Department of Computer Science
University College London
University of London

# Structured Discussion and Early Failure Prediction in Feature Requests

*Camilo Edward Benedict Fitzgerald*


Submitted for the degree of Doctor of Philosophy
at University College London

2012

**“The only real failure in life is one not learned from”**

***- Anthony J. D’Angelo***

I, Camilo Edward Benedict Fitzgerald, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

# Abstract

Feature request management systems are popular tools for gathering and negotiating stakeholders' change requests during system evolution. While these frameworks encourage stakeholder participation in distributed software development, their lack of structure also raises challenges. We present a study of requirements defects and failures in large scale feature request management systems, which we build upon to propose and evaluate two distinct solutions for key challenges in feature requests.

The discussion forums on which feature request management systems are based make it difficult for developers to understand stakeholders' real needs. We propose a tool-supported argumentation framework, DoArgue, that integrates into feature request management systems allowing stakeholders to annotate comments on whether a suggested feature should be implemented. DoArgue aims to help stakeholders provide input into requirements activity that is more effective and understandable to developers. A case study evaluation suggests that DoArgue encapsulates the key discussion concepts on implementing a feature, and requires little additional effort to use. Therefore it could be adopted to clarify the complexities of requirements discussions in distributed settings.

Deciding how much upfront requirements analysis to perform on feature requests is another important challenge: too little may result in inadequate functionalities being developed, costly changes, and wasted development effort; too much is a waste of time and resources. We propose an automated tool-supported framework for predicting failures early in a feature request's life-cycle when a decision is made on whether to implement it. A cost-benefit model assesses the value of conducting additional requirements analysis on a body of feature requests predicted to fail. An evaluation on six large-scale projects shows that prediction models provide more value than the best baseline predictors for many failure types. This suggests that failure prediction during requirements elicitation is a promising approach for localising, guiding, and deciding how much requirements analysis to conduct.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I am indebted to my supervisors Emmanuel Letier and Anthony Finkelstein who have provided the balance of support and freedom necessary to complete a PhD without getting completely lost at sea, thereby making this thesis a body of work of which I am truly proud of. This research reflects throughout Anthony's astounding ability to inspire direction and vision of the bigger picture, and Emmanuel's refusal to accept anything less than the utmost scientific rigour, relevance and honesty.

Thank you Yijun Yu and Mark Harman for acting as external examiners for this thesis and providing feedback that has improved its quality and strengths. Shin Yoo, Yuanyuan Zhang, John Shawe-Taylor and Licia Capra for providing similar feedback at mock and internal VIVAs. John-James Bulstrode, Mark Fassler, Edmund Fitzgerald and Valpy Fitzgerald for their patient proof reading.

I wish to thank Carina Alves for taking me in as a 'visiting researcher' in UFPE, Recife in the first few months of my PhD and providing much useful advice on working with my supervisors and the years of research that would follow. All those I have worked alongside at UCL who have kept me entertained and sane over the years, especially Andy Maule, Pan Xueni, Aitor Rovira and Will Heaven. Also Benjamin Klöpper, Shinichi Honiden, and all those I worked with at the National Institue of Informatics in Tokyo, where I learned to stand on my own two feet as a researcher.

Finally, I am grateful to the EPSRC for supporting me financially, my family for providing a sound bedrock of unconditional love, and all the staff and regulars at the Kazbar and Cafe Tarifa, Oxford who have kept me in touch with the important things in life...

# 1 Introduction

*In this Chapter we describe the context of distributed software development and feature request management systems, and introduce a running example of a feature request from the Firefox project. We then motivate two key challenges in feature request management systems that this thesis addresses: Improving stakeholders' ability to explore their needs and making these needs more easily understood by developers; and deciding how, where and how much upfront requirements analysis should be performed in feature requests. Finally, we give a roadmap of the thesis and an overview of its contributions.*

## 1.1 Context

### 1.1.1 Distributed Requirements Engineering

The steady trend in the globalization of businesses and software development has given rise to many projects where stakeholders are geographically distributed [Herbsleb and Moitra, 2001]. Correspondingly, platforms have been developed to cater for the needs of web-based collaboration that present both opportunities and challenges. Effective communication and collaboration is difficult to achieve in the face of distributed knowledge management, web-based collaboration and asynchronous communication across time differences [Damian, 2004b] [Bird et al., 2008]. Requirements engineering activities are no exception to these challenges, and solutions are needed for all stages of the requirements lifecycle; namely elicitation, modeling and analysis, communication of requirements, agreement upon requirements and evolution of requirements [Nuseibeh and Easterbrook, 2000].

Tools for documenting and maintaining requirements across sites are plentiful; Rational DOORS and eRequirements are two examples of heavyweight desktop-based and lightweight browser-based tools respectively. These tools, however, only allow users to store, manage, and analyse requirements artefacts and do not provide a rich infrastructure for communication, collaboration and negotiation among developers and stakeholders.

The majority of medium to large distributed projects communicate on requirements using asynchronous messaging systems, such as forums, mailing lists or issue trackers [Bird et al., 2008]. Synchronous communications including conference calls and chat rooms are more often used by smaller or less distributed projects, and among smaller teams within larger projects such as senior management [Meyer, 2008]. As a project or team grows larger or more distributed, however, time differences and human resource management issues make the use of synchronous mediums difficult [Damian, 2004b], leading to a switch to asynchronous forms of collaboration. Collaboration via the use of asynchronous messaging systems suffers from both the standard requirements pitfalls (such as introducing conflicts into a specification and failing to fully understand stakeholder needs) and the challenges introduced by distributed collaboration (such as effective knowledge management and facilitating collaboration).

The *de-facto* standard for distributed requirements collaboration in medium to large projects are mailing lists and specialised forums. Specialised forums centred on the concept of changes requests, or *feature request management systems*, are particularly popular in larger projects as they help to manage the process of dealing with large volumes of data and communication activity [Laurent and Cleland-Huang, 2009], while maintaining a platform for free and open discussion. Addressing requirements difficulties in these environments without infringing upon open discussion and low barriers to entry is a current research challenge [Cheng and Atlee, 2007].

### 1.1.2 Feature Request Management Systems

Most distributed medium to large scale open-source projects and an increasing number of enterprise-level projects rely on web-based feature request management systems to collect and manage change requests [Laurent and Cleland-Huang, 2009]. These apply feature-centric development, where a feature can be defined as "a unit of change in the software product" [van Lamsweerde, 2009]. Feature request management systems allow stakeholders and developers to suggest new features, or *feature requests*, and subsequently collaborate on deciding whether to implement the feature, developing the feature, and

integrating it into the product. In Section 3.1 we describe in detail the framework and processes of a typical feature request management system.

The term feature request management forum is commonly used in academic literature for descriptive purposes [Laurent and Cleland-Huang, 2009]. In practice they are often simply referred to as *issue trackers*, which is the term for the larger system into which they are usually incorporated. Issue trackers make use of the forum structure to facilitate collaboration and management on all stages of the software development lifecycle; including the management of work tasks, and the reporting and fixing of bugs.

A *stakeholder* is an individual who can affect or is affected directly or indirectly by the achievement of the a software project's objectives [Freeman, 1984]. In feature request management systems any stakeholder can potentially contribute new feature requests and comment in existing ones. In this thesis we use the term *developer* to refer to stakeholders who have additional capabilities of assigning feature requests for implementation, rejecting them, developing code, and integrating implemented features into the product. Finally, we use the term *project manager* to reference individuals responsible for overseeing the entire body of feature requests and managing project outcomes.

The contributions of this thesis have been developed and evaluated using the Bugzilla issue tracking system, but could be applied to other feature request management systems such as IBM Jazz[1] and JIRA[2]. Bugzilla combines the reporting of bugs and feature requests; feature requests threads are distinguished from bug reports by being marked as 'enhancement requests'. Examples of projects using Bugzilla are the Firefox project[3], Eclipse[4] and Facebook[5] - whose products range from open-source to commercial and from mass-market web-based tools to specialist software development environments.

[1] `http://jazz.net/`
[2] `http://www.atlassian.com/software/jira/`
[3] `https://bugzilla.mozilla.org/`
[4] `https://bugs.eclipse.org/bugs/`
[5] `http://bugs.developers.facebook.com/`

### 1.1.3 A Running Example: Firefox Feature Request #171702

Throughout this thesis we use a real feature request from the Firefox project as an illustrative example, which is shown in Appendix A. The feature request is one of over 6,000 instances in the Bugzilla issue tracker for Firefox. It is concerned with the addition of a new feature to Firefox which would allow users to customise the menu bar at the top of the browser window. It was first suggested in September 2002 and integrated into the product 10 months later.

The example illustrates the typical process by which a feature request is developed. The first 12 posts are mostly concerned with requirements negotiation (whether or not the suggested feature would be a good addition to Firefox and should be assigned for implementation). Posts 13 to 21 are mostly concerned with the implementation of the feature, and in post 19 a code attachment is submitted to the feature request containing an implementation. The final posts are concerned with the verification of the code and its integration into the product. The current status of the feature request, meanwhile, can be seen from the meta-data displayed in its header. A history of the changes made to the feature request's meta-data over its life-cycle can also be accessed (shown at the end of Appendix A).

## 1.2 Challenges in Feature Request Management

Project managers and developers are responsible for managing feature requests. This essentially involves making decisions on how to move a feature request forward in its life-cycle and updating its meta-data accordingly. Many projects deal with extremely large volumes of feature requests. At the end of 2010 the Firefox project contained 6,379 instances, 483 of which were active in the last three months; Eclipse contained 46,427 feature requests of which 5,155 were active in the same period. Discussions in feature requests are often quite lengthy, as exemplified by #262459[6] of the Firefox project, which contains 64 posts, approximately 4600 words and lasted four years. The sheer volume of

[6] `https://bugzilla.mozilla.org/show_id=262459`

this data makes analysis, negotiation and prioritization of feature requests very difficult for stakeholders, developers and project managers alike.

A survey based study by Paula Laurent and Jane Cleland-Huang [Laurent and Cleland-Huang, 2009] reported that stakeholders in feature request management systems identified the following issues to be those that predominate feature requests:

1. Finding discussions on a topic is difficult. Stakeholders with similar interests do not, therefore, engage in a shared discussion of their needs.
2. Despite the active discussions in many of the forums, project managers and developers find it hard to truly understand stakeholders' needs.
3. Analysis, negotiation and prioritization of requirements becomes very difficult leading to poor requirements decisions being made.

The first problem has received a relatively large amount of attention in the literature. Machine learning approaches have been proposed to reduce the spread of discussions on similar topics by automatically detecting duplicate feature requests [Sun et al., 2010], and to make it easier for stakeholders to contribute to discussions that interest them by grouping similar feature requests [Cleland-Huang et al., 2009]. Other machine learning approaches have been proposed for assisting the process of triaging (categorising and prioritising new feature requests) [Laurent et al., 2007] [Anvik et al., 2006], which could improve the structure of a body of feature requests and make it easier for stakeholders to find topics relevant to their interests.

The second and third problems, meanwhile, have lacked attention in the literature. Sections 1.2.1 and 1.2.2 frame these problems in terms of unanswered questions; and in Chapters 4 and 5 we propose and evaluate two distinct tool-supported solutions that integrate into feature request management systems to address them.

### 1.2.1 Understanding Stakeholders' Needs

The open discussion forums on which feature request management systems are based encourage stakeholder participation, but also lack structure. Stakeholders' comments often go unnoticed, which frustrates them and leads to features being developed that satisfy their needs to a lesser degree [Laurent and Cleland-Huang, 2009].

Clarifying and structuring the complexities of a body of arguments in collaborative online forums is part of a wider problem that has been approached by the sense-making and design rationale community [Klein et al., 2006]. Discussions in online forums have been found to typically suffer from problems of noise, flawed argumentation, scattered content and the 'soapbox problem' (where the opinions of those who speak the most are over-represented) [Klein and Iandoli, 2008].

A prominent question in this context is therefore: *"Is it possible to structure feature request discussions so that stakeholders' needs can be expressed and understood more effectively without significantly altering existing communication practices?"*. A good answer to this question would provide a solution that encourages stakeholders to explore and express their needs more clearly, and makes it easier for developers and project managers to understand and respond appropriately. Such a solution, however, should maintain an environment where stakeholder participation is encouraged through free discussion, so as to maximise adoption in real feature request management systems.

We present a tool-supported argumentation framework in Chapter 4 as an answer to this question. Stakeholders' expression of their needs could be improved by encouraging them to explicitly state if their comments contain concepts relating to requirements defects and the exploration of alternative design spaces. Further, explicitly labelling comments in this way allows developers to better understand and act upon them.

### 1.2.2 Guiding Upfront Requirements Analysis

The costs of rectifying product and process failures (errors in a software product or the software development process) can be reduced by performing additional requirements analysis before beginning to implement a feature; including requirements elicitation, communication and negotiation, validation, and documentation [Nuseibeh and Easterbrook, 2000] [van Lamsweerde, 2009]. In feature requests these activities typically lack attention, leading to poor decisions being made as to whether and when to implement newly suggested features [Laurent and Cleland-Huang, 2009].

