen Releasecycle gegeben von einem Jahr, wo Module wieder geupdatet werden mussten. Mittlerweile hat sich das sehr verzögert, also Drupal 6 ist noch Anfang 2008 herausgekommen, Drupal 7 dann Anfang 2011, also 3 Jahre später, und Drupal 8 wird wahrscheinlich wieder so 3 Jahre später herauskommen, das heißt mittlerweile haben sich die Releasezyklen so verlängert dass man doch über einen gewissen Zeitraum eine sehr stabile Version von Drupal auf die man setzen kann. Die Philosophie bei Drupal ist: 'We protect and update your data, but not your code'. Das heißt die Daten die man aus einer alten Drupal-Installation hat kann man dann eh updaten, nur muss man halt darauf warten bis die ganzen Module wieder zur neuen Version kompatibel sind, oder vielleicht gar nicht mehr notwendig sind, oder sich sonst irgendwie verändert haben. Das hat aber auch die Konsequenz dass am Release-Tag von Drupal sicher sehr wenige Seiten erst diese Drupal-Version einsetzen. Es gibt also immer eine gewisse Verzögerung bis dann die Drupal Version so richtig abhebt, meistens ist das ungefähr ein Jahr. Bei Drupal 6 war es ungefähr ein Jahr, bei Drupal 7 war es wieder ein Jahr ungefähr, wo man dann merkt: 'Aha, jetzt zieht Drupal so richtig an'. Und die meisten Seiten updaten dann auf die neue Version, weil eben die ganzen Module dann geupdatet sind, weil es andere Möglichkeiten gibt wie man die neue Seite betreut, und so weiter. Andererseits gehen natürlich Firmen wie wir zum Beispiel den Weg dass sie Drupal-Versionen überspringen. Wir zum Beispiel haben eine Drupal-Seite sehr lange auf Drupal 5 laufen lassen, das irgendwann 2007 herausgekommen ist, haben Drupal 6 übersprungen, und haben sie dann auf Drupal 7 neu gebaut. Mittlerweile ist das nicht mehr so ein Problem, weil immer die aktuelle und die vorhergehende Drupal-Version supportet wird. Das heißt wenn eine Drupal-Version herauskommt, dann ist fast schon garantiert, dass die für sechs Jahre supportet werden soll. Vom heutigen Standpunkt aus. Früher war das natürlich um einiges kürzer, aber wenn man sich den Releasecycle von Drupal 6 anschaut, das 2008 herausgekommen ist, wird das wahrscheinlich immer noch bis 2013 oder 2014 verfügbar sein und supportet werden.

MS: Aber da muss man ja auch relativ lange Bugfixes nachziehen, oder? Entwickelt man aber prinzipiell auf der neuesten Version, oder wie viel Zeit verbringt man damit Probleme in alten Versionen zu beheben?

KP: Nicht so viel, würde ich mal sagen. Drupal hat einen extremen Stabilisierungsfaktor, bis ein Jahr nach dem Release geht es noch ein bisschen drunter und drüber, da gibt es auch noch schwere Bugs, es muss sich erst einspielen, die ganzen Produktivseiten nehmen dann erst die Version auf, und dann werden noch ein paar Probleme entdeckt die dann gefixt werden müssen. Aber dann kommt eine Phase, eine sehr lange Phase, die wir auch bei Drupal 5 und 6 schon mitbekommen haben, seitdem ich mit Drupal arbeite, wo es zu einer extremen Stabilisierung kommt. Da treten sehr wenig gröbere Fehler auf, man hat dann wirklich eine sehr beständige API. An die man natürlich auch gekettet ist, neue Features kommen ja keine mehr hinein in die alten Versionen, das heißt man kann nur mit Modulen rundherum arbeiten oder sonst irgendwie damit auskommen. Dafür hat man aber einen sehr hohen Stabilitätsgrad, und ich kann mich dann über einen längeren Zeitraum sehr gut auf die Drupal-Version verlassen, und habe auch sehr wenige Bugfixes und auch Securityfixes die ich beheben muss.

MS: Umgekehrt hat man aber in der Zeit in den neuen Versionen die absolute Neustrukturierung und Schaffensphase?

KP: Es ist keine triviale Aufgabe jetzt eine Drupalseite von einer Version auf die nächste zu bringen, weil sich eben der Code bei sehr vielen Modulen ändert. Meine Daten sind sicher, wenn es einmal Code gibt dann gibt es meistens auch die Update-Routinen um die Daten mitzunehmen, aber dass sich der Code von den Modulen wieder stabilisiert hat, und Drupal Core sich wieder stabilisiert hat, das braucht natürlich eine Zeit, und da es dann auch einige Umstrukturierungen gibt muss ich dann auch meine Seite umkonfigurieren und habe da schon ein bisschen Arbeit damit.

MS: Okay. Und die Prozesse während der Entwicklung, zum Beispiel das Testen? Es gibt ja im Projekt einen relativ umfangreichen Review-Prozess - ich habe gesehen du bist da auch ziemlich engagiert darin - wie funktioniert das so? Und was wird sonst noch gemacht um die Qualität vom Code zu erhöhen und sicherzustellen? Gibt es automatische Tests, oder ...

KP: Ja die gibt es. Mit der Version 7 wurden eben dann automatische Testfälle in Drupal Core eingeführt, und da gibt es jetzt glaube ich 35.000 Assertions die dann abgeprüft werden mit diesen Testfällen. Also Drupal Core ist mittlerweile sehr gut getestet, und auch die großen Module sind sehr gut mit automatischen Testfällen abgedeckt. Und der Reviewprozess von neuen Features und Bugs in Drupal Core funktioniert immer so, dass wenn ein Patch gepostet wird, dann wird automatisch der Testbot getriggert, das heißt dieses Patchfile wird an ein externes Botsystem geschickt, der Patch wird angewandt auf die aktuelle Drupal-Version, der Testbot führt die Tests durch, wenn die erfolgreich sind dann leuchtet das grün auf in dem Issue, wenn sie nicht erfolgreich sind dann wird das Issue auf 'Needs work' gesetzt, das heißt da ist noch Arbeit notwendig.

MS: Aber diese Tests schreibt nicht der Entwickler selber, sondern die sind im Core System drinnen, oder wie muss ich mir das vorstellen?

