János Hegymegi

BUGS AS FEATURE: CODIFYING TACIT KNOWLEDGE IN PROJECT MANAGEMENT

Degree Programme in Innovative Business Services

2014

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

BUGS AS FEATURE: CODIFYING TACIT KNOWLEDGE IN PROJECT MANAGEMENT

Hegymegi, Janos
Satakunta University of Applied Sciences
Degree Programme in Innovative Business Services
2014
Supervisor: Koivula, Reijo
Number of pages: 44

Keywords: software development, project management, knowledge management

---

Abstract. Web development has become increasingly complex in the past decade. Delivering features faster and more reliably than the competition is one of the main aspects of success but with speed comes a higher chance of failure. These tendencies are the main reason shorter and safer development cycles become increasingly important. This, in turn, also leads to a shift in what an issue with the program (called a `bug`) means, how it is processed, and what fixing it means.

The aim of the thesis is to examine the role of knowledge transfer in software development, specifically in the domain of defect processing. An overview of bug management and software development history is followed by an introduction to knowledge transfer studies. Research on knowledge transfer is applied to a bug database, examining how tacit knowledge is codified during the process. Topics and suggestions for further research are outlined.

CONTENTS

1 INTRODUCTION

There's no denying that bugs were born with - and are just as old as - software. It has been part of the IT industry and software development for better or worse. Engineering and management methodologies have treated it differently but both looked at it as an unwelcome but unavoidable aftereffect. Little has been investigated how bugs contribute to software development as a medium of tacit knowledge.

A computer system has two distinctive parts. Hardware is an assembled physical good that consists of standardised items in a standardised architecture. Choices and problems are binary and solutions come from mathematical and mechanical practices, which are documented and codified. It is the reason why learning and knowing about hardware what science says explicit knowledge.

On the other hand, softwares are iterative by nature and software engineering has thus always been a highly volatile field. Fixing one issue may lead to two other ones, and every new feature can possibly introduce tens or hundreds of issues. Based on these observations it is safe to assume that software is never totally bug free – that is to say, one can always find information from the implicit feedback that comes with an issue. Examining and understanding such bugs brings valuable tacit knowledge.

1.1 Problem definition

Traditionally, bugs have been processed in a vacuum, or as part of quality-assurance, far from feature and product development. These teams could outnumber engineers by an 8:1 ration in some cases (Vestbø, 2007) but as open source gained market share and trends shifted in web development there has been a push to make development cycles shorter and faster.

Many steps of the software development process were unified or altogether defied, in order to achieve that. This not only means faster delivery of the product to its end-user but also a higher chance of delivering issues with it. Understanding how bugs are helping software development is one main point of this thesis.

Reviewed literature in the field has investigated bugs from an engineering, for example: the time a bug fix amounts to (Kim and Whitehead Jr, 2006), information systems, f.e.: distribution of bugs in softwares (Murgia, 2011), and quality assurance perspective, f.e.: methods for automating bug prognostication (Shivaji, 2013). However, the exchange of information during bug processes have been scarcely investigated.

Bugs can signal issues that provide knowledge not just on the nature of the problem but also on the nature of the product. A knowledge management perspective is outlined to be applied when analysing the findings of this paper. The research hopes to answer questions whether changing software development model leads to better results in the bug fixing process.

1.2 Thesis' scope

The many facets of software bugs have been investigated and hypothesised by literature in the fields of Business Management, Quality Assurance, Information Systems, Computer Science and Software Engineering. This thesis has limited review to works from Information Systems, Software Engineering and Management. Other fields can be referenced but I have tried to keep the focus on the aspects of knowledge transfer and communication through bugs and reports.

Scope and limitations are outlined at the beginning where applicable, while a selection of key points which have been discussed are going to be highlighted at the end of each chapter and subchapter.

2 OVERVIEW OF SOFTWARE DEVELOPMENT HISTORY

This chapter is aiming to provide a background to this thesis. It first gives historical context to where alternative software development processes stem from, introducing

traditional software houses which developed closed source softwares and communities embracing open source development. Then it describes old and new software development processes, with a focus on the role of bugs.

2.1 Background

In Computer Sciences and Software Engineering bugs are interchangeable with software defect. In the following, however I refer to bugs as an object. That does not necessarily represents a defect. It can signal enhancement, it can describe tasks, and is part of a complex system of network that produces software to the end-user.

2.2 Software defects

Although the term ”bug” itself has been coined long before, it was after a moth in the Mark II relay calculator was caught and then attached to the computer’s log book, that the expression (and the verb ”debugging” along with it) was popularised and added to the software lexicon (Shapiro, 1987).

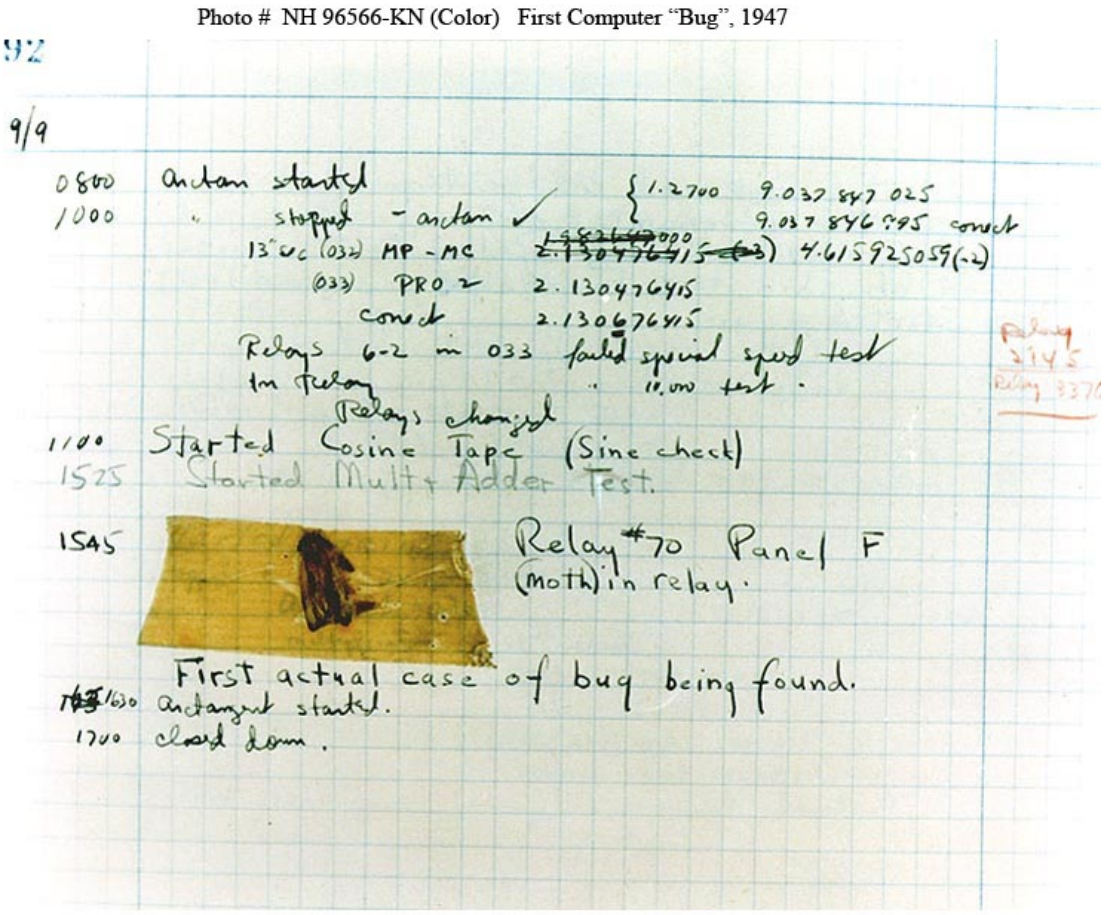

Fig. 1: The first actual bug caught (Naval History and Heritage Command, 1999)

There's no anecdote or evidence of any other actual bug penetrating computer systems. It was not just because of this, however, that computers were easy to keep bug free. Before the first operating system was introduced there was no independent field of software development. It was the Garmisch conference in 1968 that based much of what we know today as software engineering (Philipson, 2004).

2.3 Proprietary software

Separating software from hardware meant that an entirely new field was coming to age. In the beginning software development, quality assurance, testing were all concepts in their infancy and mainly rooted in engineering practices of hardwares and the physical realm which necessitated laying down a series of axioms for software practices (Brooks, 1995).

The most prominent methodology of this era is called the Waterfall model because each step follows the next as a cascading waterfall (Mohammed, Munassar and Govardhan, 2010).

This methodology was hoped to help create more complete systems and better performing project management throughout different businesses. It was thought to be better than ad-hoc development but most research concluded this method to be a huge bureaucratic burden that wasn't applicable but to the largest organisations and hindered completing projects even on such a huge scale (Middleton, 1994).

The companies that used these methods sold only a license to software usage but not the source code (the master code the program is running on) itself, prompting the term 'proprietary software'. Users of such softwares had no right to fix bugs that prevented them from using softwares or to improve upon concepts. Methodologies used by these companies hindered fast delivery. Softwares with slow bug-fix release cycles and the ethical issues presented were the driving factors of the open source movement (Vestbø, 2007).

2.3.1 Open source software

In the middle of the 1980s open sourced softwares started to gain traction. The name open source refers to the nature of the source code of the software. A code that has been made available to the public is called open source and is readable, usable, hackable by everyone interested.

This was a major paradigm shift compared to how the traditional software houses have been built. Most for-profit corporations operated, and still operate to date, with closed source softwares. In practice this means a user is allowed to use the software as it was made available to by the publishing company. The code the program is running is closed from the public.

Given the difference in ethos and nature open software projects had a very different hierarchy and a very different set of processes. Code was available online for anyone

wishing to participate. This bigger group of contributors was co-ordinated by a core group of the project's developers. The fact that contribution was voluntary lead to two important developments.

First of all, it meant, that only the most enthusiastic and knowledgeable developers worked on open source software development. This high ceiling also brought in a lot of specialists who found challenges that peaked their interest in such projects.

Secondly, it meant that contributors all chose to work on issues and features that they could work with. Many of the developers being specialists they touched and worked with parts of the software they understood and could contribute to.