An important question that would assist developers with requirements analysis in feature request management systems is ‘‘*Where and how should upfront requirements analysis be focused to reduce process and product failures?*”. A good answer to this question should identify which feature requests would benefit most from requirements analysis, and how to direct this analysis to maximise its benefits.

The question remains, however, of whether it is worth the effort of performing additional upfront requirements analysis before a feature is assigned for implementation: *“When is the benefit of performing upfront analysis outweighed by the costs?”*. Requirements engineering literature typically suggests that more needs to be done, and that it is much more expensive to fix defects during implementation than early on in the requirements phase (5 to 10 times more for smaller projects and between 10 and 100 times more for larger ones) [Boehm and Papaccio, 1988] [van Lamsweerde, 2009] [McConnell, 2004]. In practice, however, many managers say “I know that we should work out the requirements in detail, but we don’t have time. We have to get started on the programming because we have a short deadline to deliver the code!” [Berry et al., 2005]. A good answer to this question should provide an indication of whether the costs of performing extra upfront requirements analysis are likely to be less than the resulting value gained by reducing process and product failures.

We present an early failure prediction framework in Chapter 5 as an answer to these questions. Providing early information to developers on which feature requests are likely

to fail and the types of failure that may occur can help them make more informed decisions on where and how to focus upfront requirements analysis. Further, the approach uses a cost-benefit model to assess the value of acting upon such a set of predictions, providing information on whether performing this additional upfront analysis is likely to provide benefit to a project.

## 1.3 Overview of Contributions and Thesis Structure

The main contributions of this thesis are the following:

- We define a taxonomy of failures in feature requests that extends the usual code failures to include process failures such as abandoned development, stalled development and rejection reversal, which are important to assess the consequences of requirements defects. We present a study of these failures in seven large scale projects: Apache, Eclipse, Firefox, KDE, Netbeans, Thunderbird, and Wikimedia. The study shows that these failures occur in abundance and incur significant costs.

- We present a tool-supported argumentation framework for structuring and reviewing discussions about feature requests, which encourages stakeholders to explicitly identify defects and explore design alternatives. The framework aims to make it easier for stakeholders to express their needs, discuss them in a constructive way, and reduce requirements defects. A case study evaluation of the framework on 50 feature requests from the Firefox project suggests that it requires little additional effort to use, and that it captures the key discussion concepts on whether to implement a given feature.

- We present a tool-supported framework for generating and evaluating early failure prediction models from historical feature request data. Project managers and developers can use this framework to reduce project costs by performing additional upfront requirements analysis on feature requests with a high probability of failure before taking a decision on whether to implement them.

- We define a cost-benefit model for evaluating whether the benefits of acting upon a set of failure predictions will outweigh the costs - thereby determining whether a project can expect to gain value from the use of a given failure prediction model.
- We present experiments on predicting failures in six large scale projects: Apache, Firefox, KDE, Netbeans, Thunderbird, and Wikimedia (excluding Eclipse due to the poor quality of data for our experiments). The experiments suggest that the early failure prediction approach could be a useful strategy for guiding upfront requirements analysis in a feature request management system and reducing the cost of software development. The results also indicate the types of failure that may be more susceptible to early predictions, and the characteristics of feature request discussions that act as good predictors.

This thesis is structured as follows. In this Chapter we have reviewed the context of distributed collaboration on requirements activity in feature request management systems and motivated two key challenges in feature request management systems: making stakeholders' voices count; and guiding upfront requirements analysis. Chapter 2 contains a review the literature on which this thesis is grounded. Chapter 3 describes the life-cycle of a feature request and studies the types of defect and failure that can occur in them. Chapter 4 presents and evaluates the tool-supported argumentation framework structuring feature request discussions. Chapter 5 presents the tool-supported framework for constructing and evaluating early failure prediction models, the cost-benefit model for assessing and comparing the value of early failure predictions, and reports the experiments on the evaluation of early failure prediction models for five types of failure in seven large scale projects. Chapter 6 concludes the thesis by stating its contributions, limitations and future perspectives.

Material from Chapters 3 and 5 has been published at the 19th IEEE International Requirements Engineering Conference [Fitzgerald et al., 2011], and in an extended version of the paper in the Requirements Engineering Journal [Fitzgerald et al., 2012].

## 1.4 Chapter Summary

Feature request management systems have emerged as a leading platform for communication and collaboration on requirements activities in distributed software development. Such systems have presented many opportunities for engaging stakeholders and encouraging wide-spread involvement in requirements management. Challenges have also arisen: stakeholders' needs can be difficult to understand and act upon appropriately; and information is difficult to come by on what type of upfront requirements analysis should be carried out in which feature requests, and whether this analysis will provide value to the project.

In this thesis we propose and evaluate two distinct tool-supported frameworks that respond to these challenges. An argumentation framework that encourages stakeholders to explicitly express and explore their needs, and an early prediction framework that predicts which types of failure are likely to occur in which feature requests and whether performing additional upfront analysis on them will be of value to a project. The Chapter that follows contains a study on defects and failures in feature requests that provides the grounding for these two approaches.

# 2 Background

*This Chapter presents the literature on which this thesis is grounded and which it extends. We begin with the work relevant to the DoArgue framework in Chapter 4: Collaborative requirements review, structured multi-party discussions and design rationale theory. This is followed by work relevant to the early failure prediction framework in Chapter 5: Machine learning approaches for predicting software failures from a project's historical data and their practical applications.*

## 2.1 Requirements Review and Structured Discussion

### 2.1.1 Collaborative Requirements Review

A project's success is crucially dependant upon requirements specification reviews to remove requirements defects such as ambiguities, inconsistencies and conflicts [van Lamsweerde, 2009]. The cost of fixing these defects early on in the requirements phase as opposed to during later implementation is typically 5 to 10 times less for smaller projects and between 10 and 100 times less for larger ones [Boehm et al., 1988] [Lamsweerde, 2009] [McConnell, 2004]. Fagan is the first to define a software inspection process, which consists of 5 stages: 'overview' and 'preparation' stages to educate a review team, an 'inspection' stage to find defects, a 'rework' stage to fix them, and 'follow-up' stage to ensure all errors are fixed correctly [Fagan, 1976]. The exact review process, however, has been shown to be dependent upon the context in which it is used. For the selection of an inspection team, for example, Freimut suggests that system developers and domain experts should lead design reviews [Freimut et al., 2005], while at Microsoft it was found that stakeholders with lower levels of computing expertise (degrees in mathematics or physics as opposed to computer science) found more defects [Carver et al., 2008].

Software projects are becoming increasingly more global, and there is a need for platforms to enable all stakeholders in the software development process to collaborate and communi-

cate effectively across time zones and geographical locations [Herbsleb and Moitra, 2001]. Software requirements review is no exception, and a body of literature exists aimed at facilitating this process in a web-based environment. Finkelstein and Fuks made an early attempt to facilitate the review of text based requirements passed between stakeholders (such as via email), in which they propose a set of standardised mark ups on statements made within textual requirements documents. These denote, for example, challenges on the grounds for a statement or questions regarding a statement [Finkelstein and Fuks, 1989]. The recent ease of web-based tool development has seen an explosion in frameworks that facilitate collaborative requirements review. These include Decker et al.'s wiki platform that facilitates the review of use-case oriented requirements [Decker et al., 2007], and WikiWinWin which facilitates the review and negotiation of early stage requirements using the WinWin approach [Yang et al., 2008]. Yet another example is StakeSource, which discovers and selects stakeholders to conduct the prioritisation and negotiation of requirements using a social network [Lim et al., 2011].

The frameworks described above have not seen much usage in practice, with projects preferring to use existing collaboration and communication frameworks such as feature request management systems [Laurent and Cleland-Huang, 2009]. These, however, are a long way from overcoming the issues arising from geographically distributed software engineering (discussed in Section 1.2). In Chapter 4 we propose the DoArgue framework, which facilitates collaborative requirements review in feature request management systems, rather than proposing an alternative framework.

### 2.1.2 Structured Online Discussion

There is a long tradition of attempting to structure collaborative discussions dating back to the Socratic method. In a seminal paper Mackenzie presents a formal system of dialogue that attempts to model the essential elements of multi-party argumentation [Mackenzie, 1984]. Collaborative discussions between two parties are modelled using concepts such as questions, challenges, grounds. Following from this, a field has flourished within computer science to address the problems of the collaborative exploration and comprehension of

an argumentation space known as design rationale, or sense-making [Klein et al., 2006]. Many practical web-based frameworks have been developed to logically map natural arguments, including the MIT Collaboratorium [Klein and Iandoli, 2008], Compendium [Shum et al., 2006], and Cohere [Shum, 2008]. These tools differ in their implementations and concepts but all contain some form of the following: 'questions/issues' responded to by 'answers/ideas', which are justified by 'arguments/justifications' [Shum, 1996].

Figure 2.1 shows an example of how the Compendium tool can be used to map arguments and record design decisions surrounding the implementation of a new software system. Questions are represented as question marks and are used as the root of an argument map (such as "How do we implement system X"). Ideas are represented as light bulbs and respond to questions denoting possible approaches to resolving them (such as the suggestions of "Do it ourselves", "Hire consultants", and "Co-develop"). Positive and negative arguments can be made for or against ideas, and are denoted by plus and minus signs respectively. Compendium also allows for new questions to be placed on any node (such as those arising from the root question, the idea "Do it ourselves", and the negative argument on in the example). The example shows how a collaborative discussion can be intuitively mapped producing an effective visual representation of a complex set of arguments.

A popular underlying framework on which many design rationale tools are based is the IBIS framework [Conklin and Begeman, 1988]. The Compendium tool implementation of the IBIS framework, for example, has been downloaded 40,000 times, has been used by NASA to map their mission control systems, and by Wisconsin Public Television to map their shared knowledge and assets[1]. The tool has been shown to facilitate the process of articulating the key issues, alternatives and trade-offs when used to map collaborative arguments on design decisions [Shum, 1996]. In Chapter 4 we describe this framework further and explain how we have built upon it to develop the DoArgue framework for facilitating collaborative requirements review in feature request management systems.

[1] `http://compendium.open.ac.uk/`

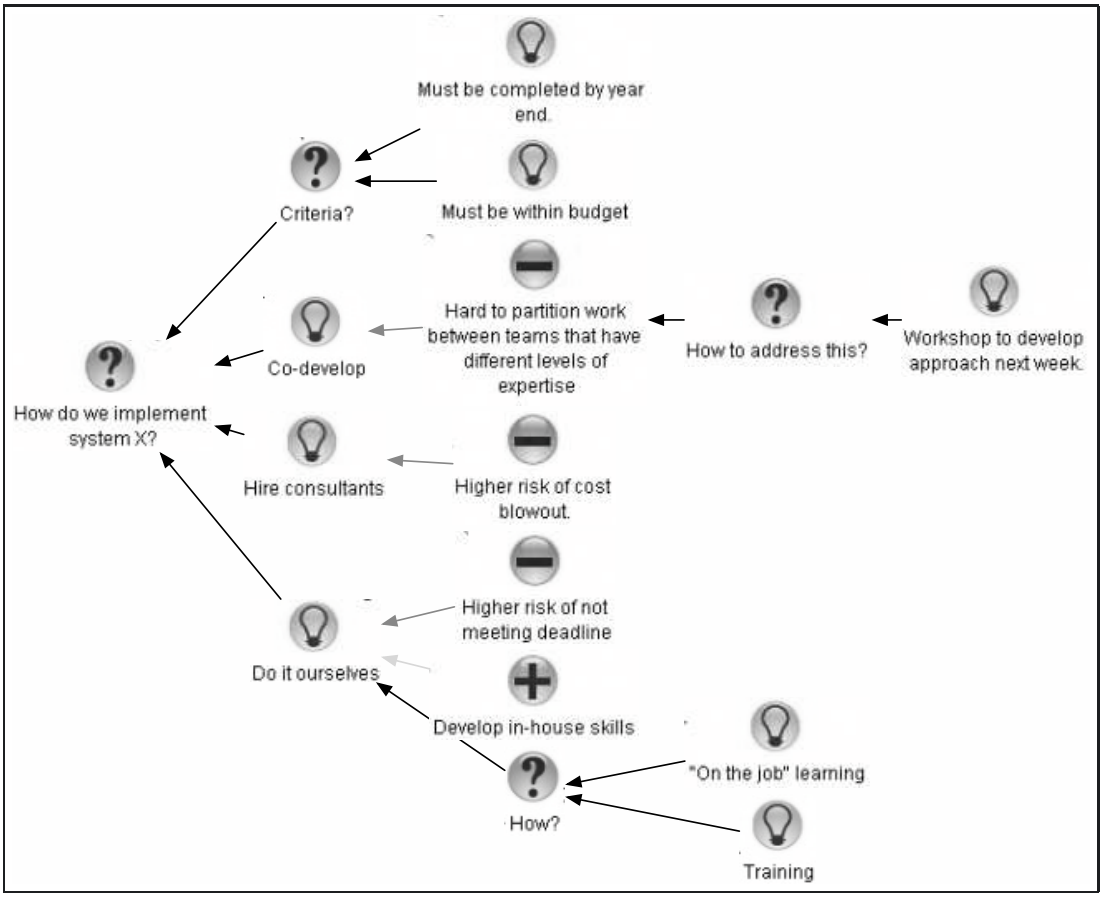



Figure 2.1: Compendium Argument Map: Implementing a New System

## 2.2 Automated Failure Prediction

### 2.2.1 Machine Learning for Prediction

With the increasingly large amount of easily-accessible historical data on software projects, machine learning has become a powerful and practical approach to automatically facilitate many aspects of the software development process [Zhang and Tsai, 2003]. This includes estimating the software development effort for a project [Srinivasan and Fisher, 1995], improving the detection and removal of errors in the software testing process [Briand, 2008], and the automatic discovery of traceability links within software artefacts [Cleland-Huang et al., 2010].