KP: Genau. Natürlich kann es sein dass ich irgendeine Änderung in Drupal Core mache, die auch erfordert dass ich den Testfall anpassen muss. Kann natürlich sein. Aber die sind vor allem dazu gedacht, dass man einerseits Unittests hat, damit man weiß dass abgeschlossene Einheiten richtig funktionieren, und dass man andererseits Integrationstests hat, damit man weiß: 'Wenn ich den Patch anwende, bricht nicht mein ganzes Drupal zusammen', oder 'Wenn ich an dem Ende irgendwo einen Bug fixe, dann ist nicht am anderen Ende wieder etwas komplett kaputt'. Darum hat man diese große Anzahl von automatisierten Testfällen, die das gewährleisten sollen, dass Drupal immer noch funktioniert, wenn Bugs gefixt werden oder neue Features dazukommen.

MS: Okay klar, das ist also ein kontinuierlicher Integrationsprozess auf die Main Version.

KP: Genau.

MS: Und der Review-Prozess, da wird also wirklich jeder Code der eingecheckt wird von jemandem angesehen, muss von jemandem angesehen werden?

KP: Ja. Es wird kein Code committet der nicht in einem Issue irgendwo gepostet wurde. Es wird auch kein Code committet der nicht im Status 'Reviewed and tested by the community' ist. Das heißt wenn ich einen Patch poste, dann kann nicht der Maintainer hergehen und den einfach committen, sondern er muss erst warten bis andere Mitglieder der Community sagen: 'Ja das funktioniert tatsächlich' und dann kann es der Maintainer committen. Bei Drupal Core ist das besonders streng, bei den Contributed Modules im Universum steht es natürlich den anderen Maintainern frei wie sie das handhaben. Da gibt es natürlich auch Fälle wo das die Leute früher committen. Da ist der Anspruch an die Qualität sozusagen nicht so hoch weil eben Drupal Core wirklich der Common Ground für alle ist und die Module haben halt eher mehr eine Schnelllebigkeit und wollen auch schneller entwickelt werden, deswegen ist der Reviewprozess da auch um einiges lockerer als bei Drupal Core.

MS: Wie weit zieht sich dabei der Release-Zyklus bis zu den Modulen durch, oder werden die eher kontinuierlich mitentwickelt?

KP: Die Module haben sozusagen eigene Release-Zyklen. Natürlich sind sie an Drupal Core gebunden, das heißt ein Modul ist immer zu einer gewissen Drupal Core Major Version kompatibel. Drupal sechs, sieben, acht, was auch immer. Und während dem Lifecycle von einer solchen Drupal Major Version bringt das Modul natürlich auch neue Versionen heraus. Es kann auch sein dass das Modul neu geschrieben wird während dem Lifecycle einer Drupal Core Version, das heißt da wird dann eine 2.x Branch aufgemacht die das ganze irgendwie anders handhabt oder vielleicht sogar eine 3.x Branch. Das ist dann den Maintainern im Contributed Universum freigestellt wie sie das handhaben.

MS: Okay. Eine andere Frage: Wenn ich neu zum Projekt dazukomme, wo kriege ich meine Informationen her? Ist es am Besten wenn ich mich einfach durch die Dokumente durchlese, oder wende ich mich lieber an bestimmte Personen? Wie ist so generell das Wissen über Projektdetails in der Community oder im Projekt vorhanden, oder wie wird das weitergegeben?

KP: Also generell wird natürlich versucht, möglichst viel zu dokumentieren. Drupal Core, und ich glaube auch die meisten Module, sind sehr exzessiv mit Kommentaren im Code kommentiert, und meistens sind die Kommentare im Code auch die bessere Referenz als irgendwelche Documentation Pages auf drupal.org. Speziell um neue Leute einzubinden gibt es dann eigene Sektionen auf drupal.org wo neue Leute sich Tasks aussuchen können, die sie erledigen können um ein bisschen Gespür zu kriegen für Drupal und um ein bisschen in die Community hineinzukommen. Es gibt einmal in der Woche Core Office Hours, im Chat gibt es da einen gewissen Zeitraum wo dann Maintainer anwesend sind und da können dann auch Neulinge hineinschauen, Fragen stellen und sich Aufgaben abholen. Also zur Koordination von neuen Leuten die gerade frisch ins Projekt kommen.

MS: Ich habe da auch etwas von einem Mentor-System gelesen?

KP: Gibt es auch. Das ist aber eher sehr locker organisiert. Da gibt es keinen straffen Prozess dafür, es gibt halt Leute die sich gewissen Einzelpersonen annehmen. Ich zum Beispiel beim Project Application Review Prozess. Damit ein Modul auf drupal.org veröffentlicht werden kann, mit einer schönen URL und mit Releases, muss es zuerst einen Review-Prozess durchlaufen. Das machen natürlich vor allem neue Contributor, die noch nicht Maintainer sind von irgendeinem Modul, die müssen zuerst ihr Modul zur Verfügung stellen damit das auf Security überprüft werden kann, auf Code-Qualität, und erst dann wird es auf drupal.org als volles Projekt veröffentlicht. Es gibt eben verschiedene Kanäle. Es gibt auch das Google Summer of Code Programm, wo Google ein bisschen Geld sponsert damit eben Studenten sich da beteiligen können und Aufgaben ausfassen die sie dann über den Sommer lösen und so auch in die Community reinrutschen und Kontakt dazu kriegen. Natürlich gibt es auch die vielen Konferenzen und Drupal-Camps, also Veranstaltungen wo sich die Leute irgendwo treffen, wo es Vorträge gibt und Workshops. Die gibt es immer wieder in ganz Europa, ich würde sogar sagen weltweit. Zweimal jährlich gibt es ja die DrupalCon, die Konferenz. Einmal in Nordamerika, einmal in Europa, heuer gibt es glaub ich die erste in Südamerika, und so wird versucht die Leute irgendwie im Real Life zusammenzubringen damit sie da was ausmachen können. In Wien zum Beispiel gibt es die Drupal Austria Group, die trifft sich einmal im Monat, und da werden auch Sachen besprochen und irgendwelche Aktionen gemacht, zum Beispiel haben wir gerade die Drupal Austria Road Show laufen, wo ein paar Leute vom Drupal Austria Verein in Österreich herumfahren und so kleine Vorträge machen und Drupal vorstellen um in den Bundesländern ein bisschen die Leute auf Drupal aufmerksam zu machen und ein bisschen die Community zu konsolidieren, wer denn da eigentlich aktiv ist, und mit wem man sich vernetzen kann.

MS: Auf die feste Community wird also Wert gelegt.

KP: Ja. Also im Vergleich zu anderen Content Management Systemen wie zum Beispiel Joomla gibt es bei Drupal nicht dieses ’Modul verkaufen’. Alle Module die es gibt sind auf drupal.org und die sind Open Source und frei verfügbar, währenddessen es bei Joomla soweit ich weiß, ich weiß nicht ob sich da in letzter Zeit was getan hat, gibt es sehr viele Module die man eben kaufen muss bei irgendeinem Anbieter und es gibt dort kein zentrales Repository wo ich alle Module finde, das sind dann andere Probleme. Aber Drupal hängt sich ja selbst den Spruch um: 'Come for the software, stay for the community.' Es wird immer wieder sehr großen Wert auf gute Community, gute Vernetzung, gutes Miteinander, Teilen, auf so etwas wird immer wieder Wert gelegt.