Transparency was a key to the success of open source softwares and the need for it became the driving factor for developing tools and organisational hierarchy that helped to achieve this.

Proponents and advisors of open source were the likes of Linus Torvalds and Eric Steven Raymond. The latter wrote the then-groundbreaking *The Cathedral and the Bazaar*. The book discusses the reasons open source projects can continuously report a higher level of code and product quality than traditional software companies and processes.

The success of open source as a business model and organisational principle affected more traditional companies as well as inspired a new generation of developers and engineers.

2.4 Comparison of Software Engineering Models

The software development life cycles are represented with abstract representations known as software development process models. A various amount of perspectives are defined to describe such a process however most commonly the development process models share the following four (Mohammed, Munassar and Govardhan, 2010):

1. Specification
2. Design
3. Validation
4. Evolution

In the following, we will describe four of the most common software development processes: Waterfall, Agile, Scrum, and Lean. One was chosen to represent classical software development theories and three to display different concurrent approaches that have evolved in the recent decades. All four models describe a whole software development life cycle and have features comparable to one another.

We examine the literature of these models to map advantages and disadvantages of each, and describe how these processes enable or reduce the organisation's ability to transfer knowledge at a high rate.

2.4.1 Waterfall

Formally first described in 1970 for each company that needed to develop large software systems (Royce, 1970) it has had many modified versions that are all labeled as a Waterfall model. This model is the standard of all software engineering models and is still in use by European governments, agencies and many big corporations (Middleton, 1994).

The model was intended to make projects more thorough by heavily documenting every step of the early planning stages. The steps are following each other in a sequence and only after having a finalised documentation of a stage could the next one be started. More complete systems could be achieved cheaper this way in theory however oversights actually made such an approach more expensive as each of the previous stages had to be revisited if an issue would go unnoticed (Mohammed, Munassar and Govardhan, 2010). The steps of the traditional Waterfall model are summarised as such (Brooks, 1995):

1. Plan
2. Code
3. Component Test
4. System Test
5. Field Support

Stages are named wildly differently in modified Waterfall models but they all represent the same non-iterative steps. An example of this is a system such as the SSADM. The steps are the following (Middleton, 1994):

1. Strategic Planning
2. Feasibility Study
3. Requirements Analysis
4. Requirements Specification SSADM
5. Logical System Specification
6. Physical Design
7. Construct and Test

Research shows that no actual project could do withouth modifying or tailoring these steps in the actual development cycle (Middleton, 1994).

The Waterfall software development process was designed for by big corporations and governments projects to tackle the disadvantages of ad-hoc engineering (Royce, 1970), thus most advantages are only present on large scale. It has a well known theory, it presents clear deliverables for the management, and is document driven (Mohammed, Munassar and Govardhan, 2010) yet it is mostly known for it's disadvantages as described in (Brooks, 1995), (Mohammed, Munassar and Govardhan, 2010), (Petersen, Wohlin and Baca, 2009). It is too costly to revisit the linear steps for a possibly new iteration because the process implies that the cycle will be only completed once and that it can be built issue and bug free. It is also document-heavy and requires a high cost and effort before actual development can take place. Additionally it is hard to properly define customers needs upfront which can result in products developed through a whole cycle that were completely unneeded.

Although the Waterfall model is used to demonstrate all the conceivable illnesses of sequential, conservative frameworks we only would like to focus on how the model performs when it comes to defects and bugs.

In any version of Waterfall the strict order of steps and the idea behind it, that systems have to be implemented fully, push bug fixing to the testing and maintenance phases. The system is tested against the requirements set by the customer and specifications set during the planning stage. As shown in many case studies such as (Petersen, Wohlin and Baca, 2009) this is suboptimal for numerous reasons. Bugs found during Quality Assurance are treated in the vacuum of Testing & QA, testing wholly integrated systems is problematic as the number of possible issues grows exponentially with each component introduced, bugs and issues are too numerous, and it is hard to act upon them as both design and implementation steps have to be revisited making it a costly process.

The possible knowlege transfer bugs provide in a Waterfall model is quite low but, just as the model in general - due to its age and wide acceptance - it is a good candidate to be the basis of comparison for all other software development methodologies when evaluating knowledge transfer.

2.4.2 Agile

Released in 2001 as a Manifesto, the Agile Software Development Process, or Agile in short, is an umbrella term for most contemporary frameworks in software development (Agilemanifesto.org, 2001)[1].

Some of the best known methodologies are summarised in (Lindvall et al., 2002):

- Extreme Programming (XP)
- Scrum

[1] http://agilemanifesto.org/

- Feature Driven Development (FDD)
- Dynamic Systems Development Method (DSDM)
- Crystal
- Agile modeling

The list is a bit dated but it gives a nice overview of the versatility of Agile. The manifesto collected and described lessons that have been learnt in methodologies that pre-date the manifesto and methodologies that were born or gained traction afterwards were able to use and incorporate these lessons.

Agile is not the sum of its parts, which is why it can't be examined only through methodologies it contains but has to be analysed on its own as well. Agile treats certain ideas differently then the methodologies it borrows from, for example: in Scrum continuously delivering "working software" is the result of executing successful sprints (Sutherland and Schwaber, 2007), whereas in the Agile Manifesto it is a guiding principle in itself (Agilemanifesto.org, 2001).

This can be attributed to the fact that Agile acts as a guideline, and models not exact steps of software development but a set of values that is mutually agreed upon by the advocates of these methodologies.

These values and principles are an attempt to answer problems older models introduced or failed to address in the first place, parallel with Free, Libre, Open Source softwares (FLOSS) gaining mainstream acceptance. The main differences between these models are displayed in Table 1 below.

	Waterfall Model	**Agile Model**
Project Size	Big	Small
Life cycle timeframe	Long-term	Short-term
Life cycle design	Sequential	Iterative
Flexibility	Low	High

Table 1: Short comparison of properties of Waterfall and Agile methodologies

The values and properties are not equally important and popular as shown in researches in the field (Begel and Nagappan, 2007). Key values and benefits of Agile are coding standards, continuous integration, and user stories. These are all attributes important to maintain an organisation with a high rate of knowledge transfer.

Although many of these models implement and describe interesting ideas due to time constraints and available literature two out of the many Agile Software Development methods have been investigated: the Scrum and the Lean methodologies. Scrum is the most popular and one of the most widely recognised contemporary methods, while Lean is one of the most recent models to gain traction.

2.4.3 Scrum

Scrum is an agile methodology first described in 1996. The most popular of the many agile methods (Begel and Nagappan, 2007), it treated software development as a volatile, highly unstable process opposed to the accepted philosophy where a software development life cycle was plannable and could be executed successfully in a calculated manner. Rather than trying to form arbitrary steps and force software life cycle to respect them it accepts the unpredictable nature and adopts a ”do what it takes” approach (Sutherland and Schwaber, 2007).

Development life cycle in Scrum projects is broken into three stages. There is a pre-sprint planning period, followed by the sprint, which is when actual development takes place and after the sprint is done there is a post-sprint evaluation (Sutherland and Schwaber, 2007).

Any project that adopts the Scrum methodology can and is empowered to use tools and development techniques that fit the development best and Scrum only prescribes a few methods to align these activities. The key idea is the sprint which has a fixed duration and encapsulates most of the development due to the rapid nature of Scrum. When planning is done the tasks and To-Dos the team has committed to are added to a so called ”sprint backlog” and have to be completed during the sprint. These are prioritised and progress of the team is maintained on a burn-down chart. The team of

developers who work on a sprint is a Scrum team. There is rarely any hierarchy in these teams. Developers choose tasks and only meet for short daily meetings to share knowledge, keep common goals aligned and check upon progress. Team members have to answer what they have accomplished since the last meeting, what they plan to accomplish until the next and if they faced any issues during working on their tasks (Bhavya et al., 2012).

The flexibility, rapidity, and freedom Scrum methodologies provide are all reasons it is one of the most widely adopted model among smaller companies and projects. Teams in Scrum are small, 4 to 6 developers who work in flat hierarchy. There has been no formal description on Scrum for distributed teams but the low bureaucratic overhead and short, object-oriented sprints make it a good candidate for such teams as well, and this topic currently serves as a basis for ongoing research (Bhavya et al., 2012).

Bugs and issues in Scrum projects have a very direct relationship with development. Contrary to Waterfall models, to realise the short development cycles, bugs are communicated and acted upon directly within sprints. Issues aren't processed separately but are part of the "product backlog" (Sutherland and Schwaber, 2007). With only a single list ranked by priority the bugs have to provide as much value to the external owner and the business product as any feature. Comparing the amount of knowledge that transfers between user and developer in bugs and features is an interesting topic for future research.

2.4.4 Lean Software Development

Known in manufacturing as Lean Production, the concept has been adopted in the recent years for Software Development. This application of Lean to this specific field is also called Lean Software Development (LSD) and is part of the group collectively known as Agile.

Lean Production was devised by a Toyota engineer in the 1950-60's. The expression was coined first by John Kraficik, a member of an MIT team investigating the auto-

motive industry (Womack, Jones and Roos, 1990). It successfully been applied to various industries such as food manufacturing (Lehtinen and Torkko, 2005), banking services (Wang and Chen, 2010) and health care (Murrell, Offerman and Kauffman, 2011). Despite it being one of the most successful frameworks for factory level manufacturing it wasn't formally adapted for use in software development until 2003 (Poppendieck and Poppendieck, 2003).

The lean development model has seven main principles and a total of 22 tools for use but the degree and amount of adoption is left up to the companies and software developers to decide.

The seven main principles that were originally defined by Taiichi Ohno, the father or Lean Manufacturing Processes are (Poppendieck and Poppendieck, 2003):

1. Eliminate waste
2. Amplify learning
2. Decide as late as possible
2. Deliver as fast as possible
2. Empower the team
2. Build integrity in
2. See the whole

The numbered list above is intended to display and to some extent visualise that the central tenet of Lean is eliminating waste. This is the fundamental principle from which every other step of Lean Process was born.