Machine learning approaches can also be applied to automatically predict which software artefacts (components, files, requirements, feature requests, etc.) might cause failures to occur. Such approaches benefit projects by providing information on where development effort should be focused to avoid failures. Historical data on the complexity and textual content of code packages and files in a software repository have been used to predict the number of bugs reported against them [Nagappan et al., 2006, Zimmermann et al., 2007,Zeller et al., 2011]. The communication structure between developers and their peers, meanwhile, has been used to predict whether code they integrate into the software product will result in build failures. In Chapter 5 we extend this body of research by automatically predicting failures much earlier in the development life-cycle (from historical information on requirements activity as opposed to coding activity), and extend the types of failures predicted to cover software process failures in addition to product failures.

### 2.2.2 Practical Applications

Current tools that automatically predict failures from a project's historical dataset do not yet play a significant role in real software development projects. Tools are available on the market that predict overall software reliability from causal models based on the software

development practices used by a project, such as for example λPredict[2] and AgenaRisk[3]. The approach used by these tools differs in that it does not make use of a specific project's historical data, but instead bases predictions on known outcomes of previous projects. Further, predictions are made on the project as a whole, as opposed to pinpointing which specific software artefacts are at risk of failure.

Existing literature on automated failure prediction present their results using the measures of precision and recall. Wolf et al., for example, predict build failures from the communication structure of developers with a recall of 0.55-0.75 and a precision of 0.5-0.76 [Wolf et al., 2009]. A practical tool, meanwhile, would benefit from a self-evaluation in terms of risk management or the value a project could stand to gain. AgenaRisk, for example, presents the results of predictions as information to make a trade-off between time, resources and quality [Fenton et al., 2007]. In Chapter 5 we extent the existing evaluation measures of recall and precision used by similar failure prediction approaches: A cost model is provided that estimates the value that a project could gain from conducting more requirements analysis on the software artefacts (feature requests) that are predicted to fail.

## 2.3 Chapter Summary

In this Chapter we have described the advances made in supporting collaborative requirements review in web-based environments. The frameworks proposed in the literature, however, have not been widely adopted and projects prefer to use existing web-based environments for software development such as feature request management systems. We have also presented the background on design rational theory that we build upon in Chapter 4 to develop the DoArgue framework, which facilitates collaborative requirements review within existing feature request management systems.

We have described the background literature and current state-of-the-art for automated failure prediction from historical datasets in software projects. In Chapter 5 we extend the

[2] http://predict.reliasoft.com/
[3] http://www.agenarisk.com/

existing body of work by: predicting development process failures in addition to product failures; predicting these failures earlier in the software life-cycle at the requirements stage; and providing a cost-benefit model to assess whether value can be gained from acting upon a set of predictions.

# 3 Defects and Failures in Feature Requests

*In this Chapter, we explore and study the context of feature request management systems specific to the two main contributions of this thesis: a collaborative argumentation framework, and an early failure prediction framework. The studies also reveal information on the types of defects and failures that are specific to feature request management systems.*

*We begin by defining the lifecycle of a feature request in such a way that it can be automatically traced from the meta-data of an existing feature request management system. An exploratory study of the current practice of reporting defects in the Firefox project suggests which types of defects may be relevant to feature requests. Five types of product and process failures that can be automatically identified from a feature request's path through its life-cycle are defined. Finally, a study of these failures in seven large-scale projects shows that they occur in abundance and incur significant costs.*

## 3.1 The Life Cycle of a Feature Request

The typical framework of a feature request management system is shown in Figure 3.1. Stakeholders and developers submit new feature requests by creating a new thread representing them, which is then used to contribute to discussions about the feature and track its status and development. Contributions are made to feature request discussions by submitting posts to the associated discussion thread. Once submitted, these posts may not be edited. Each feature request is associated with meta-data that records its current state (e.g. whether it is opened, in development, or integrated in the product), to which developer it has been assigned, and once developed whether there are bug reports that have been linked to the feature. Developers can also share code implementing a feature by submitting it to the feature request's thread. When code for a feature is fully developed and submitted a reviewer can verify the code and integrate it into the product. The meta-data that can be associated with a feature request varies across systems.

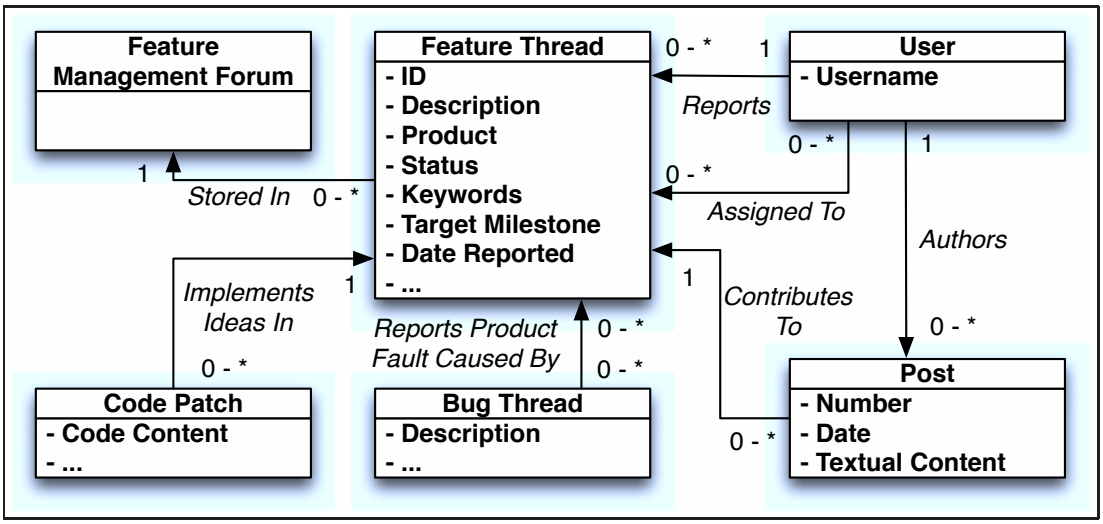



Figure 3.1: Feature Request Forum Framework

The typical life-cycle of a feature request is shown in the state transition diagram of Figure 3.2. When first created, a feature request is in the state 'New'. In this state, a decision needs to be made to either reject the feature or assign it for development. When a decision is made to assign the feature for development, it becomes 'Assigned'. The state 'Code Submitted' denotes that code implementing the feature request has been submitted. This is a submission to the feature requests' thread and does not necessarily imply that it has also been submitted to a project's code repository. The 'Verified' state denotes that code implementing the feature request has been verified and is ready to be integrated into the product.

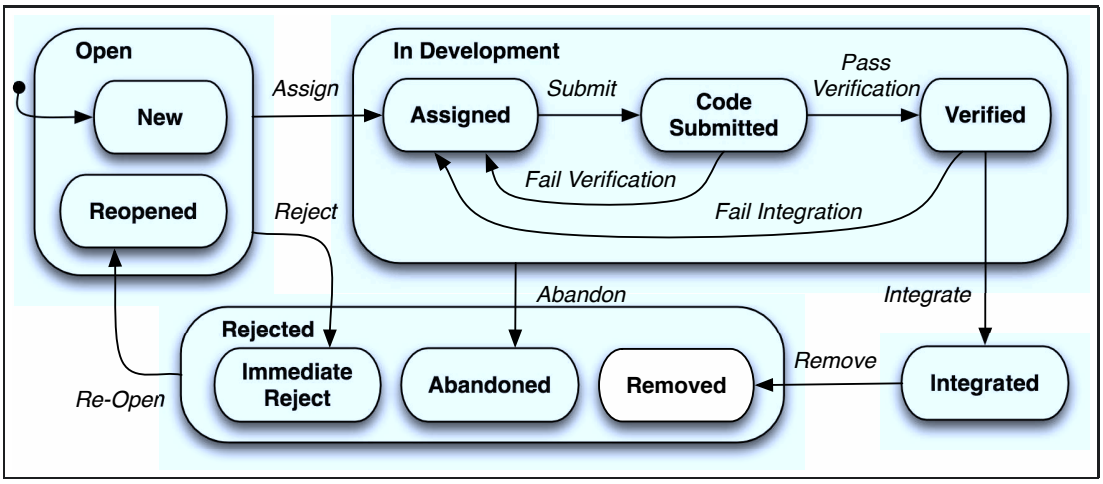



Figure 3.2: Feature Life Cycle

At different stages of its life-cycle, project managers may decide to reject a feature request, this may happen before it is assigned to a developer for implementation, during implementation, or after the feature has been integrated in the product. We use the labels 'Immediate Reject', 'Abandoned', and 'Removed' to distinguish these three cases. A rejected feature request can be reopened, meaning that it is again open to decisions to either reject it or to assign it for implementation.

This model is not one that is followed rigorously or explicitly in the projects we have studied. It is rather a model that helps clarify the different conceptual states of feature requests and define the different failure types that can be associated with them. The state of a feature request in this model can be inferred from meta-data in the Bugzilla issue tracker or similar systems. We now explain this mapping for the KDE project[1] (a project concerned with the development of a unix graphical desktop environment). Each project has slightly different ways of labelling its bug reports, but the general approach is similar. Feature requests in the KDE project correspond to "bug reports" whose severity is labelled as 'enhancement' to distinguish them from bug reports corresponding to system failures. A feature request is in the state 'New', 'Reopened', 'Verified' or 'Assigned' of our model if the status of the bug report has the corresponding value in Bugzilla. A feature request is in the state 'Integrated' if the bug report status is set to 'Resolved' and its resolution type is 'FIXED'. A feature request is in the state 'Rejected' if the bug report status is 'Resolved' and its resolution type is 'WONTFIX', 'INVALID' or 'WORKSFORME'. Our study and analysis of feature requests excludes bug reports marked as 'DUPLICATE' because when this happens, the duplicate request is abandoned and all discussions and decisions continue with the original request.

The example feature request in Appendix A illustrates one possible path through the feature request life-cycle. We have marked in the discussion where changes in the meta-data have occurred that signal a change of state in the life-cycle. The feature request begins life in the state 'New', and progresses to the 'Immediate Reject' state after post #1 when a developer rejects it. After post #12 this decision is reversed and the feature request moves into the 'Assigned' state. In post #19 a code patch uploaded to the feature

[1] `https://bugs.kde.org/page.cgi?id=fields.html`

request moves it into the 'Code Submitted' state. After post #23 the code implementing the feature is verified, moving it into the 'Verified' state. Finally, the feature request is marked as resolved and fixed in its meta-data, signalling the transition into the 'Integrated' state.

## 3.2 Defects in Feature Requests

### 3.2.1 Defect Classifications

A *defect* in this context is an error in the description of a feature request. This may either be a problem with the wording of the description itself, or a problem with the feature that is requested. The feature request in Appendix A illustrates such defects. Post # 1, in which a developer states that the proposed feature is 'Not part of the plan', suggests a defect since it does not contribute to the project goals. Post #12 meanwhile, suggests that the feature request's description is defective as it contains an ambiguity that resulted in a misunderstanding of the proposed feature.

Requirements defect taxonomies can be applied to classify defects in feature requests. A typical example of such a taxonomy is shown in Table 3.1 [van Lamsweerde, 2009]. Many alternative taxonomies have been proposed that vary in their scope and classification of defects [Fenton et al., 2008] [Madachy and Boehm, 2008] [Bell and Thayer, 1976] [Hayes, 2003] [Parnas and Weiss, 1985]. There is no one classification scheme that will fit all domains [Brykczynski, 1999], and selection of a taxonomy will depend on the context in which it is used (e.g. technical knowledge of reviewers, requirements documentation platform, stage of development lifecycle) and the indented outcomes of its usage (e.g. defect removal, quality assessment) [Freimut et al., 2005].