MS: Weil du Dinge wie Module verkaufen ansprichst, wie ist in Drupal der Einfluss von Firmen, und gibt es da Finanzierungsmodelle?

KP: Ich glaube der Einfluss von Firmen ist sehr groß bei Drupal. Da gibt es einige größere Firmen die eben Leute dafür bezahlen, Vollzeit an Drupal Core zu arbeiten zum Beispiel. Natürlich wird auch großteils das umgesetzt das umgesetzt bei Drupal was diese Firmen vorgeben. Das heißt Drupal, das eigentlich aus einem sehr kleinen Content Management Bereich gekommen ist, und klein angefangen hat, wächst durch diesen Firmeneinfluss auch immer mehr in den Enterprise-Bereich hinein, also es wird darauf geachtet dass Drupal gut skaliert, dass es an verschiedene Enterprise-Systeme leicht angebunden werden kann, dass es an andere Datenbanken angebunden werden kann, und so weiter. Da macht sich der Einfluss von Firmen, und natürlich auch das Geld, das diese Firmen für ihre Entwickler ausgeben, die dann diese Entwicklungen für Drupal machen, da macht sich das dann schon bemerkbar.

MS: Das bestimmt dann aber auch die Richtung in die sich das Projekt allgemein weiterentwickelt? Wie ist so der politische Einfluss von größeren, geldhabenden Betrieben und Gruppen?

KP: Ich glaube dieser Einfluss ist nicht zu unterschätzen. Es wird sehr viel Wert darauf gelegt dass Leute in der Community als Personen sprechen. Ich habe das noch nirgends gesehen in Drupal dass Leute explizit für eine Firma sprechen, sondern es ist immer noch ein sehr persönliches Verhältnis von Personen untereinander. Natürlich macht sich trotzdem bemerkbar wenn Leute Vollzeit an etwas arbeiten, an einem Projekt in Drupal. Da kommt dann natürlich auch was raus, und das ist dann auch ein Benefit für die Firma.

MS: Also persönlicher Kontakt, aber im Hintergrund auch die Firmen die das ermöglichen.

KP: Genau. Es gibt natürlich immer noch einen sehr großen Freiwilligenaspekt bei Drupal, Leute beschäftigen sich mit Themen aus reiner Neugierde, als Hobby, oder entwickeln aus anderen Motiven, das darf man glaub ich nicht unterschätzen. Aber auf der anderen Seite stehen natürlich sehr viele Firmen dahinter dass gewisse Sachen umgesetzt werden und vorangetrieben werden. Aber es gibt niemanden mit einem Firmenaccount auf drupal.org der jetzt als Firma posten würde, so etwas gibt es nicht.

MS: Okay, gut, dann sage ich dazu mal danke. Eine kurze Frage nach einer persönlichen Einschätzung: Ich weiß nicht wieweit du vertraut bist mit verschiedenen Projektmanagementarten und Prozessmodellen. Etwa agile Methoden, strukturierte Prozesse. Wo würdest du ein Projekt wie Drupal da sehen, gibt es Gemeinsamkeiten mit der einen oder anderen Richtung, oder kann man das gar nicht vergleichen?

KP: Hmmm. Ich glaube es gibt da einen sehr großen, wie soll man das sagen, Cultural Clash. Ich glaube der Hackerethos in der Drupal Community ist sehr groß, und ich glaube die meisten Leute sind sich nicht bewusst, dass sie eigentlich in der Issue Queue zum Beispiel agile Methoden anwenden. In Drupal hört man immer wieder: 'Talk is silver, code is gold.' Ich würde mal sagen damit ist nicht nur Reden gemeint, sondern auch Projektmanagementprozesse, das ist eigentlich zweitrangig, es geht darum Code zu zeigen, es geht darum was umzusetzen. Trotzdem, durch diese mittlerweile doch sehr raffinierten Prozesse die es auf drupal.org gibt, dass Tests automatisch ausgeführt werden, dass es einen sehr intensiven Reviewprozess gibt, dass es Initiativen gibt was denn jetzt eigentlich in die nächste Drupalversion hinein soll und was nicht, wird es andererseits schon wieder durchbrochen eigentlich, dieser reine Hackerethos vom Coder, von diesem typischen Coder der eben keine Prozesse braucht oder will, weil es im Hintergrund eben doch wieder einen gewissen Projektmanagementprozess gibt. Aber ich glaub er ist doch versteckt und nicht sehr offensichtlich für die Leute.

MS: Ja da stimme ich zu, und danke für diese Einschätzung. Gibt es von deiner Seite irgendetwas was du noch gerne loswerden würdest zum Thema, irgendein persönliches Anliegen, oder haben wir etwas vergessen was den Entwicklungsprozess betrifft?

KP: Was mir noch einfällt zur Dokumentation: Auffallend bei Drupal ist dass es sehr wenige Diagramme gibt. Es wird fast nirgendwo etwas gezeichnet um etwas klarzumachen. Ich glaube es ist ein großer Schwachpunkt von Drupal dass es nicht mehr auf - es müssen nicht unbedingt UML-Diagramme sein - aber ich glaube Drupal ist mittlerweile ein sehr komplexes System, und ich glaube gerade im Dokumentationsbereich, was die Aufbereitung von der Architektur betrifft, ist Drupal schon noch sehr hinten nach. Ich glaube da könnte man einiges tun um es den Leuten, den Entwicklern, klarer zu machen wie Drupal funktioniert und wie es eigentlich tut.

MS: Ich stelle mir das aber auch schwierig vor durch die Verteiltheit der Leute, durch die Nichtanfassbarkeit der Prozesse, hier solche visuellen Modelle aufzustellen.

KP: Ich glaube dass es viele Leute gäbe die das gerne machen würden, oder die so etwas sinnvoll machen würden, aber ich glaube es bleibt sehr wenig Zeit dafür übrig. Die Leute sind einfach damit beschäftigt ihre Module zu warten, ihre Patches zu posten, ihren Code durchzupushen, und wenn sie dann mal in Drupal drinnen sind haben sie natürlich eh das Verständnis für das System und haben auch nicht das Bedürfnis das niederzuschreiben, und so bleibt das dann erst recht wieder auf der Strecke. Was ich gesehen habe ist, dass Dokumentation am besten funktioniert wenn es Leute machen, die gerade dabei sind Drupal komplett neu zu lernen. Wenn die dann mitschreiben, was sie gelernt haben, dann kommt eigentlich die beste Dokumentation raus.