When developing the Toyota Production System, the iconic and first Lean Process, in order to be able to eliminate them seven types of manufacturing waste were identified. Mirroring it, exactly seven types of waste in software development were identified by (Poppendieck and Poppendieck, 2003) as well. These seven include partially done work, task switching, waiting, and defects.

Lean Software Development, as other Agile methods, is a proponent of continuous integration, immediate testing, releasing software fast, and doing so in development

cycles as short as possible. In line with these basic principles, bugs and their impact is measured by the time elapsed between they were introduced and have been fixed, not purely on their criticality (Poppendieck and Poppendieck, 2003).

In practice this means that in Lean Software Development too, bugs and defects are handled as part of development which make it an ideal candidate to examine how knowledge transfer affects software development.

2.5 Overview

This chapter aimed to display the four common software development methodologies, assess their advantages and disadvantages, examine selected literature and case studies to form a basic understanding what the key properties are needed for effective knowledge transfer.

3 MANAGEMENT OF KNOWLEDGE

The breakthrough research on organisational knowledge creation (Nonaka, 1994) marked the beginning of renewed interest in knowledge transfer and management of organisational and individual knowledge creation. This chapter introduces the basics of the field of knowledge management. Afterwards, different types of knowledge are introduced, and we will look how these concepts relate to software development methodologies from the previous chapter.

3.1 The Concept of Knowledge Management

The past three decades saw a huge growth in the number of works that examine knowledge. Knowledge became the most important asset of organisations and the source of competitive advantage (Osterloh and Frey, 2000). The way organisations and individuals create knowledge, how knowledge transfers between parties, what

knowledge is codified and how is that codified knowledge interpreted are all part of the field of Knowledge Management (Vestbø, 2007).

The meaning and concept of ’knowledge’ is hard to pin down and has a different meaning in different contexts. This is part of the reason why knowledge is hard to effectively process and transfer (Vestbø, 2007), why it depends on the dynamics, size and forms of an organisations or motivation of an individual (Osterloh and Frey, 2000), and why it is an important aspect that needs to be taken into consideration when investigating the process of bug fixing in different software development methodologies.

The classic philosophical field examining knowledge is called epistemology. The two opposite approaches defined there are positivism and phenomenology. Positivism views the world obejctively, as a collection of measurable and codifiable truths, and describes that what exists – independently of any interpretation. Phenomenology identifies no independent scientific truth, and views the world subjectively. It only is interested in the interaction and experience from one’s own perspective. The subjective interpretation becomes the single point of truth (Nonaka and Peltokorpi, 2006).

This opposition is also present in the field of knowledge management. Two eras can be identified, the first generation of knowledge management researchers worked from a positivistic point of view, the second viewed phenomenology as their basis of inquiries (Vestbø, 2007). Knowledge management was first interested in building databases of organisational knowledge in a rational, computable way. It could only give limited answers to how knowledge is created on an individual, group, and organisational level. It could also not explain why shared know-how isn’t growing in a predictable manner (Nonaka and Peltokorpi, 2006). Practice-based, phenomenological investigations have focused on of knowledge creation, arguing for a holistic approach where knowledge is a social construct (Nonaka and Peltokorpi, 2006). However, subjective understanding of the knowledge creation process lacks scientific precision and is hard to generalise, therefore a unified model can only be presented if models are considered to complement each other (Nonaka and Peltokorpi, 2006).

The framework for understanding organisational knowledge in this context makes distinctions between ’tacit’ and ’explicit’ knowledge dimensions (Nonaka, 1994), and another set of dimensions differentiating between ’individual’ and ’social’ knowledge have also been defined (Vestbø, 2007). With the help of these patterns, four paths to new knowledge creation can be formalised as displayed in Table 2 below.

	Individual	**Social**
Explicit	Conscious	Objectified
Tacit	Automatic	Collective

Table 2: Patterns to new knowledge creation (Vestbø, 2007).

3.1.1 Tacit knowledge

Tacit knowledge is the type of knowledge that is hard to define and describe. It is very personal and includes opinions, traits, technical skills, knowledge applicable under certain conditions (Nonaka, 1994). These are interpretations of data and information which lead way to new knowledge being formed (Nonaka, 1994).

Being hard to formalize, and subjective to context, tacit knowledge can only be hardly copied (Osterloh and Frey, 2000). Therefore, any organisation that wants to stay competitive needs to facilitate exchange of tacit knowledge through learning and socializing.

3.1.2 Explicit knowledge

Explicit knowledge refers to statistics, numbers, forms, diagrams – any type of knowledge that is described and transferred in formal systems (Nonaka, 1994). This is the type of knowledge that is collected in organisations as well as the ones that present in libraries, databases, schools. Their exchange between individuals is direct and easy to process (Vestbø, 2007).

3.1.3 Further refinements

The basic divide made by Nonaka (Nonaka, 1994) can be further refined when adding the dimensions of individual and social level. There is an individual level of both tacit and explicit knowledge (which are either automatic or conscious), and also a social level to tacit and explicit knowledge (which are either objectified or collective). Automatic, Conscious, Objectified, and Collective are the four types of knowledge state (Vestbø, 2007), all a combination of the four types of knowledge transfer.

3.2 Four Types of Knowledge Transfer

Knowledge transfer can happen between explicit and tacit knowledge and results in knowledge being created. There are four possible scenarios for these transactions (Nonaka, 1994): conversion from tacit to tacit knowledge (Socialization); conversion from tacit to explicit knowledge (Externalization); conversion from explicit to explicit knowledge (Combination); conversion from explicit to tacit knowledge (Internalization).

- Socialization is new tacit knowledge out of tacit knowledge, a way to convert social interaction for example as pattern imitation. The shared experiences can't be understood without their context, therefore new knowledge is tacit as well.
- Externalization is the expansion of tacit knowledge into external knowledge. Through speech and display of models and abstracts, tacit knowledge can be codified, therefore it can be interpreted without fear of missing information.
- Combination is self-explanatory and the most common when examining organisational learning and culture: Different explicit knowledges are gathered, ordered, filtered, and combined to create new value, which is rooted in the shared understanding of the community and needs no tacit context.
- Internalization is the name of absorbing, and thus turning, explicit knowledge into tacit knowledge. It is most similar to what we understand to be 'learning'.

3.2.1 Spiral of Knowledge

Past research on organization knowledge creation handled tacit and explicit knowledge separately, and have failed to take the concept of externalization (3.1.4) into consideration (Nonaka, 1994). The Spiral model for knowledge creation, however, emphasizes the dialogue between tacit and explicit knowledge and argues that failure to allow this dialogue causes problems (Nonaka, 1994).

When all four types of knowledge transfer (3.1.4) are present in an organization, new knowledge is continuously created through a cycle they form. This is called the spiral of knowledge creation. Various interactions on different levels and different contexts can start a cycle. The cycle is, then, shaped by events on the previously mentioned levels, shifting between all modes of knowledge transfer (Nonaka, 1994).

Five conditions have been identified to successfully facilitate organisational knowledge creation. Two of them are relevant for bug- and project-management: Redundancy and autonomy (Vestbø, 2007).

- Redundancy examines the overlap or repeat of information. The logical approach would be to eliminate any kind of redundant information to increase efficiency. However, redundancy (redundant information) carries a big portion of tacit knowledge that helps to shape a common understanding and thus, while eliminating it results in less and more formal information that is easier to process, it also sets new knowledge creation back.
- Autonomy deals with the freedom of an individual in an organisation. Individuals empowered to work in their own pace, make their own decision, create more new knowledge and are part of the spiral of knowledge creation more often.

Successful management of tacit knowledge and a dynamic spiral of knowledge creation has proven to benefit organisational output, identifying and executing short- and long-term goals (Foos, Schum and Rothenberg, 2006). Furthermore, understanding and facilitating the spiral of knowledge has shown to reduce internal costs (Lee, 2000).

4 CASE STUDY

Bugs in the field of software development contain un-codifiable, tacit knowledge that helped contemporary, distributed software development methods to flourish and succeed. Software defects have been the subject of research in various fields from Computer Sciences, through Information Technology, to Software Engineering. Extensive body of literature exists on a wide range investigating bugs from a programming (the automation of bug defect prediction, modelling the number and distribution of bugs in codebases, the health and quality of softwares) and organisational (estimated time of fixing and iterating on a bug, differences between closed- and open source software practices, the risk of defects during new product development) perspective. However these mostly focus on properties of bugs in a vacuum, dynamics of external and internal communication about bugs and the knowledge transfer has been largely ignored.

The database that is tracking bugs in Mozilla's Firefox browser's HTML5 development project will serve as the basic for analysis. Data presented aims to define key points that affect organisational knowledge creation through the bug fixing process. Study will conclude with a summary of the findings and suggestions for further work.

4.1 HTML5 Project

HTML5 is the 5th generation release of HTML, the main language of the internet. It has gained acceptance in the past years with it's specification finalised in October, 2014 (Bright, 2014). All internet browsers are working on adding and modifying features to accomodate HTML5.

Mozilla's Firefox browser was, while being open source, developed with a longer, classical software development cycle until the April of 2011. With the introduction of shorter rapid releases Firefox aimed to leverage the benefits of Agile methodologies they incorporated into their hybrid development system. Figure 2 displays how the project has progressed over the years. Since adopting a more rapid development cycle the HTML5 score has grown significantly and steadily. The latter is especially

important on the competitive market. Figure 2 also displays how Mozilla's Firefox browser performs against Microsoft's Internet Explorer, where the Waterfall model is still dominant in development methodologies (Begel and Nagappan, 2007).