Certain types of defects found in standard requirements documents are not applicable to feature requests (such as forward referencing and remorse), while others that are given little attention in standard requirements taxonomies are prominent. A survey-based study into issues identified by stakeholders in feature requests [Laurent and Cleland-Huang, 2009]

| | |
|---|---|
| Omission | Problem space feature not stated by any RD item e.g., missing objective, requirement, or assumption; unstated response to some input. |
| Contradiction | RD items defining some problem space feature in some incompatible way. |
| Inadequacy | RD item not adequately stating some problem space feature. |
| Ambiguity | RD item allowing some problem space feature to be interpreted in different ways e.g., ambiguous term or assertion. |
| Unmeasurability | RD item stating some problem space feature in a way that cannot be precisely compared with alternative options, or cannot be tested or verified in possible solutions. |
| Noise | RD item yielding no information on any problem space feature. |
| Overspecification | RD item stating some feature not in the problem space but in the solution space. |
| Unfeasibility | RD item that cannot be realistically implemented within the assigned budget and schedules. |
| Unintelligibility | RD item stated in an incomprehensible way for those who need to use it. |
| Poor structuring | RD items not organized according to any sensible and visible structuring rule. |
| Forward reference | RD item making use of problem space features not defined yet. |
| Remorse | RD item stating some problem space feature lately or incidentally. |
| Poor modifiability | RD items whose modification may need to be globally propagated throughout the RD. |
| Opacity | RD item whose rationale, authoring, or dependencies are invisible. |

Table 3.1: Defects in a Requirements Document (RD)

suggests prominent defects to be infeasibility, a lack of satisfaction of stakeholders needs, and conflicts between other features and high-level goals. Defects that apply to text-based collaborative discussion and deliberation [Klein and Iandoli, 2008] can also apply to feature requests, including: flawed argumentation, scattered content, and the 'soapbox problem' (where the opinions of those who speak the most are over represented).

### 3.2.2 An Exploratory Study of Defect Types in the Firefox Project

We conducted an exploratory study to identify the types of defects reported by stakeholders in feature requests. This study was limited to the Firefox project and manually inspected 50 feature requests at random. The sample was taken from feature requests with a full conversation history by selecting those that had been resolved (either integrated into the product or rejected). Requirements defects in feature request management systems are currently not explicitly labelled, and we therefore identified comments in which a stakeholder implied that a requirements defect existed in the feature request's description. Further, we identified whether we considered these defects to be resolved at the end of

the discussions. The study was limited to 50 feature requests, since at this number of experiments no new defect types were being identified. Discussions in the sample ranged from 2 to 240 posts in length, with an average length of 23 posts.

The specific types of defects reported in the experiments, their frequencies, and the number of them that remained unresolved are shown in Table 3.2. The first column shows the defects as we have classified them in the study. These classifications are not taken from a specific existing defect taxonomy, but instead represent descriptions that we felt best abstracted the critiques made by stakeholders in the sample. The definitions of these defect types are open to re-interpretation (an 'Outdated' defect, for example, could be re-expressed as 'Does Not Contribute to Project Goals' defect). While it may be desirable to make classifications more mutually exclusive and concise for a generalised taxonomy of requirements defects in feature requests, we have deliberately left them in their current form to reflect the range of critiques raised by stakeholders in the study. The second column, meanwhile, shows how these defects might be re-classified using an existing requirements defect taxonomy. Where possible we have used concepts from Axel van Lamsweerde's classification scheme [van Lamsweerde, 2009]; for those defects that could not be re-classified in this way alternative taxonomies have been used which are referenced in the table.

There was an average of 1.42 defects identified by stakeholders per feature request and few of these remained unresolved (16%). This suggests that there is an existing practice of reporting and resolving defects in feature request management systems, albeit a non-explicit one.

No requirements taxonomy from those discussed in the previous sub-section can be used to classify all the defect types identified in the sample. This suggests a mis-match between existing defects schemas and a schema which would be of use for a feature request management system. Further, it can be seen that a range of defect types are grouped as an inadequacy defect when using van Lamsweerde's schema, which would therefore result in a loss of information if used. The focus on inadequacies can be explained by the development practice in feature requests: stakeholders suggest features and discuss with their peers whether the feature would make a valuable addition to the software product.

| Labelled Defect Type | Alternative Classification | Reported | Unresolved |
|---|---|---|---|
| Ambiguous Feature Description | Ambiguity* | 6 | 2 |
| Incomplete Feature Description | Omission* | 2 | 1 |
| Infeasible to Implement | Infeasible* | 4 | 0 |
| Conflict with Existing Feature | Contradiction* | 9 | 3 |
| Feature Description is Not Atomic | Poor Structuring* | 2 | 0 |
| Stakeholders' Needs Not Understood | Inadequacy* | 7 | 3 |
| Does Not Contribute to Project Goals | Inadequacy* | 4 | 0 |
| Low Value (Compared to Cost) | Inadequacy* | 9 | 0 |
| Incorrect Assumptions | Inadequacy* | 1 | 0 |
| Outdated (Feature No Longer Useful) | Inadequacy* | 16 | 0 |
| Feature is Already Implemented | Inadequacy* | 4 | 0 |
| Duplicate of Other Feature Request | Duplicate** | 2 | 0 |
| Feature Could Provide More Value | Better Design Possible*** | 5 | 2 |
| Total | | 71 | 11 |

** [van Lamsweerde, 2009] ** [Hayes, 2003] *** [Bell and Thayer, 1976]*

Table 3.2: Frequencies of Defects Labelled in 50 Firefox Feature Requests

During the course of this discussion those against the feature typically point out all the inadequacies associated with it. A range of defect classifications that abstract the types of inadequacies that a feature can posses could therefore potentially increase the utility of a feature request defect taxonomy.

The main purpose of this study was to explore the types of defects currently reported by stakeholders in feature requests. The study is not a large-scale empirical study into a defect taxonomy for feature requests. This would require manual identification of defects in a large sample across projects, which was not possible during the time-frame of this thesis. Further, the study is a reflection of the requirements defects currently identified by stakeholders, and does not give an indication of unreported defects. We suspect that the high numbers of inadequacies identified are a reflection of the development practice in feature requests as discussed above, and that other types of defects (such as opacity and omission) go unreported.

## 3.3 Failures in Feature Requests

### 3.3.1 Failure Classifications

A *failure* in this context, in contrast to a defect, is an observed error in a feature request's development process or implementation. A failure may be caused by one or more defects, and a defect need not necessarily give rise to a failure.

We have identified a set of five failures that can be traced from a feature request's progress though its life-cycle (see Figure 3.2), meaning that a large body of feature requests can be automatically scanned for their presence. As with the classification of defects, the utility of a failure classification scheme depends on the context of its usage and the intended outcomes. The fact that these failures can be automatically detected makes them candidates for data mining and machine learning methods. The five failures are defined as follows:

**Product Failure** - A feature request has a product failure if it is in the 'Integrated' state and has at least one confirmed bug report associated with it. This definition does not cover unreported failures so might be better understood as 'reported product failure'.

Feature request #451995[2] of the Thunderbird project is an example of this failure. An archive for auto-saved emails is implemented and integrated into the product after which two bugs, #473439 and #474848, are reported as having resulted from the introduction of the feature.

**Abandoned Development** - A feature request is abandoned if it was once assigned for implementation and the development effort has been cancelled before the feature is integrated into the product. These correspond to feature requests that are in the 'Abandoned' state in the model of Figure 3.2. A subset of feature requests exhibiting this failure that have been abandoned after code has been submitted (i.e. those that were in state 'Code Submitted' or 'Verified' when rejected). We refer to such failures as '**Abandoned Implementation**'. The studies in this thesis focus on abandoned implementation as opposed to the more general case because effort has been spent developing implementations

[2] `https://bugzilla.mozilla.org/show_id=451995`

making this failure more costly. They could, however, be extended to cover all abandoned development failures.

Feature request #34868[3] of the Apache project is an example of an abandoned implementation failure, in which a change to a server authentication service is suggested, assigned for implementation and has two code components and an example developed for it. After this it is discovered that it is not possible to reach a state when running Apache where this feature would be useful and it is rejected.

**Rejection Reversal** - A feature request has a rejection reversal failure if it was once rejected before eventually being integrated into the product. Rejecting a feature request is an explicit decision to remove it from the backlog of feature requests considered for implementation. This is different from a decision not to assign a feature request for implementation at the moment but leaving it in the backlog. We view the initial decision to wrongly remove a feature request from the backlog as the fault that causes this type of failure. This fault delays the introduction of features of value to stakeholders, can upset stakeholders, and cost them time and effort to argue for the feature request to be reopened.

Feature request #171702[4] of the Firefox project (our running example from Appendix A) is an example of this failure. A developer rejects a feature request in the first post after it is suggested by simply stating that it is "not part of the plan", without fully understanding the feature request's full meaning or eliciting stakeholders' needs. Subsequently, stakeholders become frustrated and effort is spent arguing the feature's merits. The feature is eventually re-opened, implemented and integrated into the product.

**Stalled Development** - A feature request has a stalled development failure when it remains in the 'assigned' state and no code has been submitted for more than one year. The duration of one year is arbitrary; we have chosen it here because it corresponds to a duration within which we would expect code to have been developed for the feature requests studied in this thesis. Our studies could easily be repeated with shorter deadlines for this failure type. If a feature request management system contains information on the

[3] `https://issues.apache.org/bugzilla/show_id=34868`
[4] `https://bugzilla.mozilla.org/show_id=171702`

estimated development time for each feature request it could be used to detect 'late development' failures. If the period of inactivity after which we consider stalled development failures to occur were shortened we would expect the rate of this failure to fall and the costs associated with it to increase, and vice versa.

Feature request #49970[5] of the KDE project is an example of this failure. A fullscreen mode option is suggested and assigned for development in late 2005 and does not get fixed until mid 2007. In comment number 17 a developer states that the assigned developer had not been working for some time and apologises that the feature would not make it into the planned release.

**Removed Feature** - A feature request is removed if it has been rejected after having been integrated into the product. This corresponds to the 'Removed' state in Figure 3.2. Feature requests rejected in this way signify that a decision was made to discard the feature before shipping the next release of the product. This is a failure because it is caused by the need to remove a feature that introduces undesirable behaviours for stakeholders. Such failures may be caused by insufficient upfront analysis of the feature request before its development.

Feature request #171465[6] of the Netbeans project is an example of this failure. A feature is implemented and integrated into the project, after which it is disabled due to a lack of support in the Netbeans framework for the feature. Finally, a developer states that the feature "does not fit into our current strategy".

### 3.3.2 An Empirical Study of Failures in Seven Projects

We conducted a study assessing the frequency of failures in seven large-scale open-source projects. We developed a scraper in PHP that automatically traced the life-cycle of all feature requests in these projects and checked for failures against the rules defining them in Section 3.3.1. The purpose of this study was to assess the degree to which failures can be automatically identified and determine their impact on projects. These projects were

[5] http://bugs.kde.org/show_id=49970
[6] http://netbeans.org/bugzilla/show_id=171465

the Apache web server[7], the Eclipse development environment[8], Firefox web browser[9], the KDE operating system[10], the Netbeans development environment[11], Thunderbird email client[12], and the Wikimedia content management system[13]. The projects were all large in size, ranging from 5,000 to 50,000 feature requests.

Table 3.3 shows occurrences of the five types of failure detected in the 7 projects. Failures were automatically identified from the meta-data of all feature requests within the projects from their start date until the time of writing. The first four rows show the number of years for which the feature request management systems have been in use, the total number of feature requests that have been created in that period, and how many of these have been assigned or rejected at some stage in their life-cycle. The following five rows show the rate at which failures occur in terms of the *failure density* - the percentage of failures among all feature requests to which the particular failure type applies. For product failure, abandoned implementation, stalled development, and removed feature, this corresponds to the number of failures divided by the number of assigned feature requests; for rejection reversal, it corresponds to the number of failures divided by the number of rejected features. In the three cases where the failure density is not shown no failures were automatically identified. The failure densities are visualised in Figure 3.3.

The majority of the failure types have relatively high densities, occurring in abundance within the seven projects. It is important to note, however, that the process of identifying these failures is not infallible as it relies on developers updating the meta-data of feature requests consistently. We now discuss possible inaccuracies in the automated failure detection process.