MS: Gibt es da eigentlich Richtlinien, wie die Dokumentation auszusehen hat?

KP: Gerade für Code gibt es sehr strenge Regeln. Für Documentation Pages gibt es weniger strenge Regeln, es gibt auf drupal.org ein Documentation Team, das kümmert sich ausschließlich um die Dokumentation um sie zu organisieren, schauen was fehlt, alte Seiten upzudaten, etc. Es gibt schon ein gewisses Wertlegen darauf, aber nicht überschwänglich würde ich mal sagen, die Dokumentation ist sicher keine Stärke von Drupal.

MS: Ich verstehe. Gut, dann sage ich vielen Dank für die Unterstützung und für deine Einblicke in die Entwicklung von Drupal!

APPENDIX C

Interview: XWiki

This record of the interview held with developers of XWiki is presented in a version that has been slightly edited for better readability and spelling. The original mail-exchange can be found in the April to June 2012 archives of the project's developer mailing list at [65].

Martin Schönberger: As far as the process is concerned directly, which are the parts of development that profit most from the fact that XWiki is open source?

Ecaterina Moraru: Regarding this topic there is also a blog series written by Ludovic where you can find some of the answers from the company perspective: `http://www.xwiki.com/xwiki/bin/view/Blog/XWikiVisionOpenSource`

Guillaume Lerouge: We get plenty of testing of all parts of XWiki from plenty of people, some of whom contribute specific fixes that would otherwise not be worked on.

Vincent Massol: Obviously it's hard to contribute to the core of an open source project than it is to contribute to extensions/plugins. This is the case with XWiki. Actually what we've started doing a few years ago and this is still underway is to split the monolithic XWiki code into small modules, each implemented a specific feature (tag, querying, wysiwyg editor, rendering, etc). We have hundreds of such small modules which makes it easier to contribute than before. In addition we've started to make all those modules extensions, i.e. features that can be added/removed from an XWiki runtime. And we now have an Extension Manager in the XWiki runtime to manage all extensions. This means that when someone contributes an extension on extensions.xwiki.org it's directly visible from within everyone's installed XWiki runtime, in exactly the same way as extensions created by the XWiki Dev Team. Thus I believe we'll continue to see more and more contribution in the area of extensions.

We have several layers of code contribution: `http://dev.xwiki.org/xwiki/bin/view/Community/Contributing#HContributeCode`. (Note: I've updated `http://dev.xwiki.org/xwiki/bin/view/Community/Contributing` this morning)

Now more generally the XWiki open source project benefits from:

- extensions
- testing by the community and bug reporting
- discussions on the mailing lists to give ideas, to direct our work

MS: It seems like XWiki has a rather large core team closely working together, and the Hall of Fame lists rather few external contributors. Does this reflect the actual distribution in the project?

GL: Yes and no. Sure, most full-time XWiki developers are paid by XWiki SAS, but it's also because the company went to hire active community members. There are also some former employees who are still contributors although no longer employed by XWiki SAS.

VM: The list is valid for the committers. We're about 15 active committers (ie. XWiki Dev Team). We've always had a hard time identifying contributions since there are lots of ways to contribute. Thus we asked contributors to add their names to the Hall of Fame page but lots of people are shy or simply don't think about putting their names there. We've now moved to GitHub and it's much easier to recognize code contributions since we now use pull requests. I'm preparing a new section for the hall of fame that'll list all code contributors (and not only committers).

MS: What are the costs and benefits of using open source development methods in this situation? Would communication patterns, knowledge management and development cycles be approached differently if XWiki was developed purely in-house?

GL: Not really. The same processes would be used, only limited to internal mailing lists.

VM: Definitely. One important difference is that there's no power hierarchy in XWiki development. All the committers are equal. We have built our rules on the Apache model which is a Meritocracy. See `http://dev.xwiki.org/xwiki/bin/view/Community/Committership` and especially our vote strategy. It's enough that a single person doesn't agree for something to not happen.

EM: Regarding the other questions all that I can say is that I would love to have much more external contributors. Some of us start as independent contributors and get into the company, others work for companies related to XWiki and give back to the community in a form or another (tests, bug reports, improvements requests, community feedback and help, translations, documentation, etc.) The Hall of Fame reflect just long time participants, but the sum of all contributions should be made with all the users from l10n.xwiki.org; jira.xwiki.org; github.com/xwiki; xwiki.org, etc. In my case, first I contributed to XWiki as a GSOC student and after that period I was offered a position inside the company. And I am not the only case, same happened for other GSOC students like Eduard, Ana-Maria, Asiri, Sergiu, etc. So IMO getting from an external contributor to an internal one is great. From my perspective (as an Interaction Designer) is great that I can work in an open source environment. I get to receive issues (bugs) reports from all over the world, I can ask our users what improvements they would like to see and what they think of my proposals. Sometimes is very hard to make everyone happy, but since I am the only one in the company that is responsible with this topic, if we didn't have this open environment it would be impossible to do my work: being in a remote team and being without a way to reach the users and learn what they need, I couldn't know what to do. Also, being in the open I can reference my work and get critics on it (and I would love an even bigger community so that I could learn even more things).

MS: Another thing I'm interested in is how the scope of the project and the direction of development are decided on. To what degree do different stakeholders influence the course of XWiki, what is caused by personal itches of the developers, requests of paying customers, or complaints and suggestions of casual users?

GL: All of those factors play a role. Some customers pay for the development of very specific features or extensions. For instance, the Office document Import feature was paid for by a client. There is an internal roadmap process at XWiki SAS where stakeholders from every department can voice their priorities (sysadmin, dev, project management, sales, marketing, research...). Once this has been done, a roadmap is proposed, that members of the open source community can add to if they wish. In addition to this, about 40-50 % of the time of core developers is preserved so that they work on maintenance and bugfixing tasks, which they choose what to spend on.

MS: Is there a special time in the release cycle when new content is agreed upon? How far and in what detail can the content of future releases be planned ahead by the developers? Are detailed plans desirable, or is it more advantageous to react to circumstances when they arise?