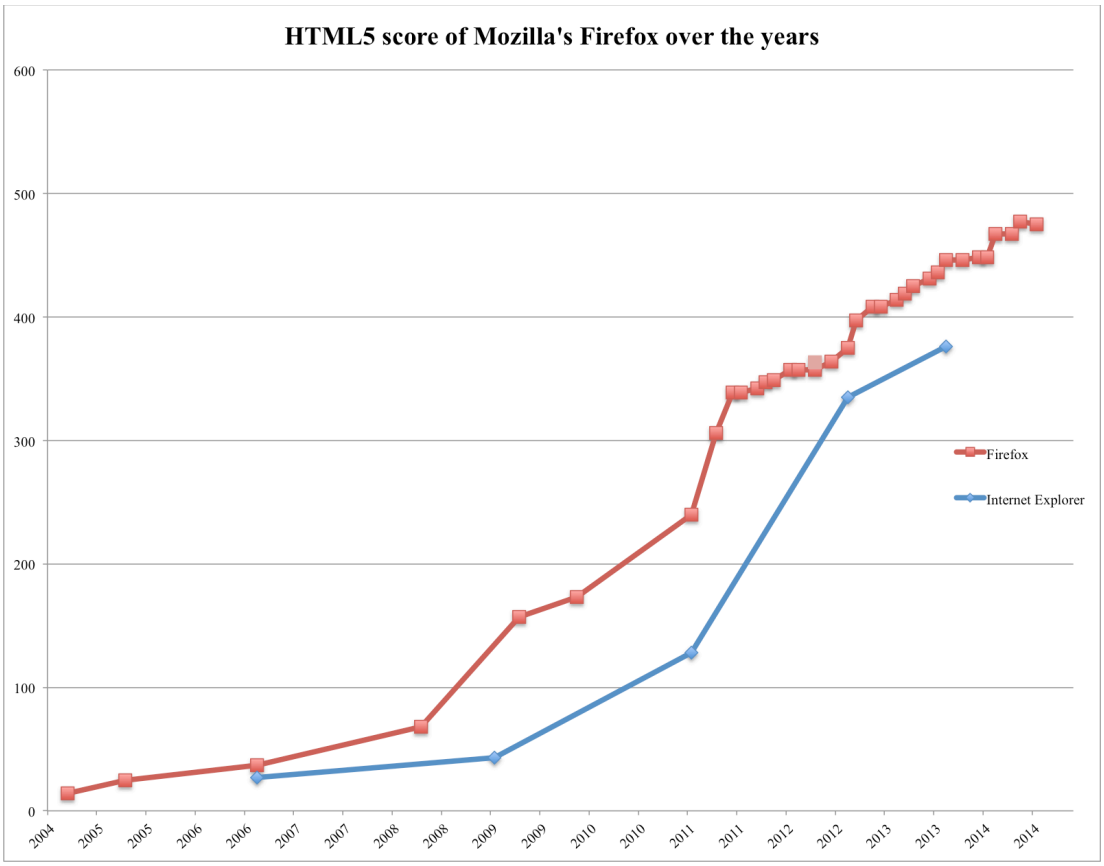

Figure 2: Firefox's and Internet Explorer's HTML5 score over the years (HTML5test.com, 2014)

4.2 Methodology

Examining the development of HTML5 for Mozilla's Firefox browser was a fit for various reasons:

- Mozilla is open source, thus data is easily obtainable.
- Direct comparisons can be made between the old and new development methods.

- It is a development that has implications for the whole stack as shown in Figure 3, data examined is thus easier to generalise.
- A medium-term iteration of an already existing feature can display specific qualities. New feature and bug-fixe implementation are both present in the dataset. Furthermore, this makes historical analysis is possible. The yearly overview of tickets created is shown in Figure 4 below.

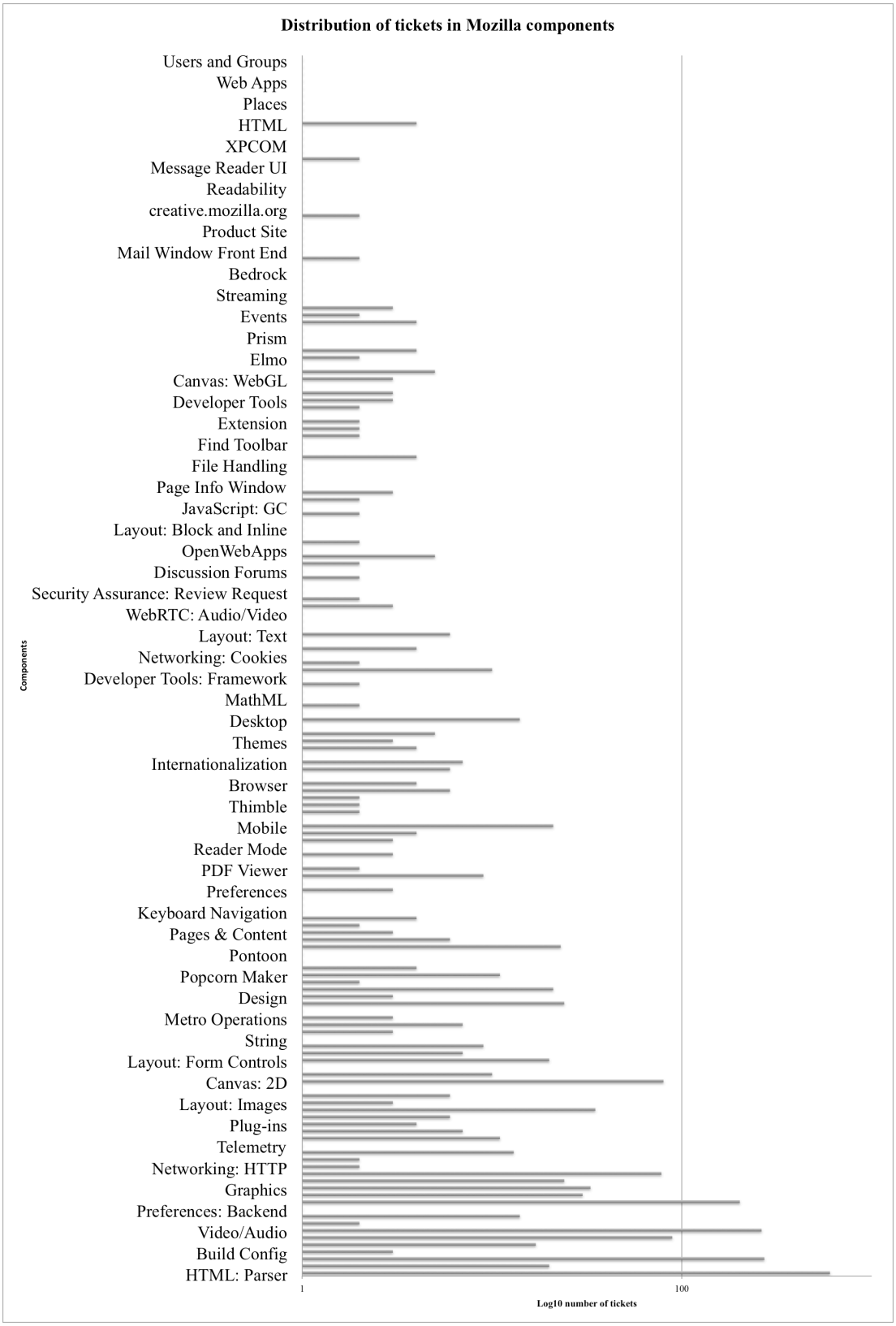

Figure 3: Log10 distribution of tickets between components with at least 1 ticket

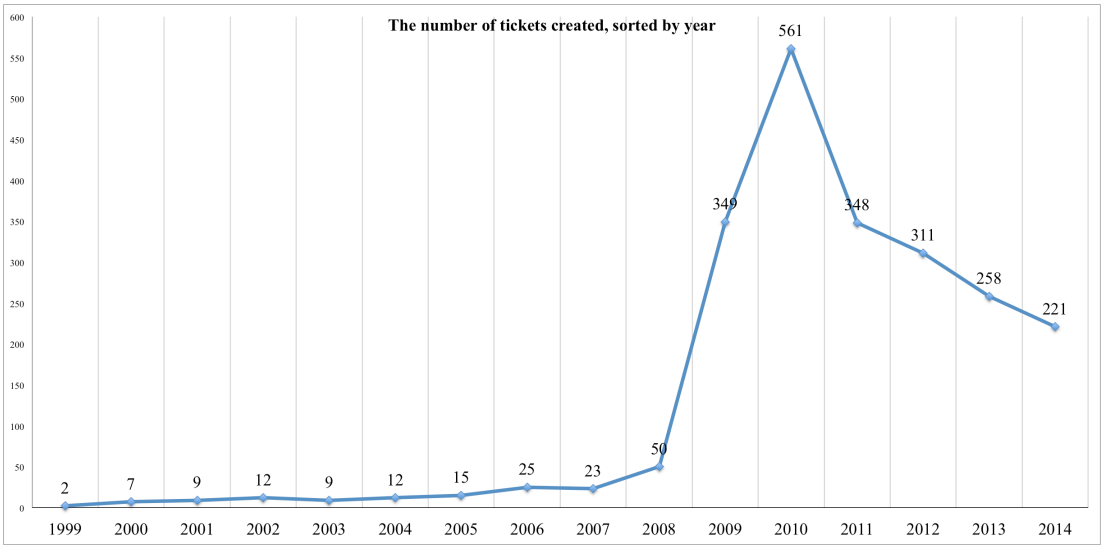

Figure 4: Tickets created, sorted by year

Hypothesises applied to the dataset look to answer questions about specific aspects of knowledge management. Open source projects employ a dedicated, core developer team, that drives product vision and makes the most important decisions. Examining whether this translates into a bias towards more severe issues was one of my concerns. The database has a field for entering custom tags, `flags`. Whether the field allows to codify certain ideas into tags is analysed. Bugs exist in a complex interdependent network, therefore a comparison of dependencies in different software development models were taken into consideration. Finally, an inquiry was conducted to whether voting has a place in bug management and, in a wider sense, organisational knowledge creation.

4.3 Data

Development and bug management of Firefox is transparent and data is available online on Bugzilla, Mozilla's bug tracker and management tool. The database contains millions of rows of bugs and provides a high number of fields to categorise and process reports correctly.

A custom query was used to search the database. The query was searching for every bug which has been added to the Firefox product, that is related to the HTML5 deve-

lopment. Duplicates were removed and not every column used in the query was used after the cleanup.

Due to restraints of time and scope any bugs created later than the 30th of October were excluded from the dataset. Furthermore, out of the investigated set of bugs, 0,74% had no data associated whatsoever. These have been excluded as well.