[7] `https://issues.apache.org/bugzilla/`
[8] `https://bugs.eclipse.org/bugs/`
[9] `https://bugzilla.mozilla.org/`
[10] `https://bugs.kde.org/`
[11] `http://netbeans.org/bugzilla/`
[12] `https://bugzilla.mozilla.org/`
[13] `http://bugzilla.wikimedia.org/`

| | Apache | Eclipse | Firefox | KDE | Netbeans | Thunderbird | Wikimedia |
|---|---|---|---|---|---|---|---|
| Duration | 10.8yrs | 9.2yrs | 8.3yrs | 11.1yrs | 10.3yrs | 11.8yrs | 7.2yrs |
| Assigned Feature Requests | 354 | 7,564 | 382 | 1,800 | 2,152 | 199 | 824 |
| Rejected Feature Requests | 1,444 | 12,247 | 1,975 | 11,156 | 5,008 | 791 | 3,129 |
| Total Feature Requests | 5,242 | 46,247 | 6,379 | 55,349 | 24,167 | 5,150 | 13,797 |
| Product Failure | - | 0.1% | 21.7% | - | 2.3% | 14.1% | 0.2% |
| Abandoned Implementation | 4.0% | 0.8% | 6.1% | 1.3% | 1.4% | 3.5% | 4.5% |
| Rejection Reversal | 5.9% | 2.5% | 13.1% | 4.3% | 4.4% | 2.3% | 9.6% |
| Stalled Development | 26.1% | 60.6% | 12.9% | 26.0% | 34.3% | 14.1% | 15.6% |
| Removed Feature | - | 0.1% | 0.8% | 0.6% | 2.3% | 0.5% | 1.7% |

Table 3.3: Feature Requests and Failures in Large-Scale Projects

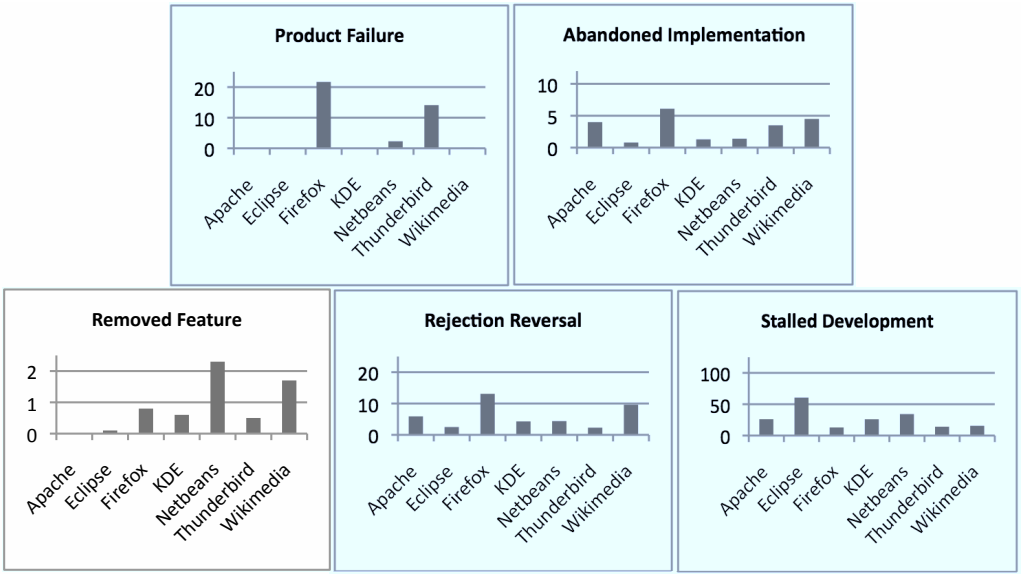



Figure 3.3: Failure Densities (%)

### Threats to Automated Failure Detection

Product failures will not have been caught if developers do not place a trace between feature requests and any bugs in the product they give rise to. In the KDE and Apache systems, for example, the user interface hides the means of placing such traces. The Firefox project, meanwhile, has a culture of maintaining the consistency of these links. While many product failures were not caught using our approach, we found that a sample of 100 failures of this type over the 7 projects corresponded to cases that were clearly instances of this failure.

Automated detection of abandoned implementation failures relies on developers submitting code they have developed to the feature request. In the Eclipse project, for example, we found that there was a culture of working on code outside of the feature request management system, and therefore fewer failures of this type could be detected. In the other projects, meanwhile, the feature request management system was the primary means of collaborating on code, and a higher failure density for this failure was detected correspondingly.

Rejection reversal failures were caught consistently in these projects since updating a feature request's meta-data is the only means by which a developer can mark it as having been rejected. Features that eventually make it into the final software product will invariably also have their meta-data updated accordingly. One may argue whether all the instances identified are necessarily failures; rejecting the feature request may have been the right decision at the time it was made and the decision to reopen it may be due to new circumstances. Attention should be given to this caveat, and in the projects studied we sampled 100 feature requests containing this failure and found that in all cases the initial rejection of the feature request appeared to be premature and wrong at the time it was made.

Accurate capture of stalled development failures depends heavily on whether code developed is submitted to the feature request, and whether features that have been integrated into the product have their meta-data updated in a timely fashion. We found this to be

the general case in all the projects apart from Eclipse, where most discussions and code development took place outside of the feature request management system through mailing lists, code repositories and IRC channels. Further, we found the Eclipse project did not have a culture of aiming to integrate features that have been assigned for development - a consequence of the project's loose reliance on the feature request management system for communicating and collaborating on feature development. Feature requests identified as a failure of this type in Eclipse might therefore not be considered as severe as in the other projects.

Identification of removed feature failures is dependent on features removed from the software product having their meta-data updated to reflect the change. We found many examples across the projects studied where this was not done; once a feature request has been implemented developers tend to forget about the thread representing it, and do not update its status when it is removed from the product. Our figures are therefore likely to underestimate the true numbers of removed features. However, even if not all removed feature failures are detected and their density is low (between 0.1 and 2.3% in our studies), detecting and addressing them is of value to a project because of their high costs (wasted effort to fully implement, integrate, and then remove the feature).

One could question whether all removed features are really failures; integrating the feature into the product may have been a good decision at the time it was made, and the feature may have been removed later only because circumstances have changed. We have inspected all 86 instances of removed feature failures detected in the 7 projects, and we found that they do correspond to cases where a feature was removed because it conflicted with another feature, stakeholder needs, or the high level goals of the project. In all cases, the feature was removed shortly after it was integrated in the product (within a month). As explained above, features that are removed long after they have been introduced in the product tend to not have the status of their original feature request updated. If this were not the case we might have found instances of removed feature that does not correspond to failures. To avoid including such good cases of removed feature, we could refine our rule for detecting removed feature failures by including only features that have been removed shortly after they have been integrated, for example within 30 days.

An analysis of a sample of approximately 100 automatically identified instances of each failure type across the 7 projects confirmed that they correspond to failures as we have defined them. The exception to this was stalled development failures in the Eclipse project for the reasons discussed above. Inconsistent updates in the meta-data of feature requests do lead to many failures not being caught using our approach; we therefore expect that the failure densities reported may underestimate the true rate of failure in the projects studied.

### Exclusion of Eclipse from Further Studies

The results of the failure study have revealed that not all feature requests identified as having a stalled development failure in the Eclipse project corresponded to real failures as we have defined them. We frequently observed feature requests that were assigned for implementation and left open without subsequent activity. These feature requests did not reflect stalled development failures, but instead developers not updating the meta-data of feature requests that had either been rejected or integrated into the product. We observed that this was not the case in the other 6 projects, and suspect this is since they have a strong culture of top down management of the body of feature requests while in Eclipse developers are left to manage feature requests themselves. This poses a major threat to validity if failure prediction models were to be constructed for the Eclipse project using the approach described in Chapter 5, since prediction models would trained on data containing falsely identified failures. For this reason we have excluded the Eclipse project from the failure prediction experiments in Section 5.5. Consideration should be given to whether all failures identified in the historical data do correspond to real failures when using the early failure prediction approach (for example, by using a sampled check of 100 feature requests as we have done here).

A further reason for not including the Eclipse project in the experiments in Section 5.5 was the low rates of the other four types of failure caught in the project, and low volumes of requirements discussion that we observed in the feature requests. We suspect this is also due to a lack of top down management of the body of feature requests, which

had led to developers inconsistently updating meta-data and low levels of engagement from stakeholders. We did attempt to generate prediction models for these four types of failure, but could not generate any that performed better than the baselines. We would not, therefore, envision that the early failure prediction approach would perform well in projects with low failure densities and sparse discussion data on which to train prediction models.

## 3.4 Related Work

Requirements defects have been studied in many different contexts, such as goal oriented models [van Lamsweerde, 2001], generic requirements documents [van Lamsweerde, 2009], and requirements documents for specific projects (such as, for example, a NASA validation and verification project [Hayes, 2003]). There is a body of research comparing the effectiveness of alternative methods for requirements defect review [Brykczynski, 1999] [Macdonald and Miller, 1999] [Parnas and Weiss, 1985] [Porter et al., 1995]. The discussion and study of requirements defects we present here, however, is specific to feature requests.

Failures in feature request management systems have been well documented and studied at the code level [Wolf et al., 2009] [Nagappan et al., 2006]. The failures presented here, however, cover those that occur at the process level of feature request development. Definitions for generic process failures in software projects have been made [Schmidt et al., 2001], but those we present are specific to feature requests and can be automatically traced from their meta-data. This latter property allows for the application of data mining and machine learning methods (such as the early failure prediction framework presented in this thesis).

## 3.5 Chapter Summary

In this Chapter, we explored and studied the context of requirements defects and failures in feature requests. An exploratory study of the types of defects reported by stakeholders

in Firefox feature requests shows that they differ from those of standard requirements documents. We have defined five types of process and product failures that can be automatically identified from the historical data of a body of feature requests. A study of these failures in seven large-scale projects shows that they occur at a high rate and incur significant costs.

Chapters 4 and 5 build upon this work to provide answers to the research challenges set out in Section 1.2. Chapter 4 proposes an argumentation framework grounded on defect identification and resolution that aims to encourage stakeholders to express their needs more clearly, and help developers to understand these needs. Chapter 5, meanwhile, proposes an approach for automatically generating early failure prediction models that can be used to localise, direct, and estimate the value of upfront requirements analysis.

# 4 Structuring Feature Request Discussions

*In this Chapter we present a tool-supported framework as response to a key challenge in feature request management systems: "is it possible to structure feature request discussions so that stakeholders' needs can be expressed and understood more effectively without significantly altering existing communication practices?". The defect-oriented argumentation framework, DoArgue, extends the capabilities of existing feature request management systems: allowing stakeholders to explicitly label comments related to the decision on whether to reject a feature or assign it for implementation. Structuring discussions in this way may encourage stakeholders to fully explore their needs relating to whether a feature should be developed, while developers can use a visual model of labelled comments to better understand these needs.*

*We begin this Chapter by describing the framework, how it is used, and how it could help make stakeholders' needs better understood in feature request management systems. We then describe a tool implementation of the framework. Finally, we report the results of a preliminary case study evaluation of the framework on the Firefox project. The study suggests that the framework captures the key concepts in a discussion of whether to implement a feature, and that comparatively little additional effort is required to use the framework.*

## 4.1 DoArgue: Defect-Oriented Argumentation Framework

In our experience with the seven feature request management systems studied in Section 3.3.2 we found that stakeholders' discussions on whether to implement a feature were centred around the identification and resolution of defects. Defects reported by stakeholders are used as grounds for rejecting a feature request, and an effort is made to resolve reported defects before implementing a feature (by altering the feature requests' description or justifying the unimportance of outstanding defects).

The running example in Appendix A illustrates an instance of the focus on defects in de-

ciding whether to implement a feature. In post #1 a developer rejects the newly suggested feature request, basing this rejection on an implied defect of type 'Does not contribute to project goals' by stating that the feature is 'Not part of the plan'. This defect, and a later 'Ambiguous' defect implied in post #12 are resolved by an alteration to the feature request's description (shown in Figure A.6) before the feature is implemented and integrated into the product.

We therefore propose a defect-oriented argumentation framework, DoArgue, that builds upon design rationale theory (see Section 2.1.2) to encapsulate the key concepts of discussions on whether to implement a feature from the perspective of defect identification and resolution. Design rationale frameworks specify models for structuring multi-party discussions on exploring, arguing for, and selecting alternative design decisions in response to real world problems. The basic IBIS framework [Conklin and Begeman, 1988] which we have adapted is shown in Figure 4.1. Stakeholders identify real world issues, and suggest design alternatives that could potentially resolve them. The IBIS framework allows for stakeholders to argue for and against these alternatives. Decision-makers can then use this information to make better informed decisions when resolving real world problems.

Mapping a design space using a design rationale framework has been shown to benefit online distributed discussions on making complex design decisions when contrasted with forum-style discussions [Ontañón and Plaza, 2008] [Klein and Iandoli, 2008]. These benefits take the form of reducing noise, flawed argumentation, scattered content, poor discussion structure and the soapbox problem (where a participant that speaks more than others gets more weight given to their opinions).