EM: Our development is made in release cycles. We do one cycle per year (for example we are now developing the 4.x cycle and we just released 4.0, so now we work on 4.1, etc.) In one year we get to have about 6 major releases per cycle (4.0-4.5). For each major release we have a Roadmap meeting, before the release starts, where we decide on what we work on. Check out `http://enterprise.xwiki.org/xwiki/`

bin/view/Main/Roadmap For each roadmap we vote on the mailing list the issues we work on, the release dates, we vote if we need to delay a bit the release because of certain problems, etc. So the decisions are transparent and everyone can give their opinion on them and interfere. A very-very important aspect is that we do timeboxing releases instead of feature-driven releases so this IMO is a big differentiator between being an open source company and a closed source one (and wait indefinitely for a certain feature). Read more about at http://xwiki.markmail.org/thread/s5wajg23uhqmtnyh. Another important aspect is that even if we are a company, we have very clear departments. We have a clear difference between who is working with our paying customers (Clients Team) and who is working for the platform (Tech Team). So even if we collaborate inside the departments we have different purposes and different stakeholders. For the Tech Team the most important stakeholder is the community. Of course we care about what the Clients Team asks, but IMO we consider them as part of the community and as XWiki users (the only difference is that maybe their complains reach faster since this complains can be made in person).

GL: Given the very frequent release cycle, meetings take place often. There is a new large release every year, interspersed by major releases every 2-3 months. There is a roadmap meeting for each of those, thus it's easy to gather feedback and make priorities evolve along the way. For a large release, a theme and a sub-theme are agreed on that set the general direction of that release. For major releases, a list of JIRA tasks is agreed on.

VM: The strategy:

- In the open source project, there are only individuals, no company. At the start of each release committers and contributors can state what they'd like to work on for this release. We publish this on our roadmap page: http://www.xwiki.org/xwiki/bin/view/Main/Roadmap (especially http://enterprise.xwiki.org/xwiki/bin/view/Main/Roadmap for XE).
- At XWiki SAS, we do internal roadmap meetings and then find/assign developers who're going to be the owners of issues/features on the xwiki open source project. During these internal meetings we have representants of all XWiki SAS domains (marketing, customer project, sales, tech, research, infrastructure,etc) present and decide in a collegial manner.

MS: The third question concerns specialized tasks surrounding the development. XWiki follows diverse strategies for testing. One of them is manual, formal testing executed by a dedicated QA team.

VM: Actually we don't have any notion of QA team in the xwiki open source project. At the open source level each developer is responsible for the quality of his code and is supposed to write automated tests and do manual testing of his code.However we have one contributor named Sorin Burjan who's helping the developers do this by systematically testing new releases himself. He's helping us a lot. But he's acting as a contributor. Now Sorin is also an employee of XWiki SAS and within XWiki SAS his role is QA engineer and his internal role is to make sure that releases are of good quality.

MS: Does this part of the approach make the many-eyeballs-method of discovering bugs less important, or is a combination of these two strategies necessary for overall high quality of the code?

GL: open source users tests parts of the software not tested by the QA team, and vice-versa. Both methods complement each other.

VM: Combination. I'd say the vast majority of issues are still reported by the many eyeballs. But Sorin finds a good lot too! :) Both are needed.

MS: Could automated testing stand on its own with sufficient coverage?

GL: No. Some things cannot be tested automatically, especially on the look & feel side.

VM: IMO it could but we haven't reached a good enough level yet. We need to become better at this and start measuring better our test coverage. We have just started automating tests on various environments (DBs, browsers) and that'll take some time. In the meantime Sorin has taken charge of manually testing XWiki on those various environments. At some point we'll also need to add automated performance tests which is done manually too at this point.

MS: On a similar note, how do the different methods pursued in customer support (like detailed documentation, peers helping each other, and specialized paid support) interact and draw upon each other?

EM: Regarding this question, I just want to say that we are a small team. We have just a single person as a dedicated QA so all the help he can get is welcomed. For example, I test and report a lot of issues/improvements even though I am not part of the QA team. Another thing is that XWiki is multi OS, multi browser, multi database, multi display, multi whatever compatible :) We always need more eyes, hands, automated tests, manual tests, any kind of tests to verify the quality and then ... we always need more committers to fix them.

GL: Documentation is updated based on the most frequent requests from customers. Paid support users are usually different from mailing list users, although some of the latter do convert to the former from time to time.

Sorin Burjan: Yes, as Caty and Guillaume said, the more eyes are looking, the better the Quality of the product is. I am the QA responsible of testing XE and XEM, but a lot of other bugs are reported by the community. We have a formal Manual Test Plan, you can check `http://www.xwiki.org/xwiki/bin/view/TestReports/WebHome`. Besides this, we have also an automated tests suite maintained by tech team. We don't have a dedicated team for Testing Automation. For example, there are certain things I always look for, for example: cross-browser compatibility, database migration, etc. The role of the community is important because we have a lot of extensions or use cases which we don't have time/resources to test, but the users which installs them write on the mailing lists, report issues or even provide patches if they find bugs. Usually when a user from mailing lists has troubles accomplishing what he/she wants, we check if that is a valid use case, our developers being very responsive in aiding them. If the use case is valid, this usually goes documented in the according place, so other users wanting to do that, won't have to write on the lists. So the QA is done collaboratively, not only by a dedicated team or individual. Hope this helps.

VM: XWiki SAS internal support will raise issues in our issue tracker like anyone else and these will get fixed eventually. In general the most people who report an issue and faster the XWiki Dev Team spends time on fixing it quickly.

[...]

MS: So my first question this time concerns the architectural design, and how it has evolved over time. Did the basic structure change / grow significantly since the early days of XWiki? Vincent mentioned the change from a monolithic code to small modules. Were these and other changes made in some major refactorings, or rather through steady refinement? To what degree is it an issue to keep backwards compatibility between releases?

VM:We've decided a long time ago (we = the committers at that time) that we preferred evolution rather than revolution. The main reason is that XWiki already had an important user base when the development team was started (before that it was just Ludovic working on it). It takes more time but it allows us to keep existing users happy. In addition we've been able to still move forward. There's a JoelSpolsky blog about this topic: `http://www.joelonsoftware.com/articles/fog0000000069.html`

I don't have any figure I can prove but I'd say we've modified a bit more than half of the code in a 5 years time span (maybe up to 75%). I can cite 2 examples of relatively large refactorings we've done without breaking backward compatibility:

- The Rendering module. XWiki had a wiki syntax initially and we wanted to fully rewrite the code that takes an input in that syntax and generate HTML out of it. So we created a new Rendering engine that can take any input in any syntax and generate an output in any other syntax (see `http://rendering.xwiki.org`). And we added polyglot support in XWiki (i.e. the ability to support several wiki syntaxes), and introduced XWiki Syntax 2.0.

- The WYSIWYG editor. We were using TinyMCE and switched to our own editor that we wrote from scratch. If you're interested we can explain the reason...