Table 3 provides a list of the selected columns that were used for analysis:

Column name	Definition
Bug ID	A single unique number given to each report
Classification	The highest order of categorisation
Product	The software or service that is developed, it can be divided into one or more `Components`
Component	A module of a product, the lowest level of categorisation
Assignee	User, who a bug is assigned to
Status	The state a bug is in
Resolution	If the state of a bug is marked as `Resolved`, it defines how
Summary	A short description of an issue
Changed	The most recent date a change has been made on a ticket
Blocks	List of tickets that depend on a ticket
CC Count	Counts people, who aren't directly involved but have subscribed to a ticket
Depends on	List of tickets blocking a ticket
Due Date	Date when a bug should have a resolution by
Duplicate Count	A count of tickets that have been marked as `Duplicate` of one
Flags	A custom status, that can be added to mark a certain state
Last Resolved	The most recent date an issue was marked as `Resolved`
Number of Comments	Count of user comments on a ticket
Opened	The date a ticket was opened
QA Contact	Person responsible for the quality assurance
QA Whiteboard	A custom field to add tags for the quality assurance contact
Reporter	The person who entered the bug into the database

Severity	The value indicates the impact of a bug
Target Milestone	The release the fix is expected to be shipped with. Set by `Assignee`
Votes	Number of votes an issue has been given
Whiteboard	A custom field to add tags
Keywords	A list specific terms, that can be added for easier grouping

Table 3: List of fields extracted for analysis from the bug database

Not each field is mandatory. Columns `Due Date`, `Flags`, `Whiteboard`, `QA Contact`, `QA Whiteboard`, `Whiteboard`, and `Keywords` are voluntary, therefore data from these columns is incomplete. Furthermore, not every ticket blocks or depends on other tickets. However, the lack of blocking of or depending on other tickets is a value in itself, just as a ticket not being resolved is a valid state of a ticket.

4.4 Analysis

Bugs have a long, hard-coded life cycle in the Bugzilla tracking system. Such a system results in bureaucratic overhead but is necessary for open source software development. No ticket can be allowed to be favored, to maintain the system's neutrality; The process has to be repeatable and universal, so there's no exclusion from the system; the tickets have to be transparent, as tickets are a tool for communication as well.

The Spiral of Knowledge and the four types of Knowledge Transfer can be applied to a bug fixing process. The hardest steps of this process, the transfer between tacit and explicit knowledge, hasn't been widely investigated.

Researches about bug pattern matching algorithms and defect prediction combine explicit knowledge to create new explicit knowledge. However, such models and systems investigate questions and problems of software engineering. Contrary to this approach we examine key details of a bug life cycle from a knowledge management perspective.

4.4.1 Severity

How severe a bug is can be a highly subjective individual decision. However, examined data shows no signs of deviation. Knowledge is extracted and shared, forming a common understanding of the field.

This field signals the impact a ticket has on the product or product component. However, the severity values are inconsistent. Values `minor`, `normal`, `major`, `critical` signal the severity of the ticket reported. `Trivial` represents the difficulty of the issue at hand. `Blocker` and `enhancement` are labels that refer to the type of the issue. Reports with minor severity could be enhancements. Trivial issues can pose major impact on a product or its component. Furthermore, these values are relative to each other. This could be the source of a system-wide issue in a closed environment, however the impact is minimal due to the open nature of the bug tracker.

Figure 5 compares each severity value's performance in light of the tickets status and resolution. The percentage of bugs that are resolved a standard deviation of 0.08. Bugs marked `fixed` are in the range of 0.11. We can identify the`blocker` value as an outlier that enjoys priority over any other field. However, most other fields are almost identical.

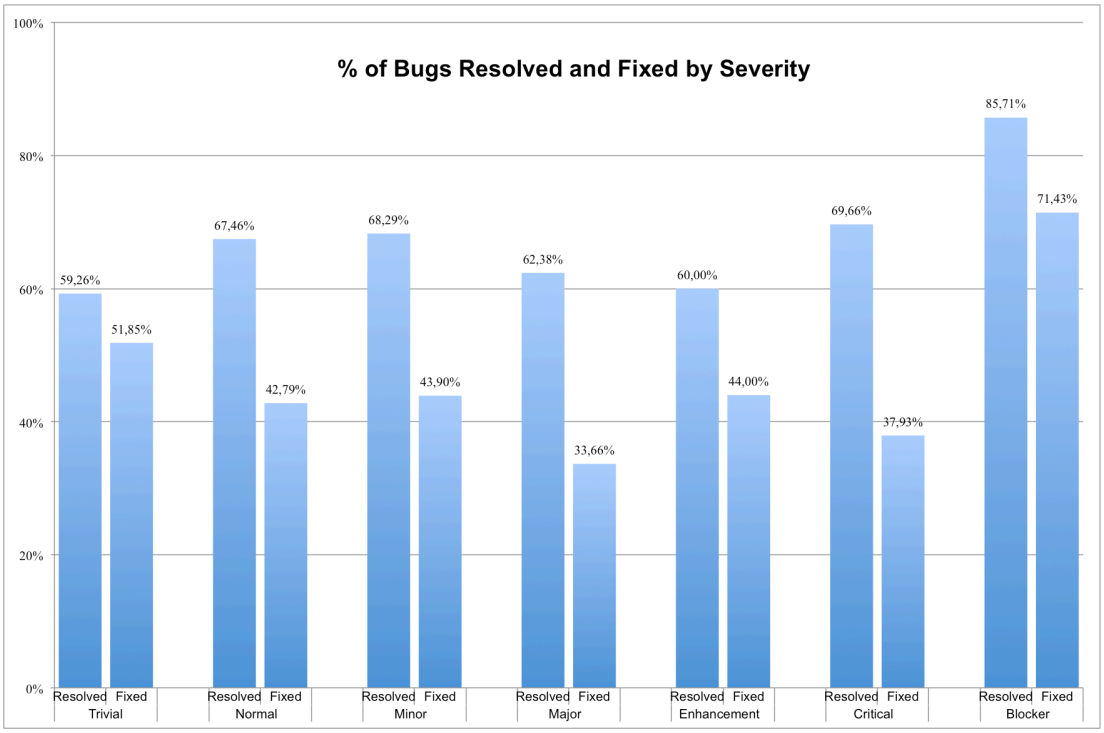

Fig. 5: Percentage of Bugs `Resolved` and `Fixed` by Severity

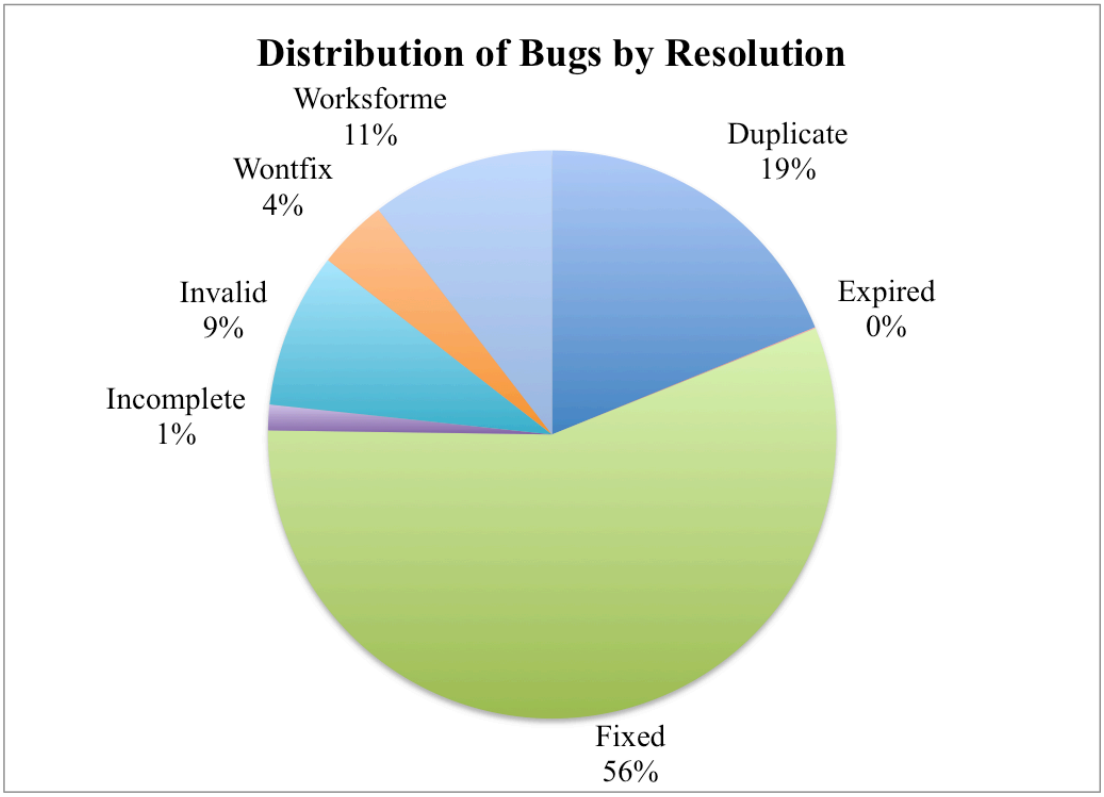

Figure 6: Percentage based pie chart of bugs by Resolution

This finding is especially valuable in light of Figure 6. 56% of every ticket created is marked as `fixed`. That this can be generalised, regardless of severity, means, that this open source software development is impact agnostic.

Examining the average time of integration also provides calculatable, reliable data. Days to integrate bug-fixes widely vary depending on the resolution (Figure 7.). Tickets resolved as `worksforme` (meaning, the developer the ticket was assigned to wasn't able to verify or reproduce the issue reported in the ticket) take an average of 223 days to reach resolution, while tickets deemed `invalid` are almost double as fast. Fixing bugs takes less time on average than tickets reaching the `wontfix` resolution. This, however, is not an issue, reaching resolution is the most important part and the goal of the bug-fix process.