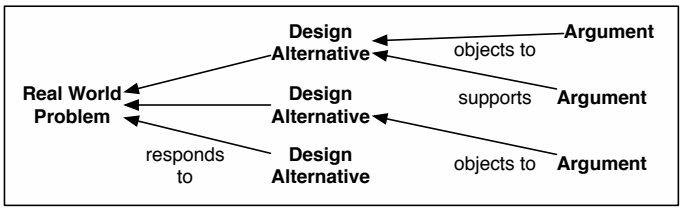



Figure 4.1: Basic IBIS design Rationale Framework

In the DoArgue framework we have developed, shown in Figure 4.2, a feature request defect is conceptualised as a specialised case of a real world problem. The proposed

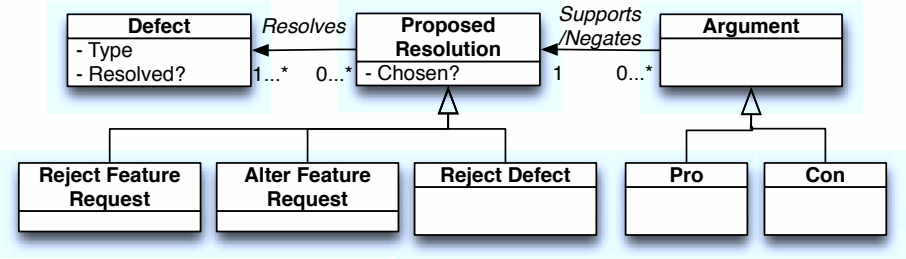



Figure 4.2: DoArgue Framework

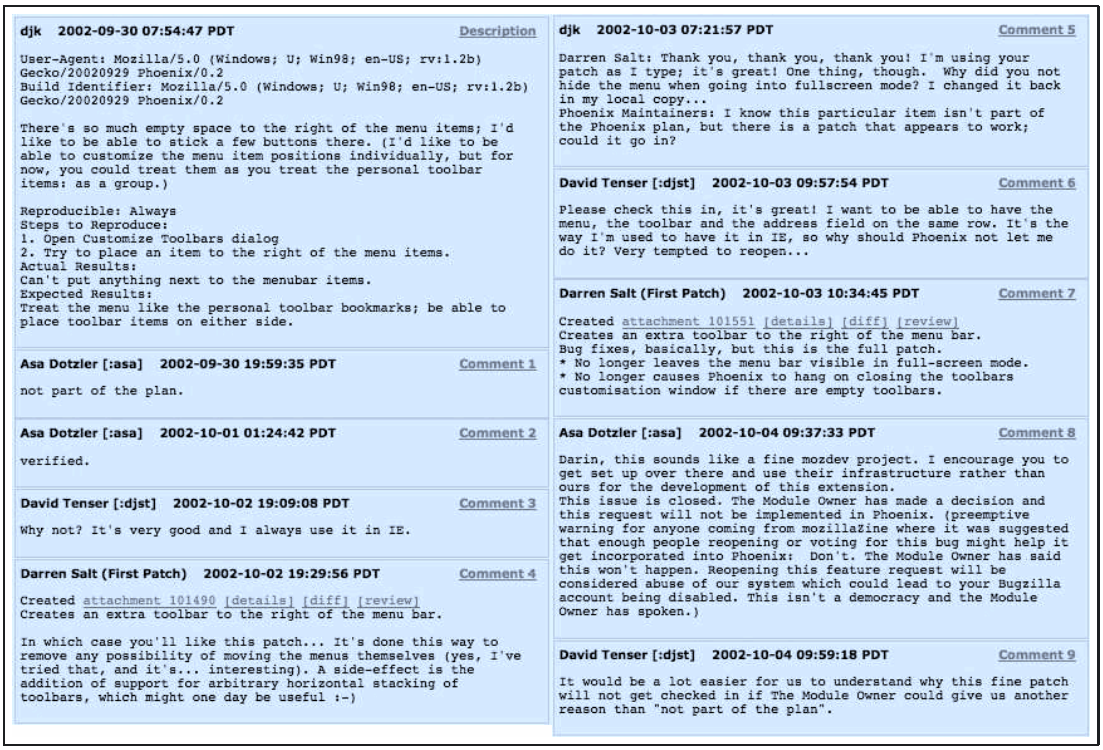
resolutions to a defect are specialised cases of design alternatives. The concepts of pro and con arguments, meanwhile, encapsulate the fact that in feature requests stakeholders argue for and against proposed resolutions. At any point in time a developer can weigh up a discussion and action one of the proposed resolutions; either rejecting the feature request, altering its description, or explicitly deciding to reject the defect.

Figure 4.4 shows how the framework conceptualises the feature request discussion in Figure 4.3 (taken from the first nine posts from our running example in Appendix A). The model is a simple, effective way of visualising the key elements of the discussion relating to whether the requested feature should be implemented. The model and the process by which it is constructed are fully described Section 4.1.1, and the benefits using the framework to model discussions in this way are discussed in Section 4.1.2.

### 4.1.1 Structuring a Feature Request Discussion

DoArgue has been designed to supplement existing modes of collaboration in feature requests, rather than replace them. Stakeholders submit posts to feature requests as they currently do, but annotate statements they make that express concepts from the framework. At any point in time a visual model of the arguments can be generated, such as that shown in Figure 4.4. Labelling of statements is not enforced, and developers can label statements in existing posts if stakeholders do not do this themselves. These design decisions were taken to minimise the underlying changes to current collaborative practices in feature request management systems and thereby facilitate DoArgue's adoption.

We now illustrate the process by which stakeholders can annotate their discussions using



**djk 2002-09-30 07:54:47 PDT** Description

User-Agent: Mozilla/5.0 (Windows; U; Win98; en-US; rv:1.2b) Gecko/20020929 Phoenix/0.2
Build Identifier: Mozilla/5.0 (Windows; U; Win98; en-US; rv:1.2b) Gecko/20020929 Phoenix/0.2

There's so much empty space to the right of the menu items; I'd like to be able to stick a few buttons there. (I'd like to be able to customize the menu item positions individually, but for now, you could treat them as you treat the personal toolbar items: as a group.)

Reproducible: Always
Steps to Reproduce:
1. Open Customize Toolbars dialog
2. Try to place an item to the right of the menu items.
Actual Results:
Can't put anything next to the menubar items.
Expected Results:
Treat the menu like the personal toolbar bookmarks; be able to place toolbar items on either side.

**Asa Dotzler [:asa] 2002-09-30 19:59:35 PDT** Comment 1

not part of the plan.

**Asa Dotzler [:asa] 2002-10-01 01:24:42 PDT** Comment 2

verified.

**David Tenser [:djst] 2002-10-02 19:09:08 PDT** Comment 3

Why not? It's very good and I always use it in IE.

**Darren Salt (First Patch) 2002-10-02 19:29:56 PDT** Comment 4

Created attachment 101490 [details] [diff] [review]
Creates an extra toolbar to the right of the menu bar.

In which case you'll like this patch... It's done this way to remove any possibility of moving the menus themselves (yes, I've tried that, and it's... interesting). A side-effect is the addition of support for arbitrary horizontal stacking of toolbars, which might one day be useful :-)

**djk 2002-10-03 07:21:57 PDT** Comment 5

Darren Salt: Thank you, thank you, thank you! I'm using your patch as I type; it's great! One thing, though. Why did you not hide the menu when going into fullscreen mode? I changed it back in my local copy...
Phoenix Maintainers: I know this particular item isn't part of the Phoenix plan, but there is a patch that appears to work; could it go in?

**David Tenser [:djst] 2002-10-03 09:57:54 PDT** Comment 6

Please check this in, it's great! I want to be able to have the menu, the toolbar and the address field on the same row. It's the way I'm used to have it in IE, so why should Phoenix not let me do it? Very tempted to reopen...

**Darren Salt (First Patch) 2002-10-03 10:34:45 PDT** Comment 7

Created attachment 101551 [details] [diff] [review]
Creates an extra toolbar to the right of the menu bar.
Bug fixes, basically, but this is the full patch.
* No longer leaves the menu bar visible in full-screen mode.
* No longer causes Phoenix to hang on closing the toolbars customisation window if there are empty toolbars.

**Asa Dotzler [:asa] 2002-10-04 09:37:33 PDT** Comment 8

Darin, this sounds like a fine mozdev project. I encourage you to get set up over there and use their infrastructure rather than ours for the development of this extension.
This issue is closed. The Module Owner has made a decision and this request will not be implemented in Phoenix. (preemptive warning for anyone coming from mozillaZine where it was suggested that enough people reopening or voting for this bug might help it get incorporated into Phoenix: Don't. The Module Owner has said this won't happen. Reopening this feature request will be considered abuse of our system which could lead to your Bugzilla account being disabled. This isn't a democracy and the Module Owner has spoken.)

**David Tenser [:djst] 2002-10-04 09:59:18 PDT** Comment 9

It would be a lot easier for us to understand why this fine patch will not get checked in if The Module Owner could give us another reason than "not part of the plan".

Figure 4.3: Firefox FR #171702 after post #9

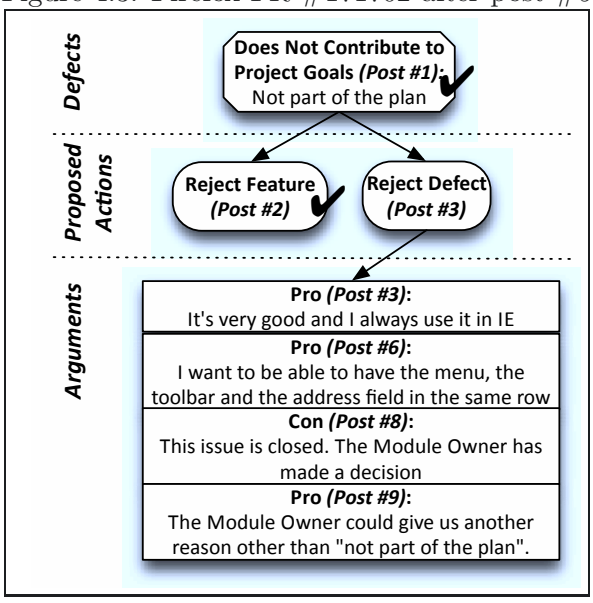



Figure 4.4: DoArgue Visualisation: Firefox FR #171702 after post #9

the running example from Appendix A. We have labelled the comments made in the discussion using the DoArgue framework 'a-posteriori', but a similar model would result if developers and stakeholders had access to the framework during the discussion and annotated their comments in real time. The process by which the model in Figure 4.4 is constructed from the comments in Figure 4.3 is as follows:

- In comment #1 Asa Dotzler states that the newly suggested feature request is 'not part of the plan'. We therefore labelled this post as identifying a defect of type 'Does not contribute to project goals'. The choice of defect type is subjective, and is purely a representation of that which the user labelling a defect believes to most closely reflect the requirements defect in question. A discussion of appropriate defect classification taxonomies for feature requests is given in Section 3.2.

- In post #2 Asa rejects the feature request, and correspondingly we have labelled this post with a 'Reject Feature' proposed resolution and selected it to resolve the defect. The selection of this resolution and the fact that the defect has been resolved are visually represented by the two ticks on the corresponding elements of the diagram. These ticks could also optionally display information on the point of time (comment number) in the discussion when they were added.

- In post #3 David Tenser questions Asa's decision to reject the feature and provides arguments against it. We have therefore labelled this post as putting forward a new 'Ignore Defect' proposed resolution and an associated 'Pro' argument towards it. We have allowed for alternative resolutions to be proposed after a defect has been resolved in the framework since we commonly observed such an event occurring in the projects studied in Chapter 3.

- Posts #6, #8 and #9 contain arguments for and against rejecting the defect, and have been labelled as such correspondingly.

Figure 4.5 shows a visual representation of the feature request in the running example after post #21 (see Appendix A for the full discussion). The developer David Hyatt realises in post #12 that there is an ambiguity in the description of the feature request

that has lead to a misunderstanding of the change it describes; he therefore changes the feature's description and re-opens it. Correspondingly, we have labelled this post with an ambiguous defect, an associated proposed resolution to change the feature description, and have selected the resolution to resolve the defect. Further, there is an implicit decision to reject the defect raised in post #1, which we have reflected in the model. The tick in grey on the 'Reject Feature' proposed resolution shows that this resolution had previously been chosen, but reversed in favour of another. Two more defects along with their associated resolutions and arguments are labelled in posts #16 and #21, both of which are over concerns that the feature request could provide more value to the project. It can be seen in the model that these two defects are yet to be resolved by the absence of ticks. These latter defects are arguably more design defects than requirements defects; requirements activity and high level design activity in feature requests are typically not clearly separated, and the framework can be used to review defects at both these levels.