Then we have a lot of smaller refactorings. We can go into more details if you're interested. The only big part that hasn't been refactored yet is the core Model. I'd really like to tackle this and we've started designing the new model a bit but it's complex and requires a long period of focused time which I personally haven't been able to get so far. We're very serious on backward compatibility and most wiki pages back from 5 years ago still work on the latest version. We've defined a precise strategy to ensure we don't break backwardcompatibility: `http://dev.xwiki.org/xwiki/bin/view/Community/DevelopmentPractices\#HBackwardCompatibility`

If you're interested in how we came up with this, it should be easy to find the mailing list threads about it on `http://xwiki.markmail.com`

GL: I'm not very qualified to answer this one, but here goes. XWiki has had user that started using the software in its early days (I have an example from around 2006 in mind) who are still using the software today and successfully managed to upgrade from version 0.9 to version 3.5.1 and still have their wiki functional. XWiki has many enterprise users, for whom backward compatibility is very important. So the XWiki dev community has always been careful not to break too many things at once and provide an upgrade path from one version to the next. Major changes (such as the switch to a new syntax) came with a migrator to help handling them. It is sometimes tough to maintain backward compatibility, but it's a major priority of the dev team.

MS: Also I am interested more closely how the functions and responsibilities are divided in the team. Caty wrote about 'very clear departments' in her last answer, and the teampage on XWiki.com lists a multitude of different and specific roles. Both of your descriptions of the testing process, however, suggest a less strict separation of tasks. So what role do the roles play? How specialized or cross-functional are the teams and people working therein? Is there a difference between XWiki.org and XWiki SAS?

VM: At the open source project level we have only 2 roles:

- contributors: anyone contributing as defined here: `http://dev.xwiki.org/xwiki/bin/view/Community/Contributing`
- committers: people who have right access to the SCM and who can vote, see `http://dev.xwiki.org/xwiki/bin/view/Community/Committership`

Committers decide what they want to work on. Some can be more design-oriented, others more test-oriented, etc. At XWiki SAS we have more roles and a more traditional structure. Tech roles:

- CTO (me ;))
- Tech Lead or Senior Dev
- Web Developers
- Back end Developer
- Tester
- QA engineer
- Designer

GL: As Vincent explained, yes, there is a difference. While the company is organized in a more traditional way, with a division of responsibilities, in the XWiki.org there are only committers and contributors, with no other formal title. Everyone is encouraged to participate according to their abilities, be that coding, documenting, testing or communicating. In addition to this, a lot of members of the community are not related to the company whatsoever. You can find out more about them here: `http://dev.xwiki.org/xwiki/bin/view/Community/HallOfFame`

MS: In a related matter, many of the role descriptions of the core developers contain manager and leadership titles.

VM: Hmm. Do you have a link? AFAIK we just have committers.

MS: I'm sorry, I guess my mistake here was not taking the separation between XWiki SAS and XWiki.org into consideration. I was referring to `http://www.xwiki.com/xwiki/bin/view/Company/Team`, where three Project Managers, a Team Leader, a Communication Manager, an Administration Manager, a Support & Documentation Manager, a Research Manager, etc. are mentioned, so I thought these roles might be relevant. The same group of people working on the same projects in two different ways, this is still a bit hard for me to grasp.

VM: It's easy: think about it in the same way in which a person always belongs to several groups:

- group of your close friends
- group of people sharing a hobby with you
- group of people playing a sport with you
- group of people going to Church with you etc.

For all these groups you're the same person but with different hats when you're in one of those groups. And each of these groups has its own rules.The only hard part here is that there's the word "XWiki" in both entities (open source project and company). I personally don't like this since it brings ambiguity when we have a very clear separation between both.

MS: What, in practice, are the main tasks of the people managing the development?

VM: There's nobody managing the development ;) We're auto-managed. Committers make proposals or start votes and others discuss and vote. We do stuff by collegial agreements.

MS: I see your point. So everyone (who is interested and dedicated) takes an equal part in making decisions?

VM: Yes. That's our vote process.

MS: Do you think these collegial agreements would still hold if the base of contributors was significantly larger or wider spread, and therefore incorporated more diverse ideas about the project? Or would the system have to be adapted to scale successfully?

VM: Right now we're about 15 active committers. I have no idea at what level it would break but my gut feeling is that it would hold till about 50 active committers. The reason for this value is that I've worked in the past in company where we all had the same title and worked collegially and we only had to introduce a hierarchy when we reached 50 employees or so. Actually I even think that the open source project could go beyond this value since it's much more free and relaxed than a company. Note that I'm talking about committers. Now there are a lot more people participating in various ways (raising issues, suggesting ideas, supporting others on the list, writing articles/blog posts/tweeting, sending pull requests, etc). Also note that there aren't that many projects with 50 active committers (Actually I can't even cite one!) so we're pretty safe for a long time IMO ;)

MS: Many meritocracies have safety nets, some rules to follow or people to go to when no consensus can be reached on important topics. Did this ever occur in XWiki? What would be done in such a situation?

VM: It happens rarely. We do frequently don't agree on some details. The proposals or vote is then discussed and amended with a new proposal/vote. Example:

- First vote: `http://markmail.org/thread/56v5thsj6wv7tpno`
- Second vote: `http://markmail.org/thread/tino4ngttflc5i3s`

Sometimes it takes longer to reach a consensus but it's always a better result IMO. Nobody in this community votes -1 without a good reason just to bother people ;)

MS: Have either the formal roles or the informal merit people earned in a special field some kind of influence on the weight of their voice in a dispute?

VM: Everyone has the same vote power but then people have more influence than others through their past actions.

GL: I don't have much to add to Vincent's answer here. The important thing to keep in mind is that when taking part to the community, your XWiki SAS title plays no part. For instance in my case, the weight of my remarks would come from my past involvement in the community (writing documentation, taking part to discussion about features, testing the product) rather my XWiki SAS role.

MS: And last but not least some questions about the access and distribution of knowledge: XWiki features an extensive written documentation of itself and the process used in its development. What is the role then of additional, personal communication, of the proverbial informal talk at the water cooler? Is the necessary time and ceremony of written documentation always justified by making the knowledge permanently available to everyone, or can you think of exceptions?

VM: We try to make it as easy as possible for someone external to follow the development or join the team. We also try to have a common style for writing code. This is needed before there's no ownership of the code and everyone can modify any part of the code. Also homogeneous code also makes it easier to join the dev team. Of course we have lots of informal discussion on the mailing list and on IRC (and people speak with other privately too but all proposals/decisions are always made publicly).

GL: When starting to work with a new piece of technology, even with the best documentation in the world there will be times when you will need to ask questions and have a discussion with others. This can happen in real life (and it does a lot at the office), but also through Skype or IRC. XWiki devs and users try to make themselves available to answer questions from users and newcomers, which greatly contributes to the sense of community around XWiki.