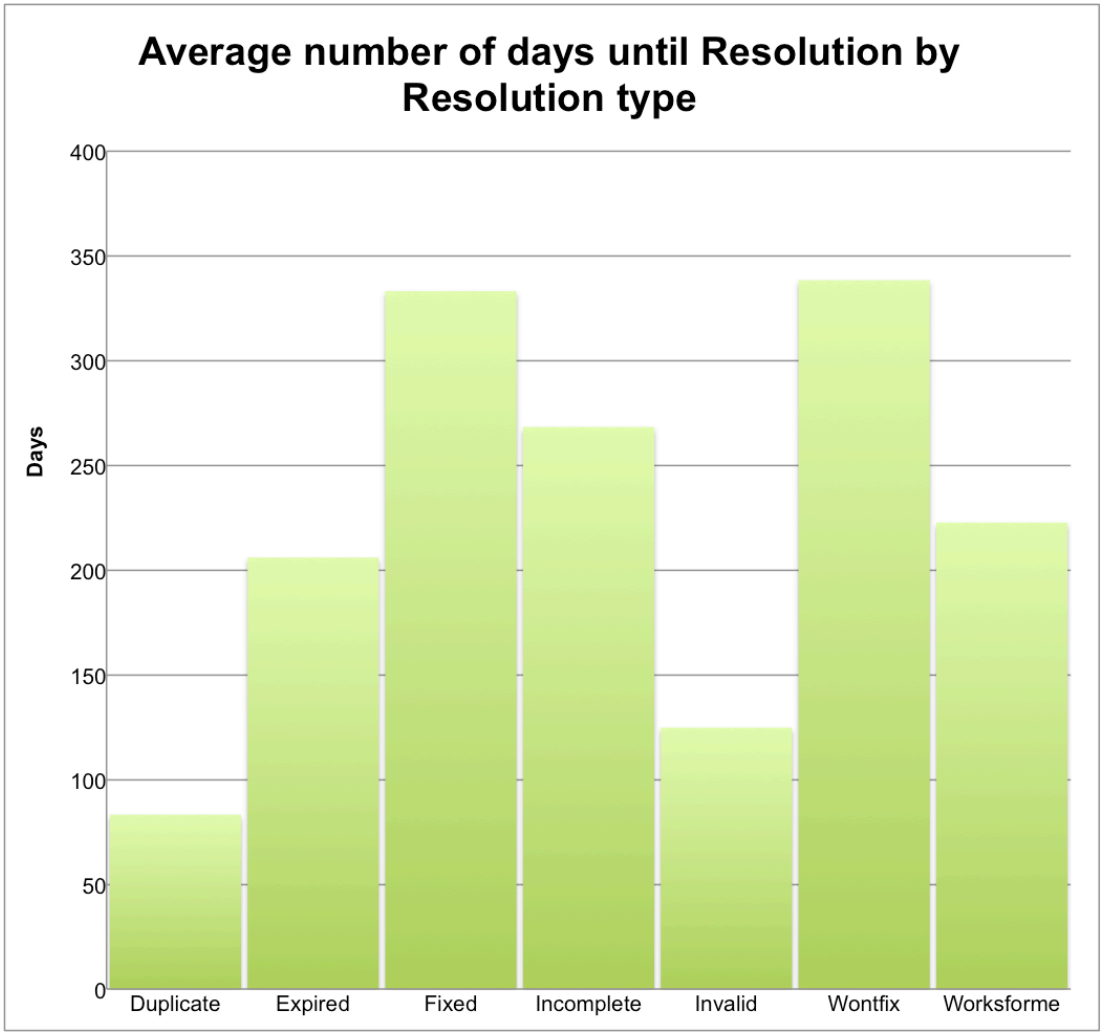

Fig. 7: Average of days for a ticket to be resolved, sorted by Resolution values

However, when sorted by severity, we can find a bell-curve-like pattern viewing average days it takes to reach a resolution. Average days it takes a ticket with `normal` severity to be integrated is 259 days. The mean is at 300 days. `Minor` issues and `enhancements` need double as many days to get resolved. Bugs with a higher impact take much less time fix.

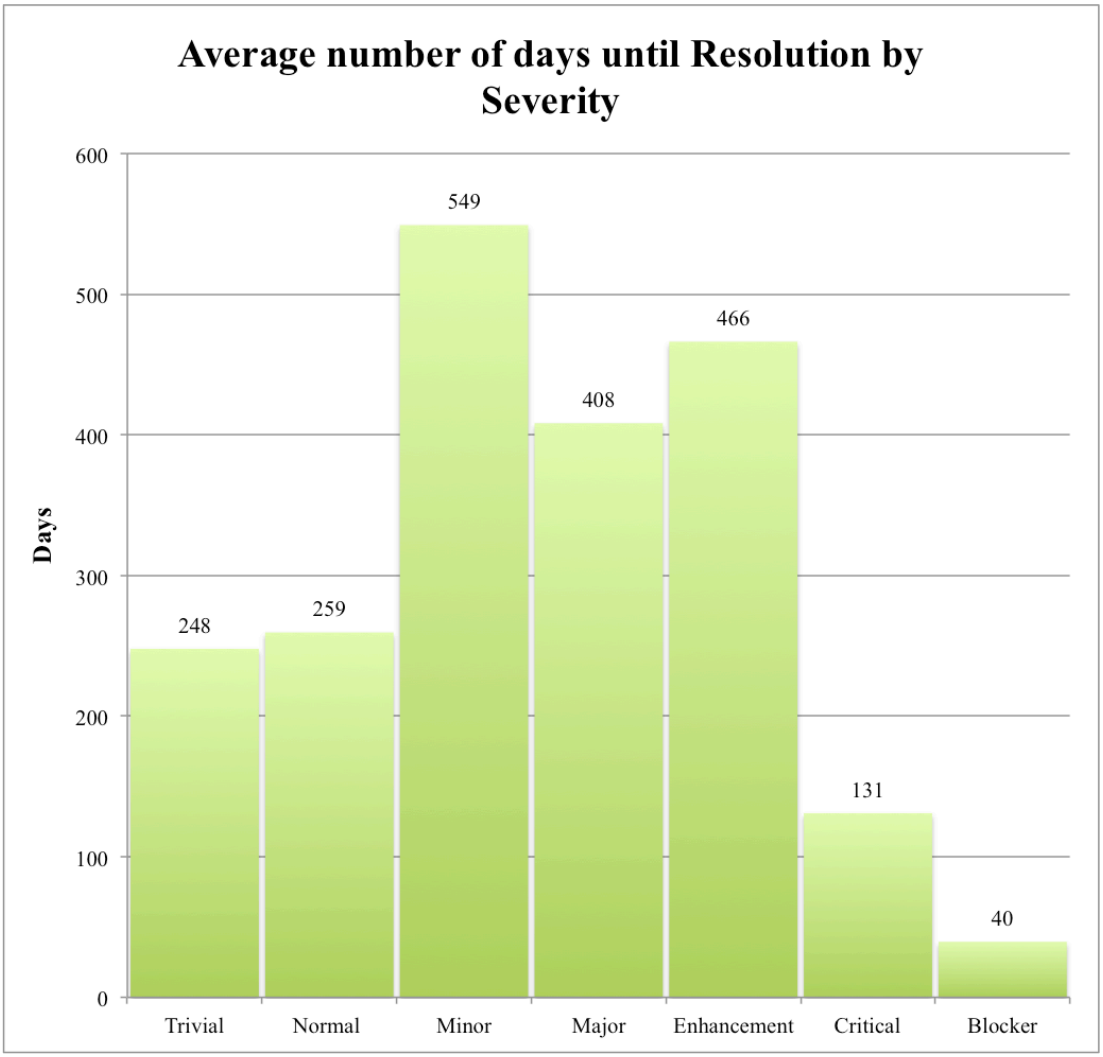

Fig. 8: Average integration time of different Severity values

Examining the number of outcomes by severity provided further proof that issues are universally indifferent to severity. The ratio of tickets in different resolutions is almost identical regardless of the severity, as it can be seen in Fig. 9.

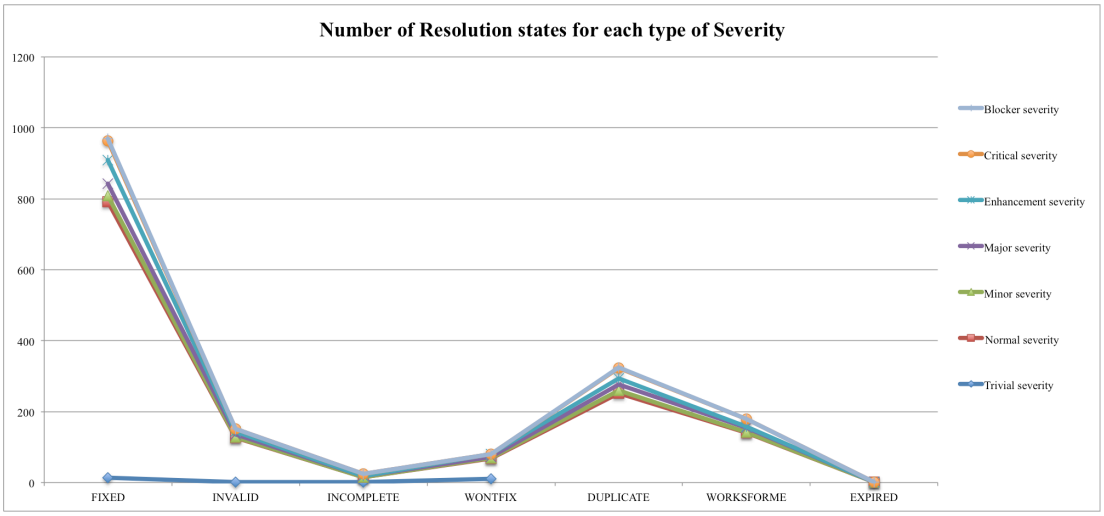

Fig. 9: Ticket resolutions for each Severity value

It can be concluded that organisational knowledge creation has no bias towards bug severity. Furthermore, there is no difference between features and bugs in actual development by severity.

4.4.2 Flags

`Flags` are custom fields in the bug database that allow to group, tag, and sort tickets that can help developers under certain circumstances. Solving edge-cases with a helping hand is useful for the assigned developer however externalising that knowledge through flagging can possibly benefit the whole organisation's Spiral of Knowledge.

Examining flagged and non-flagged tickets in the system reveals that externalising knowledge with `flags` is an effective method of improving bug-fix processes.