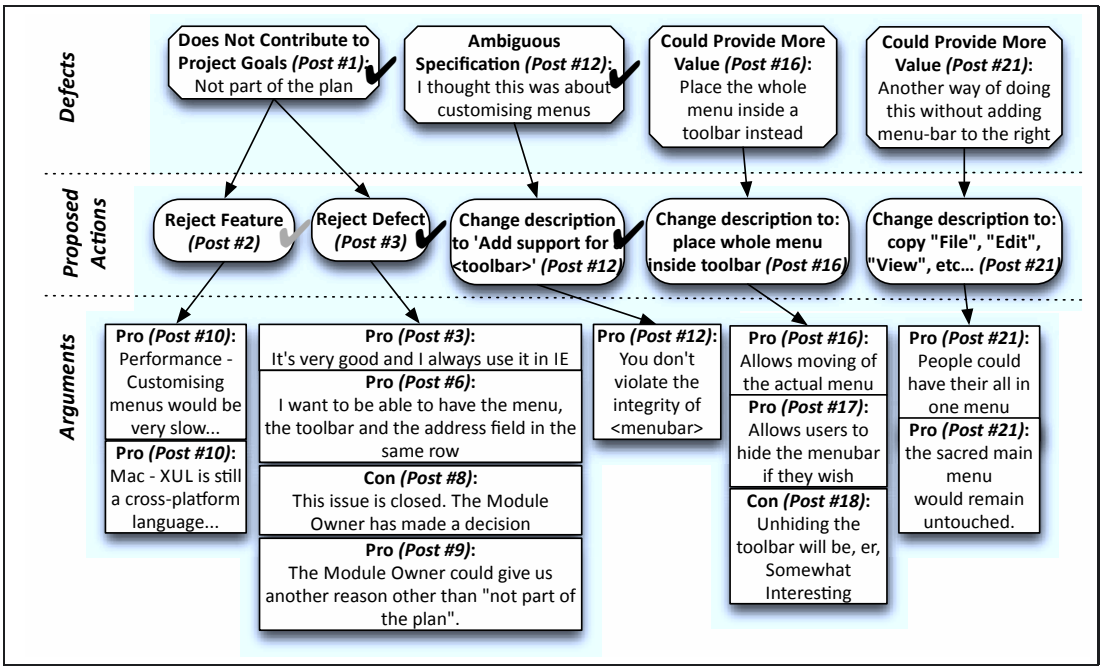



Figure 4.5: DoArgue Visualisation: Firefox FR #171702 after post #21

### 4.1.2 Understanding Stakeholders' Needs

DoArgue has been developed to address the key challenge in feature request management systems of making stakeholders' needs better understood. The benefits of using a design rationale framework to conceptualise and visualise discussions have been extensively evaluated [Shum, 1996], and the framework aims to translate these to the context of feature requests:

- ***Clarity of Discussion Structure*** - A thread with a moderate number of posts can be hard to read: the structure of arguments is unclear, information is scattered, noise is introduced by comments not relevant to the discussion, and the entire thread needs to be read in order to gain an understanding of it contents. DoArgue provides an alternative view of this discussion organised in terms of stakeholders' arguments for and against implementing a feature. Hidden decisions are made explicit (such as the rejection of the first defect in the running example). This could it make it easier for developers to understand the complexities of stakeholders' needs across a body of feature requests. Further, the barrier to entry for a feature request discussion is lowered making it easier for stakeholders to join discussions and understand the key arguments within them.

- ***Expression and Elicitation of Needs*** - Structuring discussions with elements from a design rationale framework has been shown to encourage stakeholders to fully explore a design space. This can lead to preferable outcomes when making design decisions by improving argumentation and facilitating the exploration and understanding of alternative design choices.

- ***Rationale Record*** - The rationale behind decisions taken at the requirements stage of a feature request's life cycle are concisely recorded. This allows developers to better understand historical design decisions.

## 4.2 Tool Implementation

We have developed a web-based tool that implements the DoArgue framework. A screenshot of the tool can be seen Figure 4.6. It allows a user to extract a body of feature requests from a specified Bugzilla feature request management system, and displays their content in the upper section of the tool interface. A user can highlight entire comments or statements within comments and annotate them with elements from the DoArgue framework. The lower section of the tool interface shows the visual representation of the DoArgue model.

The tool was developed to test the applicability of the framework in a real feature request management system and to conduct the evaluation in the Section that follows. It does not currently integrate into a feature request management system and needs to be run in a separate window; future development should allow access to the tool while contributing to a feature request so that stakeholders can annotate their comments in real time.

## 4.3 Case-Study Evaluation

We have conducted a qualitative case study-based evaluation of the DoArgue framework by using it to annotate real feature requests in the Firefox project. The questions we wished to answer were the following:

1. How much additional effort does it take to annotate a discussion using DoArgue?
2. Does DoArgue capture the key concepts of discussions on whether to implement a feature?

These experiments constitute a preliminary self-evaluation of the DoArgue framework. While they do show that the framework could be applied to structure existing feature request discussions, its true benefits would be hard to demonstrate. We would have ideally liked to evaluate whether the benefits listed at the end of Section 4.1.2 lead to developers

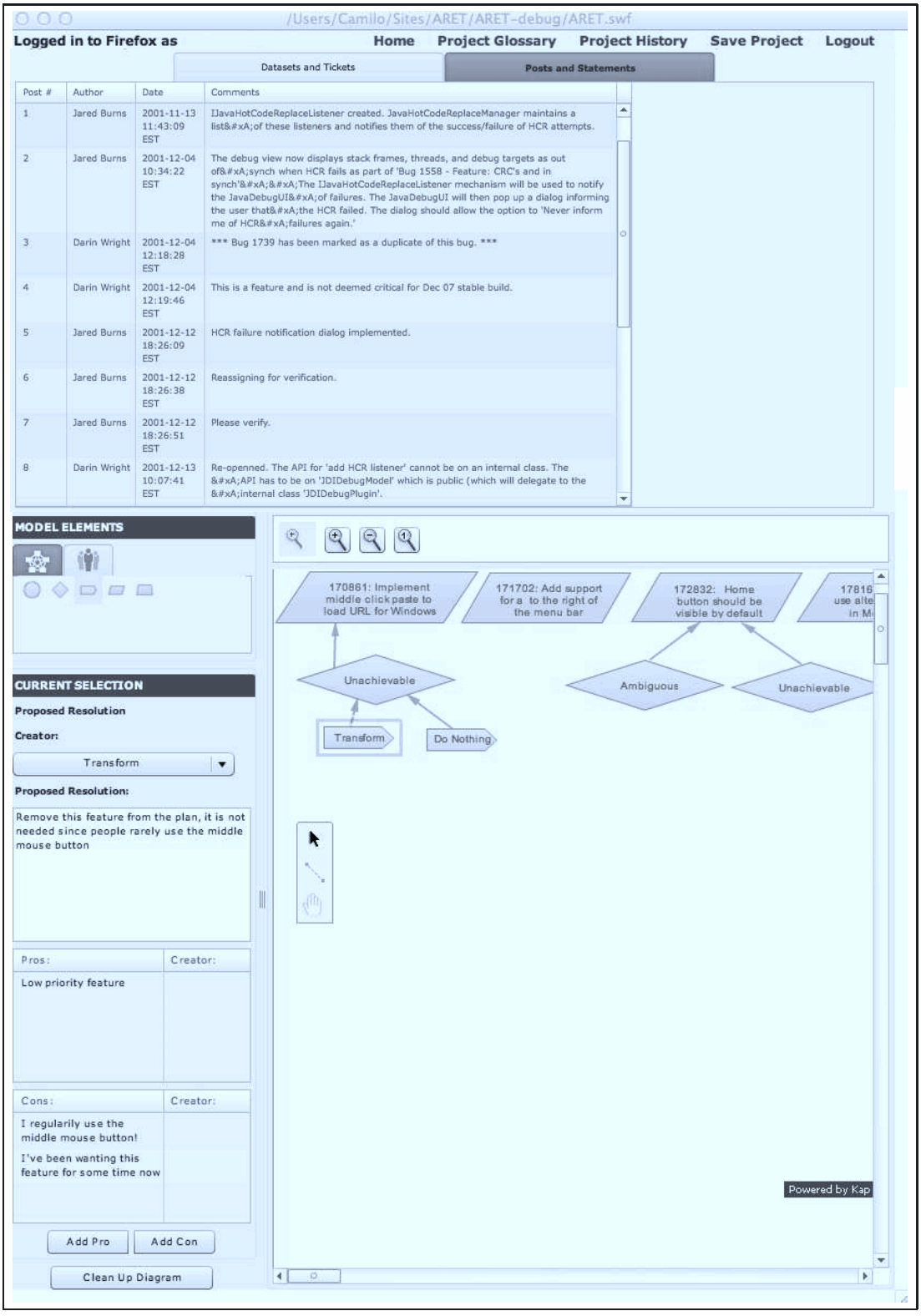



Figure 4.6: DoArgue Tool Interface

gaining a better understanding of stakeholder needs. User adoption and engagement is also a common evaluation criteria for argumentation tools [Shum, 1996]. This type of evaluation would require deploying the framework in a real project and monitoring its usage, which was not feasible in the time frame of this thesis. This work, however, informed the development of an alternative approach to decision making in feature requests, reported in Chapter 5, which does not require stakeholders to annotate discussions.

### 4.3.1 Experimental Setup

We randomly selected 50 feature requests from the Firefox project and annotated them using the DoArgue framework. The time taken to label each feature request was measured, along with a qualitative assessment of how easily the the discussion could be conceptually modelled. The full experiment is available online[1].

The feature requests used in this evaluation are the same feature requests from the study in Section 3.2; the models we constructed during this evaluation were used to identify the types of defects reported by stakeholders. We presented the types of defects reported by stakeholders earlier in Section 3.2 since they do not directly relate to the evaluation on the DoArgue framework, but instead provide context on defects in feature request managements systems.

### 4.3.2 Results

The scatter-plot in Figure 4.7 helps us answer the first question by showing the times spent labelling each feature request discussion set against the number of posts in these discussions. The time taken to label a discussion was less than 10 minutes for the majority of feature requests (92%), and the average time per post was 17 seconds. These additional times are negligible when compared to any reasonable estimation for the total time put into a discussion, or the time taken to formulate a post and submit it. Further, we found that most of the labelling process was taken up by reading and understanding posts rather

[1] `http://sre-research.cs.ucl.ac.uk/DoArgue/ThesisExperiments`

than the actual annotation process itself (illustrated by the linearity of the scatter-plot).

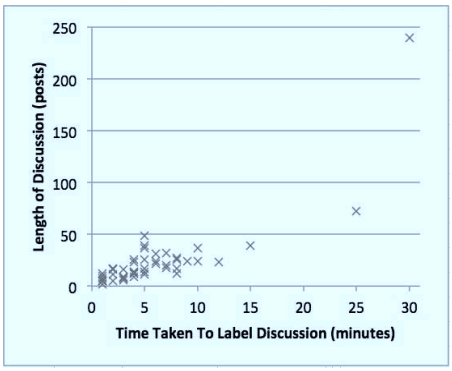



Figure 4.7: Time Spent Labelling

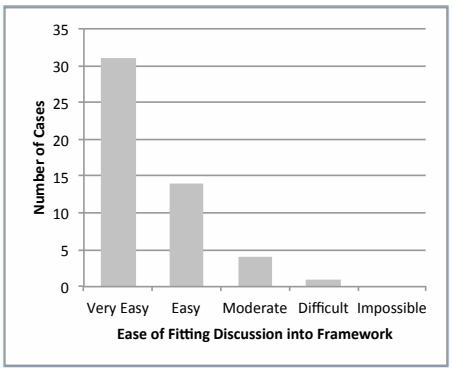



Figure 4.8: Ease of Labelling

For each discussion we also rated how easily we could fit its contents into the DoArgue framework on a 5 point scale from 'very easy' to 'impossible'. A bar chart showing the frequencies for each of these classifications is shown in Figure 4.8. The DoArgue model for the feature request that was 'difficult' is shown in Figure 4.9, in which difficulty arose from identifying the multiple links between proposed resolutions and defects. The resulting visualisation of arguments is also difficult to understand. This feature request, however, was an exception and the majority of the discussions (90%) were found to be 'easy' or 'very easy' to model in DoArgue. The overall low levels of difficulty and time spent labelling discussions demonstrate that a comparatively small amount of effort is required to extend current practices in feature request management systems with the DoArgue framework.

To answer the second question a qualitative assessment of the how well discussions on whether to implement a feature could be captured in DoArgue. While we found that all related statements could be represented in the framework, there was difficulty in labelling some arguments. These arguments were those made for implementing a feature request before any defects were identified. An example of this can be seen in the description of the running example, where the stakeholder djk states that 'There's so much empty space to the right of menu items; I'd like to be able to stick some buttons there'. This is an important argument for not rejecting a feature request, but at the time when the statement is made there is no 'reject feature' defect on which to place it. One solution to this problem would be to allow for arguments in favour of the feature request.