[. . .]

MS: As I explained in one of my earlier posts, I am researching open source development processes in comparison to other approaches of developing software, and trying to find out how they relate to each other in different aspects. Besides examining those aspects, however, I am also interested in your opinion about the process. Do you see similarities to agile methods of development, or to the well-defined processes of software engineering, in your approach? Where can these similarities be found, and where do they end?

VM: Well each open source project is completely free to organize itself the way it wants so we cannot say anything about any similarities. Now I personally like agile practices which I've been practicing before I came to know eXtreme Programming and which I've been applying to my open source projects wherever I could. Some examples of practices we're applying here on the xwiki project:

- Release often (every 3 weeks in average)
- Relentless refactoring
- Automated tests (gives courage for refactoring) and Continuous Integration
- We don't do pair programming but we do "continuous" reviews by sending email diffs to the lists so that all committers can see them and review them
- Time boxing
- Collective code ownership
- Coding standard
- Simple design/System metaphor through our proposals/vote practice on the list and through our common decision making

MS: Finally, may I ask you for a quick outlook into the future of XWiki? Which chances and challenges do you see coming up? In what direction would you like the development of XWiki to go in the following years?

VM: ahah.... Good question. Several answers:

- One current challenge is in finishing to split the XWiki code base into small modules that form our platform and that can then be assembled by users to construct the collaborative web site they wish. We've progressed a lot in this direction but there's still some work with our Extension Manager, splitting our code and introducing extensible UIs through what we call Interface Extensions.
- We have an important challenge in being able to attract people. There are 2 dangers here:
  - The XWiki software is becoming better and better and people usually want to participate to an open source project when they see it's not "finished" and they see they can help out. If a project is too well finished people won't participate. See this blog post I've written a long time ago about this: `http://web.archive.org/web/20090130001223/http://blogs.codehaus.org/people/vmassol/archives/001325_how_can_i_improve_my_oss_project_managment_skills.html`

  - The XWiki open source project is full of rules that we've voted over the years to improve the way we develop software. This has a good and bad point for attracting newcomers:
    * bad: It could be seen as daunting to have to learn and bother with all those rules when all you want to do is "just code"
    * good: the newcomer will learn a lot about software development. Moreover nothing is set in stone. Anyone can propose to remove or change a rule at any point in time and provided it's voted positively it'll be changed

  Actually I think that our solution for this is what we've started doing:

- Make everything an extension and allow anyone to contribute extensions irrelevant to how they developed it. This `http://extensions.xwiki.org`. Our challenge is in creating the tools within the XWiki software to make it easy to publish extensions.

MS: Also, do you have any points you would like to additionally mention, some vital aspect of the process I failed to address, or a special emphasis on anything you feel we did not talk about enough? I am glad for further hints and comments. :)

VM: Nothing comes to mind. Maybe just that we're all passionate people :)

APPENDIX D

The Agile Manifesto

Manifesto for Agile Software Development

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.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.

Welcome changing requirements, even late in
development. Agile processes harness change for
the customer's competitive advantage.

Deliver working software frequently, from a
couple of weeks to a couple of months, with a
preference to the shorter timescale.

Business people and developers must work
together daily throughout the project.

Build projects around motivated individuals.
Give them the environment and support they need,
and trust them to get the job done.

The most efficient and effective method of
conveying information to and within a development
team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development.
The sponsors, developers, and users should be able
to maintain a constant pace indefinitely.

Continuous attention to technical excellence
and good design enhances agility.

Simplicity–the art of maximizing the amount
of work not done–is essential.

The best architectures, requirements, and designs
emerge from self-organizing teams.

At regular intervals, the team reflects on how
to become more effective, then tunes and adjusts
its behavior accordingly.

Bibliography

[1] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New Directions on Agile Methods: A Comparative Analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, Washington, DC, 2003. IEEE Computer Society.

[2] Agile Alliance. Manifesto for Agile Software Development. http://www.agilemanifesto.org/, 2001. [Online; accessed 19-November-2011].

[3] Manuela Angioni, Raffaella Sanna, and Alessandro Soro. Defining a Distributed Agile Methodology for an Open Source Scenario. In , editor, *Proceedings of the 1st International Conference on Open Source Systems (OSS 2005) - Genova, Italy*, volume 1, july 2005. idxproject: MAPS NDA.

[4] Friedrich L. Bauer. Software Engineering - Wie es begann. *Informatik Spektrum*, 16(5):259–260, 1993.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, first edition, October 1999.

[6] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, second edition, November 2004.

[7] Barry Boehm. Get Ready for Agile Methods, With Care. *Computer*, 35(1):64–69, January 2002.

[8] Barry Boehm. A View of 20th and 21st Century Software Engineering. *ICSE'06*, 2006.

[9] Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, May 2003.

[10] Terry Bollinger, Russel Nelson, Karsten M. Self, and Stephen J. Turnbull. Open-Source Methods: Peering Through the Clutter. *IEEE Software*, 16(4):8–11, July/August 1999.

[11] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987.

[12] Dries Buytaert. Drupal. http://drupal.org/, April 2012. [Online; accessed 24-April-2012].

[13] Dries Buytaert. DrupalCon Denver 2012 - Keynote. http://denver2012.drupal.org/keynote/dries-buytaert, March 2012. [Online; accessed 24-April-2012].

[14] Lan Cao, Kannan Mohan, Peng Xu, and Balasubramaniam Ramesh. How Extreme Does Extreme Programming Have to Be? Adapting XP Practices to Large-Scale Projects. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 3 - Volume 3*, HICSS '04, pages 30083.3–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, October 2006.

[16] Kevin Crowston and Hala Annabi. Information Systems Success in Free and Open Source Software Development: Theory and Measures. In *Software Process: Improvement and Practice*, pages 123–148, 2006.

[17] Kevin Crowston and Barbara Scozzi. Open Source Software Projects as Virtual Organizations: Competency Rallying for Software Development. *IEE Proceedings - Software Engineering*, 149(1):3–17, 2002.

[18] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/Libre Open Source Software Development: What We Know and What We Do Not Know. *ACM Computing Surveys*, 44, 2010.

[19] Pete Deemer, Gabrielle Benefield, Craig Larman, and Bas Vodde. The Scrum Primer. http://www.scrumalliance.org/resources/339, 2010. [Online; accessed 12-January-2012].

[20] Ludovic Dubost. XWiki - Vision on Open Source. http://xwiki.com/xwiki/bin/view/Blog/XWikiVisionOpenSource, May 2012. [Online; accessed 24-May-2012].

[21] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc., October 2005.