Figure 10: Tickets that are flagged are implented `fixed` compared to non-flagged tickets

A comparison between flagged and non-flagged tickets was made to determine if there is a difference in their resolution. As it can be seen in fig. 10, I found that tickets flagged have an 87% fix ratio while tickets not flagged have been marked `fixed` only 33% of the cases. With an average of 42% of bugs `fixed` across the whole database, we can conclude that flagging provides organisational knowledge that improves chances of success and non-flagged items perform below system average.

Flagged items also have a higher average of comments and a higher count and average of subscribers (`CC`) as well. While examining how comments drive organisational knowledge creation was out of the scope of this thesis, I feel it is important to note, and is a good topic for further research.

4.4.3 Dependency

Examining bugs from a network perspective is an important field of research with valuable findings. Determining factors of coupling, centrality, ties help understand relationships between issues and provide tools to optimise these relationships (Murgia, 2011). Limited research on social network analysis of bugs has been done (Zanetti et al., 2013), and is a potential field for further research.

We investigated the average dependency of tickets by mapping how often and how many times they were blocking another issue from being completed, and examined how many times they themselves were blocked. In order to draw comparisons between the performance of the Waterfall model and the hybrid model the database was split into two. We queried the database for tickets closed before 2011 April and for tickets opened after 2011 April. It is important to note that the blocking and blocked by values are an independent column so there is no time constraint that could potentially distort results.

Fig. 11 displays the findings. The higher the number the bigger the dependency of a bug. In order to better display the performance of models, the dependency-number of `blocking` issues and `blocked by` issues were charted separately. The `OLD` row charts the performance before the change in software development method was made in the April of 2011. The `NEW` row charts performance after the change, from the April of 2011 onwards.

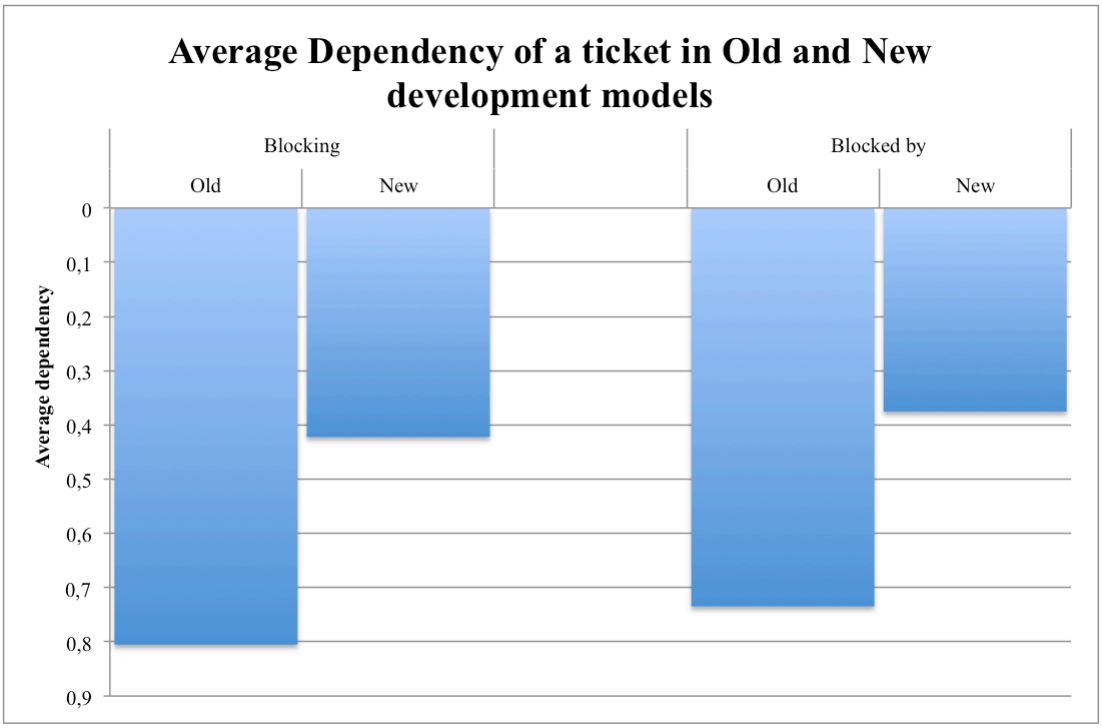

Figure 11: Dependency of tickets by software development models compared

It can be seen that tickets have a lesser dependency since the change. On average the dependency of tickets has shrunken by 51%. The dependency of tickets `blocking` is 0.42 compared to 0.81 of the old model. Frequency of any ticket given a `blocking` or `blocked by` value is lower as well. A ticket having a `blocked by` property is half as frequent since implementing the new software development model. These results can be seen in fig. 12.

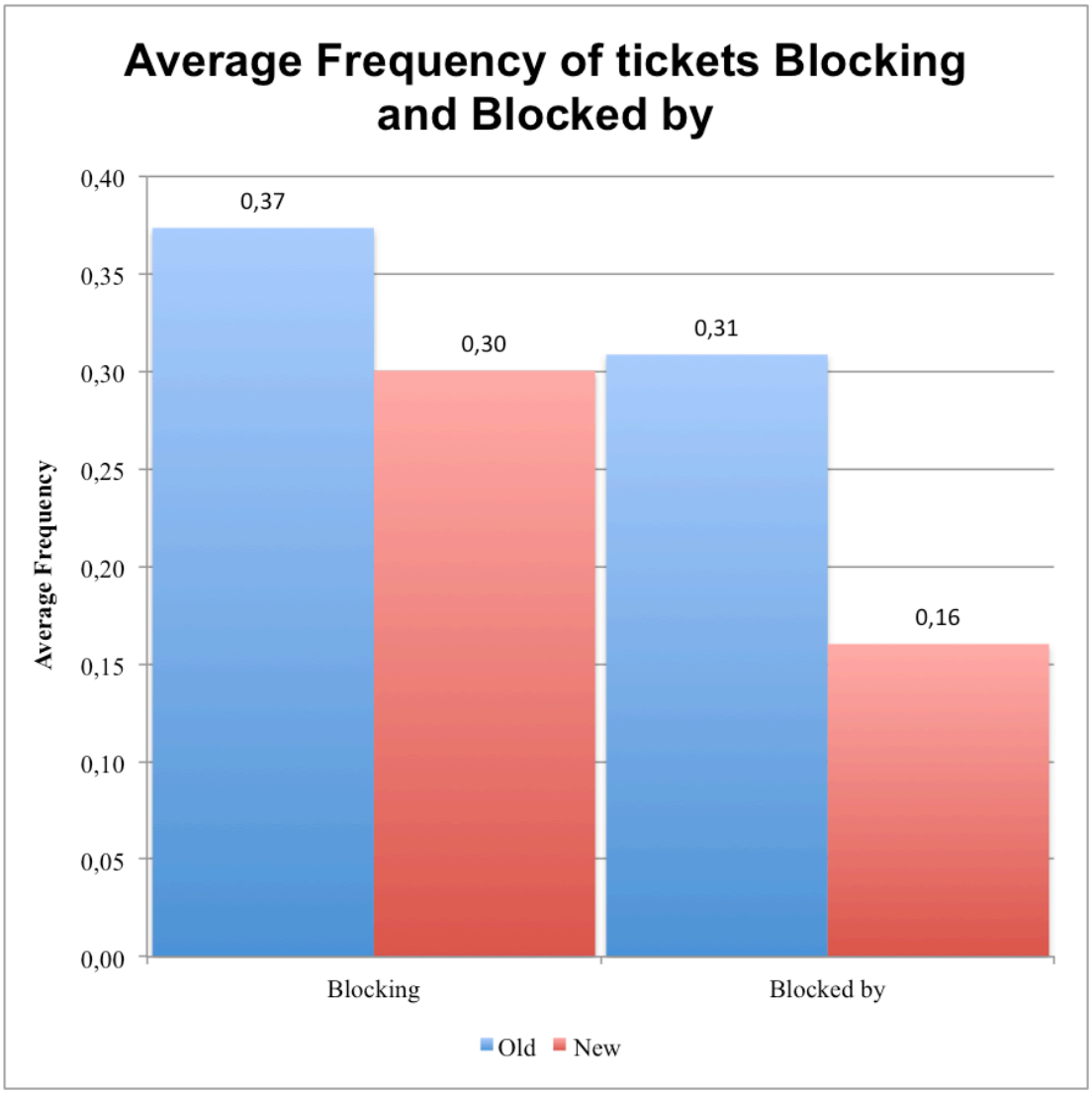

Figure 12: Average frequency of a bug `blocking` or `blocked by`

4.4.4 Voting

Ranking and rating issues can be decided by vote, which is why this system was developed. However, the voting system has no constraints: number of votes is limitless, there are potentially unlimited numbers of voters and an always changing number of bugs to vote on. There was no correlation between integration time of a bug and the number of votes it received. There's no indication that with more votes have a higher fix-rate. The dataset of the votes plotted against different fields from the bug database proves that votes serve an organisational not a technical purpose. Fig. 15 then shows the results of plotting vote distribution by severity.

Log(Votes) was used to normalise data and compensate for the number of tickets with `normal` severity, that otherwise rendered the table unusable. Each severity value is close to the median value of 1,5185 except the field `enhancement`. Tickets marked as enhancements collected an amount of votes that is close to the number given to `normal` tickets.

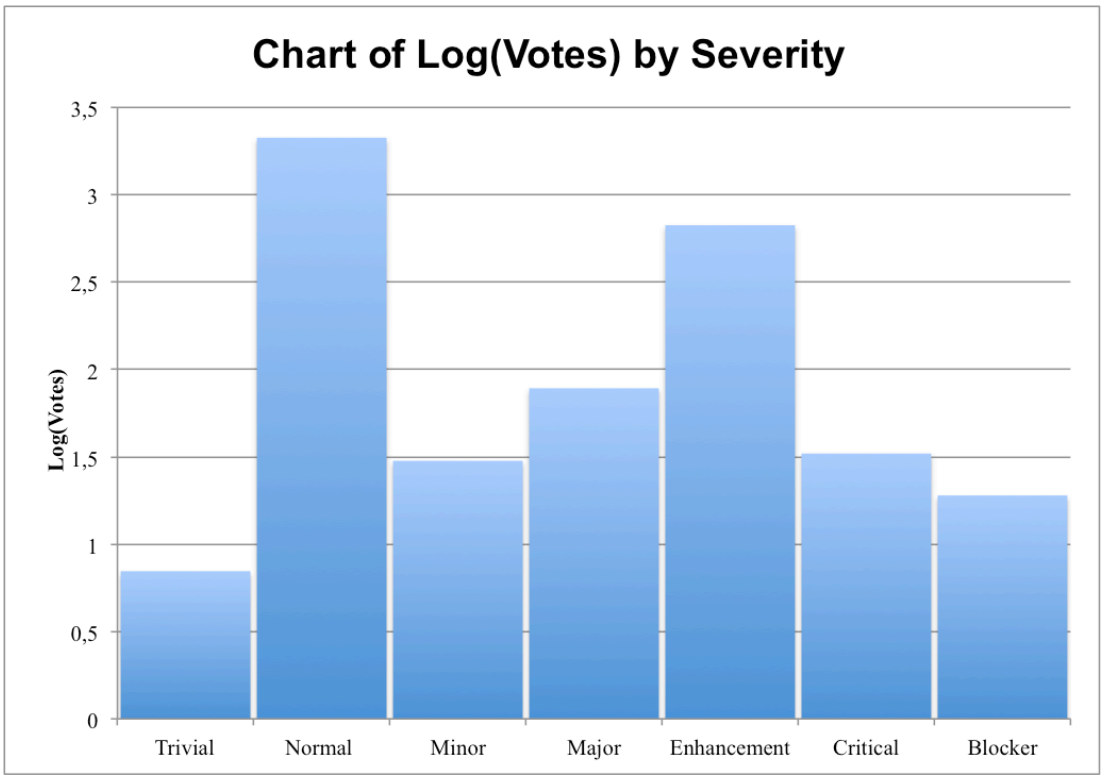

Fig. 15: Chart of Log(Votes), sorted by Severity

Tickets marked for enhancements get almost ten times as many votes as a major bug. Furthermore, this doesn't translate into better fix-ratio or shorter integration time, hence it can be deduced that votes are a unique way of externalizing individual tacit knowledge. This further proves that in software development differentiating bugs and enhancement requests is arbitrary from a project management perspective.

4.5 Conclusion

This research analysed the relationship between software development models and bug processes. The relationship was analysed from a knowledge management pers-

pective and applied those concepts in it's research and analysis of the findings. Results of the research done in this thesis can be summarised as:

- Agile, open source development shows no bias towards bug severity. Results of analysis found no difference between bugs and feature enhancements. Furthermore, data shows that severity has no impact on tacit knowledge externalisation.
- Codification of tacit knowledge improves bug-fixing processes. Flags proved to be useful in bug-fix processes. Flagged items outperformed non-flagged items. Non-flagged items were also shown to perform below the average in successful bug-fix implementations.
- Development models were proven to have an impact on bug dependencies. Switching to an agile model improved bug-fix statistics. Reversely, this can mean that effectively integrated bug processes can improve knowledge management and development.
- Voting was identified to have an organisational benefit. It has displayed that there is no separation of bugs and features from a knowledge management perspective.

4.6 Recommendations

Many investigated concepts and models for further analysis have been recommended throughout this thesis, for improving software development and bug management processes. To make these findings more useful, a summary with three categories has been added. These recommendations have been drawn from the research and work done on the Firefox HTML5 project specifically, however some might be applicable to different software projects as well.

4.6.1 Knowledge management integration

Integrating knowledge management into software development practices happens even if the organisation is unaware of it. However, purposefully adopting practices of knowledge management can help software development. It gives a better understan-

ding of the product development process. Methods adopted can be tailored to fit the organisation's purposes better. Advocating redundancy and autonomy improves performance. Mapping knowledge creation and the flow of the Spiral of Knowledge can help identify pain points. Training can be made more purposeful and knowledge transfer more efficient.

4.6.2 Integration

This research has proved that integrated bug processes improve development manyfold. Tacit knowledge is codified with the help of redundancy. Integrating processes lessens dependency and improves chances of success. It was shown that such integration save cost and developer time. Releasing product to users becomes faster and cycles of software development shorter. This increases value. Development teams and users have a direct relationship that can form product and increase user satisfaction as well.

4.6.3 Focusing on the process

One of the key findings of this analysis was that the process of bug management lacks bias towards or against enhancements. It doesn't favor one instead of the other. It was further proved by voting system that showed no correlation between bug-fix success-ratio or implementation time but signalled user interest in implementing enhancements. Giving flexibility to the developers and treating tickets as tasks regardless of their type (bug, enhancement, support, new feature, etc.) or severity (minor, critical, etc.) has been the selling point of some services.

4.7 Suggestion for Further Research

The main point of this thesis was to examine open source bug processes from a knowledge management perspective. Researching more open source projects or even closed source ones could build on the findings of this paper. Topics of further research could be analysing tacit knowledge externalisation in comments and whitebo-

ards. Furthermore, a less quantifiable approach to investigate user-developer relationships could also result in findings comparable with this research.

5 REFERENCES

Agilemanifesto.org, (2001). *Principles behind the Agile Manifesto*. [online] Available at: http://agilemanifesto.org/principles.html [Accessed 27 Sep. 2014].

Aziz, B. (2012). Improving Project Management with Lean Thinking?.

Begel, A. and Nagappan, N. (2007). Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study.

Bhattacharya, P. and Neamtiu, I. (2011). Bug-fix time prediction models: can we do better?. pp.207--210.

Bhavya, J., Rajeshwari, S., Nandita, M. and Ratnakala, (2012). Software Development Using Agile & Scrum.

Bright, P. (2014). *HTML5 specification finalized, squabbling over specs continues*. [online] Ars Technica. Available at: http://arstechnica.com/information-technology/2014/10/html5-specification-finalized-squabbling-over-who-writes-the-specs-continues/ [Accessed 30 Oct. 2014].

Brooks, F. (1995). *The mythical man-month*.

Computer World, (2011). Moth in the machine: Debugging the origins of 'bug'. [online] Available at: http://www.computerworld.com/article/2515435/app-development/moth-in-the-machine--debugging-the-origins-of--bug-.html [Accessed 12 Sep. 2014].

Foos, T., Schum, G. and Rothenberg, S. (2006). Tacit knowledge transfer and the knowledge disconnect. *Journal of knowledge management*, 10(1), pp.6--18.

HTML5test.com, (2014). *The HTML5 test - How well does your browser support HTML5?*. [online] Available at: http://html5test.com [Accessed 22 Nov. 2014].

Humble, J. and Farley, D. (2011). *Continuous delivery*. Upper Saddle River, NJ: Addison-Wesley.

Kim, S. and Whitehead Jr, E. (2006). How long did it take to fix bugs?. pp.173--174.

Lee, L. (2000). Knowledge Sharing Metrics for Large Organisations. *Knowledge*

Management: Classic and Contemporary Works.

Lehtinen, U. and Torkko, M. (2005). The Lean concept in the food industry: a case study of a contract manufacturer. *Journal of Food Distribution Research*, 36(3), p.57.

Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. and Zelkowitz, M. (2002). Empirical findings in agile methods. *Springer*, pp.197--207.

Middleton, P. (1994). Euromethod: The lessons from SSADM. pp.359--368.

Mohammed, N., Munassar, A. and Govardhan, A. (2010). A Comparison Between Five Models Of Software Engineering. *Citeseer*.

Murgia, A. (2011). *Time evolution and distribution analysis of software bugs from a complex network perspective*. Ph.D. University of Cagliari.

Murrell, K., Offerman, S. and Kauffman, M. (2011). Applying lean: implementation of a rapid triage and treatment system. *Western Journal of Emergency Medicine*, 12(2), p.184.

Naval History and Heritage Command, (1999). *Photo # NH 96566-KN picture data*. [online] Available at: http://www.history.navy.mil/photos/images/h96000/h96566kc.htm [Accessed 12 Oct. 2014].

Nonaka, I. (1994). A dynamic theory of organizational knowledge creation. *Organization science*, 5(1), pp.14--37.

Nonaka, I. and Peltokorpi, V. (2006). Objectivity and Subjectivity in Knowledge Management: A Review of 20 Top Articles. *Knowledge and Process Management*.

Osterloh, M. and Frey, B. (2000). Motivation, knowledge transfer, and organizational forms. *Organization science*, 11(5), pp.538--550.

Petersen, K., Wohlin, C. and Baca, D. (2009). The waterfall model in large-scale development. *Springer*, pp.386--400.

Philipson, G. (2004). 2 A short history of software. *Management, Labour Process and Software Development: Reality Bites*, p.13.

Poppendieck, M. and Poppendieck, T. (2003). *Lean software development*. Boston, Mass.: Addison-Wesley.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3), pp.23--49.

Royce, W. (1970). Managing the development of large software systems. 26(8).

Shapiro, F. (1987). Etymology of the computer bug: history and folklore. *American Speech*, pp.376--378.

Shivaji, S. (2013). *Efficient Bug Prediction and Fix Suggestions*. Ph.D. UC Santa Cruz.

Sinha, D. (1963). Phenomenology and Positivism. *Philosophy and Phenomenological Research*, 23(4), p.562.

Sutherland, J. and Schwaber, K. (2007). *The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process*.

Ullman, E. (1998). The dumbing-down of programming. *Salon*. [online] Available at: http://www.salon.com/1998/05/13/feature_320/ [Accessed 6 Sep. 2014].

Vestb√∏, T. (2007). *Software Quality in the Trenches*. MA. Norwegian University of Science and Technology.

Wang, F. and Chen, K. (2010). Applying Lean Six Sigma and TRIZ methodology in banking services. *Total Quality Management*, 21(3), pp.301--315.

Womack, J., Jones, D. and Roos, D. (1990). *The machine that changed the world*. New York: Rawson Associates.

Zanetti, M., Scholtes, I., Tessone, C. and Schweitzer, F. (2013). *Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities*. [online] Chair of Systems Design, ETH Zurich, Switzerland. Available at: http://arxiv.org/abs/1302.6764 [Accessed 15 Nov. 2014].