[22] Django Software Foundation. Django. https://www.djangoproject.com/, June 2012. [Online; accessed 17-June-2012].

[23] Free Software Foundation. The Free Software Definition. http://www.gnu.org/philosophy/free-sw.html. [Online; accessed 28-January-2012].

[24] The Eclipse Foundation. Eclipse Development Process. http://www.eclipse.org/projects/dev_process/development_process_2011.php, 2011. [Online; accessed 27-Mar-2012].

[25] Alfonso Fuggetta. Open source software: An evaluation. *J. Syst. Softw.*, 66(1):77–90, April 2003.

[26] Volker Grassmuck. *Freie Software: Zwischen Privat- und Gemeineigentum.* Bonn: Bundeszentrale für Politische Bildung, 2002.

[27] Adrian Holovaty and Jacob Kaplan-Moss. The Django Book. http://www.djangobook.com/en/2.0/, 2009. [Online; accessed 26-May-2012].

[28] Open Source Initiative. Open Source Definition. http://www.opensource.org/docs/osd. [Online; accessed 28-Jan-2012].

[29] Osamu Kobayashi, Mitsuyoshi Kawabata, Makoto Sakai, and Eddy Parkinson. Analysis of the Interaction Between Practices for Introducing XP Effectively. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 544–550, New York, USA, 2006. ACM.

[30] Stefan Koch. Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion. In Jutta Eckstein and Hubert Baumeister, editors, *XP*, volume 3092 of *Lecture Notes in Computer Science*, pages 85–93. Springer, 2004.

[31] Stefan Koch. Evolution of Open Source Software Systems: A Large-Scale Investigation. In *International Conference on Open Source Systems*, pages 148–153, 2005.

[32] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Addison-Wesley, 2003.

[33] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, third edition, 2003.

[34] Craig Larman. *Agile and Iterative Development: A Manager's Guide.* Addison-Wesley, 2004.

[35] Netcraft Ltd. July 2012 Web Server Survey. http://news.netcraft.com/archives/2012/07/03/july-2012-web-server-survey.html, July 2012. [Online; accessed 04-Aug-2012].

[36] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Manage. Sci.*, 52(7):1015–1030, July 2006.

[37] Bart Massey. Why OSS Folks Think SE Folks Are Clue-Impaired. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, International Conference on Software Engineering. 2003*, pages 91–97. ICSE, 2003.

[38] Steve McConnell. Open-Source Methodology: Ready for Prime Time? *IEEE software*, 1999.

[39] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[40] Siobhan O'Mahony and Fabrizio Ferraro. The Emergence of Governance in an Open Source Community. *Academy of Management Journal*, 50(5):1079–1106, 2007.

[41] Margit Osterloh, Sandra Rota, and Bernhard Kuster. Open-Source-Softwareproduktion: Ein neues Innovationsmodell. *Open Source Jahrbuch*, 2004.

[42] Project Management Institute PMI. *A Guide to the Project Management Body of Knowledge*. Project Management Institute, Newtown Square, PA, fourth edition, 2008.

[43] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly Media, revised edition, January 2001.

[44] Christian R. Reis and Renata P. M. Fortes. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In *Workshop on Open Source Software Development*, pages 155–175, Newcastle, UK, 2002.

[45] Hugh Robinson and Helen Sharp. XP Culture: Why the Twelve Practices Both are and are not the Most Significant Thing. In *ADC '03: Proceedings of the Conference on Agile Development*, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Winston W. Royce. Managing the Development of Large Software Systems. *Proceedings of WESCON*, 1970.

[47] Herbert Rubin and Irene Rubin. *Qualitative Interviewing: The Art of Hearing Data*. Sage Publications, 2011.

[48] Walt Scacchi. Understanding Requirements for Developing Open Source Software Systems. In *IEE Proceedings - Software*, pages 24–39, 2001.

[49] Walt Scacchi. Is Open Source Software Development Faster, Better, and Cheaper than Software Engineering? In *2nd ICSE Workshop on Open Source Software Engineering*, 2002.

[50] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. Understanding Free/Open Source Software Development Processes. 11(2):95 –105, March/April 2006.

[51] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, first edition, 2002.

[52] Sonali K. Shah. Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development. *Manage. Sci.*, 52(7):1000–1014, July 2006.

[53] Sonali K. Shah and Kevin G. Corley. Building Better Theory by Bridging the Quantitative-Qualitative Divide. *Journal of Management Studies*, 48(8):1821–1835, 2006.

[54] Srinarayan Sharma, Vijayan Sugumaran, and Balaji Rajagopalan. A Framework for Creating Hybrid-Open Source Software Communities. *Info Systems*, 12:7–25, 2002.

[55] Black Duck Software. Ohloh. www.ohloh.net, 2012. [Online; accessed 17-May-2012].

[56] Richard Stallman. Viewpoint: Why ‘Open Source’ Misses the Point of Free Software. *Commun. ACM*, 52(6):31–33, June 2009.

[57] Klaas-Jan Stol, Muhammad Ali Babar, Paris Avgeriou, and Brian Fitzgerald. A Comparative Study of Challenges in Integrating Open Source Software and Inner Source Software. *Inf. Softw. Technol.*, 53(12):1319–1336, December 2011.

[58] Jeff Sutherland. Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies. *Cutter IT J*, 14(12):5–11, 2001.

[59] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, pages 137–146, January 1986.

[60] Paul Vixie. Software Engineering. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O’Reilly and Associates, Cambridge, Massachusetts, 1999.

[61] Georg von Krogh and Eric von Hippel. The Promise of Research on Open Source Software. *Management Science*, 52:975–983, 2006.

[62] Juhani Warsta and Pekka Abrahamsson. Is Open Source Software Development Essentially an Agile Method? In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, pages 143–147, Portland, Oregon, 2003.

[63] Steve Weber. *The Success of Open Source*. Harvard University Press, 2005.

[64] Dave West. Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today. http://www.cohaa.org/content/sites/default/files/water-scrum-fall_0.pdf, July 2011. [Online; accessed 28-Jul-2012].

[65] XWiki. The devs Archives. http://lists.xwiki.org/pipermail/devs/, June 2012. [Online; accessed 11-Jun-2012].

[66] XWiki. XWiki. http://www.xwiki.org/xwiki/bin/view/Main/WebHome, June 2012. [Online; accessed 24-Jun-2012].

[67] yeebase media GmbH. t3n. http://t3n.de/opensource/projects/, April 2012. [Online; accessed 02-Apr-2012].

[68] Jamie Zawinski. Resignation and Postmortem. http://www.jwz.org/gruntle/nomo.html, March 1999. [Online; accessed 27-July-2012].