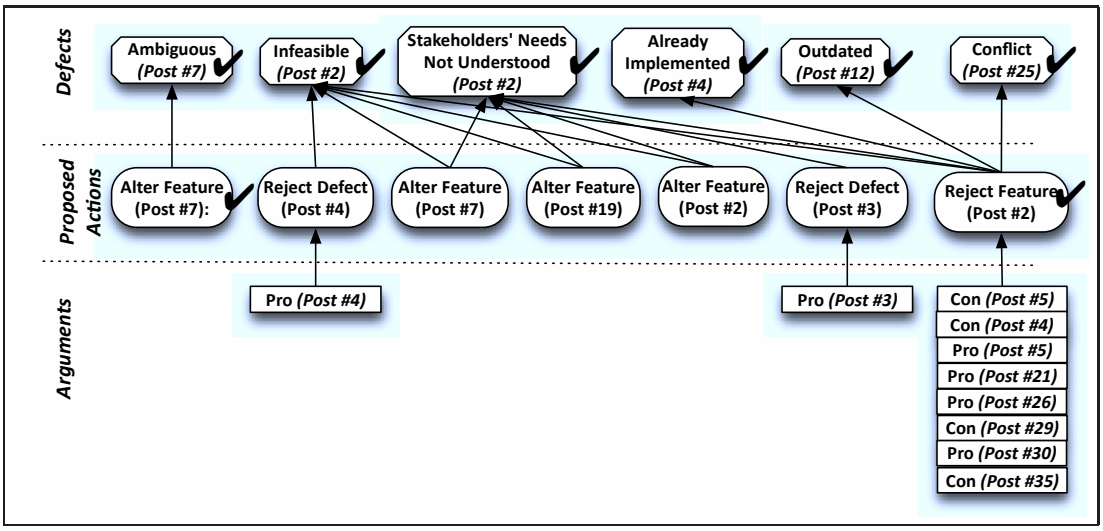



Figure 4.9: DoArgue Visualisation: Firefox FR #204999

We also recorded the frequency of each concept type that we labelled across the 50 feature requests, shown in Figure 4.10. The value in parenthesis for proposed resolutions denotes how many of each type was selected to resolve a defect, and defects the value in parenthesis denotes the number that were resolved. The discussions contained a relatively large number of concepts from the DoArgue framework (297 across the 50 feature requests). Only 7 feature requests did not contain any of these concepts, each of which reported features that were implemented without criticism from other stakeholders.

| Labelled Comments 297 | | | | | |
|---|---|---|---|---|---|
| Defects 71 (60 Resolved) | Proposed Resolutions 95 (53 Chosen) | | | Arguments 131 | |
| | Reject Feature 40 (34 Chosen) | Alter Feature 33 (7 Chosen) | Reject Defect 19 (12 Chosen) | Pro 82 | Con 49 |

Figure 4.10: Frequencies of Concepts Labelled in 50 Firefox Feature Requests

### 4.3.3 Threats to Validity

These experiments are intended to explore how much effort it takes to use the DoArgue framework, and whether it captures the key concepts relating to discussions on whether

to implement a feature. They should not be taken to be a full empirical evaluation of the framework as the results are not sufficiently valid for this purpose. Reliability is questionable, since the results are underpinned by our own biases in the interpretation of discussions. Moreover, we made the assessment of how much effort is required to use the framework ourselves, which may not reflect how easily the average user might find it to use. External validity is also lacking, since the experiments are based on a relatively small sample from one feature request management system. Further, the experiments do not directly test whether the framework could be used to transfer the benefits of design rationale frameworks to a feature request management system (as discussed in Section 4.1.2).

An extended evaluation is needed to address these threats to validity if the framework is to be convincingly validated. This could be done via controlled experiments in real projects with the availability of the framework as an independent variable. Adoption of the framework could be directly measured via stakeholder participation levels. A measurement of whether stakeholders' needs are better understood when using the framework would be harder to obtain; one approach would be to ask stakeholders and developers this question directly via a survey (such as that used by Paula Laurent and Jane Cleland-Huang [Laurent and Cleland-Huang, 2009]), while another could be to indirectly measure this by asking stakeholders to rate how well the finished product fits their needs.

Performing this self-evaluation made us realise that it would not be easy to convince stakeholders in a real project to start using the DoArgue framework. We therefore began developing an alternative approach to support decision making in feature request management systems; the early failure prediction framework in Chapter 5 that automatically predicts failures associated with feature requests without the need for users to annotate discussions.

## 4.4 Related Work

Many frameworks have been proposed to improve stakeholders' ability the express their needs and make these needs understood in distributed collaborative software development. WikiWinWin [Yang et al., 2008], for example, provides a platform for stakeholder negotiation on early stage requirements. Stakeholders collaborate using a wiki structured on concepts from the WinWin method [Boehm et al., 1995]; namely win-conditions, points-of-agreement, issues and options. The SOP project [Decker et al., 2007], meanwhile, structures a wiki on scenarios and use cases. A host of other frameworks have been proposed that structure collaboration in a web-based framework on a requirements model [Riechert and Berger, 2009, Abeti et al., 2009, Knauss et al., 2009]. These frameworks, however, are alternatives to feature request management systems. DoArgue, by contrast, extends the feature request management system - which is already a popular tool for distributed collaboration on requirements activity.

Language analysis tools are an automated alternative to defect identification. Bettenburg et al.'s tool CUEZILLA, for example, analyses bug descriptions [Bettenburg et al., 2008] for readability, and the Requirements Analysis Tool [Verma and Kass, 2008] can be used to detect statements that are indicators of defects, such as ambiguities or over-specification. These tools identify defects relating to the wording of a feature request's description, and cannot identify inadequacies in the suggested feature itself. They could, however, support the framework by automatically adding defects they identify to the DoArgue model.

As a result of their survey study on the problems faced in features request management forums Paula Laurent et al. have recommended practices to make stakeholders' needs more understandable [Laurent and Cleland-Huang, 2009]; including more developer participation in discussions, and a clearer feature prioritization strategy. These recommendations focus on improving the development culture in feature requests, while the DoArgue framework provides a tool-supported solution to the challenge.

## 4.5 Chapter Summary

In this Chapter we have presented a tool-supported framework, DoArgue, for structuring feature request discussions by allowing stakeholders to explicitly state if posts they submit are identifying defects, exploring possible resolutions to them, or providing argumentation for these resolutions. The framework can thereby generate a conceptual model of discussions relating to whether a feature should be implemented. Design rationale literature suggests that structuring discussions in this way improves the clarity and outcomes of web-based conversations on complex topics.

A case study based evaluation in which 50 feature requests from the Firefox project were modelled using the framework suggests that it is expressive enough to capture the essential information from discussions on whether to implement a feature, and requires minimal additional effort to use. This suggests that there is a strong possibility that the framework could be developed further to extend feature request management systems and encourage stakeholders to express their needs clearly and fully.

The approach described in this Chapter is a promising solution to a key challenge in feature request management systems: making stakeholders' needs more easily understood. Validation of this claim, however, requires an evaluation of user adoption and whether the software developed better suits stakeholders' needs, which could not be conducted in the time frame of this thesis. We therefore focused our efforts on the early failure prediction approach presented in the next Chapter as a response to the challenge of guiding upfront analysis in feature requests.

# 5 Predicting Feature Request Failures

*In this Chapter we present a automated tool-supported framework in response to a key challenge in feature request management systems: "Where, how, and how much upfront requirements analysis should take place in a body of feature requests?". The early failure prediction approach generates alerts for feature requests at risk of failure when a decision is taken on whether to assign the feature for implementation. Developers can use this information to perform additional upfront requirements analysis to avoid these failures. A cost-benefit model determines, for a given prediction model's recall and precision, whether the value expected from performing additional upfront requirements analysis on alerted feature requests will out-weight the costs.*

*We begin this Chapter by describing how a given failure prediction model can be validated against a set of baselines in terms of the costs and benefits to a project provided by acting upon its predictions. We then describe the framework for constructing such a failure prediction model, discuss the types of upfront analysis that could be carried out to avoid predicted failures, and describe a tool implementation. Finally, we report a set of failure prediction experiments on six large scale projects which show the approach can be used to generate prediction models that perform better than a set of baselines for many failures types and projects.*

## 5.1 Valuing Predictions

### 5.1.1 Cost-Benefit Model

An early failure prediction model for a failure of type $T$ is a function that generates an alert for each feature request that it believes will result in a failure of this type. Formally, if $FR$ is a set of feature requests for which predictions are sought, the result of applying a prediction model is a set $Alert \subseteq FR$ denoting the set of feature requests predicted to fail.

The quality of such predictions can be assessed using the standard information retrieval measures of precision and recall. Let $Failure \subseteq FR$ be the set of feature requests that will actually have a failure of type $T$. This set is unknown at prediction time. The true positives are the alerts that correspond to actual failures, i.e. we define the set $TruePostive = Alert \cap Failure$. The precision of a set of predictions is the proportion of true positives in the set of alerts, and its recall is the proportion of true positives in the set of all actual failures, i.e.

$$precision = \frac{|TruePositive|}{|Alert|} \quad recall = \frac{|TruePositive|}{|Failure|}$$

When designing a prediction model, there's an inherent conflict between these two measures: generating more alerts will tend to increase recall but decrease precision, whereas generating less alerts will tend to increase precision but decrease recall. An important question when designing and evaluating prediction models is to find the optimal trade-off between precision and recall.

A standard measure used in information retrieval for combining precision and recall is an f-score, which corresponds to a harmonic mean between precision and recall. This score, however, relies on attaching an arbitrary importance to the two measures and has little meaning in our context.

We instead assess the relative weights to be given to precision and recall by assessing the costs and benefits to a project for acting on a set of predictions. The model is deliberately simple to facilitate its use and comprehension. To use our model, a user need estimate only two parameters: $P_s$, which denotes the probability that additional upfront analysis on a feature request will be successful at preventing a failure of type $T$, and $\frac{C_f}{C_a}$, which denotes the relative cost of a failure of type $T$ compared to the cost of the additional upfront analysis. Finer-grained cost-benefit models are possible but would require estimations for a more complex set of model parameters and thereby reduce our confidence in the results.

We evaluate the expected benefit of a set of predictions $P$ as follows. Assuming that each failure of type $T$ imposes a cost $C_f$ to the project when it occurs, if we could prevent all

failures for which an alert is generated, the benefit to the project would be $|TruePositive| \times C_f$. We have to take into consideration, however, that not all additional upfront analysis will be successful at preventing a failure. If we assume that the probability of success of additional upfront analysis is $P_s$ then the expected total benefit of a set of predictions is the product $|TruePositive| \times P_s \times C_f$. Given that $|TruePositive| = |Alert| \times precison$ we obtain the following equation characterizing expected benefit:

$$ExpectedBenefit = |Alert| \times precision \times P_s \times C_f$$

We evaluate the expected cost of a set of predictions as follows. Assuming that each alert is acted upon and that the cost of additional requirements elaboration is $C_a$ for each alert, the total expected cost associated with a set of predictions is given by the following equation:

$$ExpectedCost = |Alert| \times C_a$$

The expected net value of a set of predictions is then given by the difference between its expected benefit and cost:

$$ExpectedValue = |Alert| \times (precision \times P_s \times C_f - C_a)$$

Since $|Alert| = |Failure| \times \frac{recall}{precision}$, the equation can be reformulated as:

$$ExpectedValue = |Failure| \times \frac{recall}{precision} \times (precision \times P_s \times C_f - C_a)$$

By simplifying and factoring $C_a$, the formula is expressed as:

$$ExpectedValue = C_a \times |Failure| \times recall \times (P_s \times \frac{C_f}{C_a} - \frac{1}{precision})$$

Difficulties lie in estimating absolute values for $C_a$ and $C_f$, so instead we assume that the cost of additional upfront analysis provides the unit of measure and ask model users to

estimate the relative cost of failure with respect to the cost of additional upfront action. This relative measure is often used in empirical studies about the cost of failures in software development projects, although there are caveats about the context of applicability of the different results [Shull et al., 2002]. The ratio between the cost of fixing a requirements defect after product release compared to fixing it during upfront requirements analysis is commonly cited to be between 10 and 100 for large projects [Boehm and Papaccio, 1988] [van Lamsweerde, 2009] [McConnell, 2004], and between 5 to 10 for smaller projects with lower administrative costs [Boehm and Turner, 2003] [McConnell, 2004].

Fixing the cost of action to 1 gives our final formula characterizing expected value:

$$ExpectedValue = |Failure| \times recall \times (P_s.\frac{C_f}{C_a} - \frac{1}{precision})$$

We have kept the term $\frac{C_f}{C_a}$ instead of simply $C_f$ to make explicit that the cost of failure is relative to the cost of additional analysis. Since the number of failures is a constant for a given set of feature requests, we can compare alternative prediction models by assessing their expected value per failure:

$$\frac{ExpectedValue}{|Failure|} = recall \times (P_s \times \frac{C_f}{C_a} - \frac{1}{precision})$$

In this formula precision and recall are characteristics of a prediction model that can be estimated from its performance on past feature requests. If one wants to know the expected value per feature request, this can be obtained by multiplying the expected value per failure by the failure density $\frac{|Failure|}{|FR|}$ - a constant that can be derived from past feature requests.

The parameters to be estimated by model users are $P_s$ and $\frac{C_f}{C_a}$. The expected value actually depends on the product of these parameters denoted $\alpha$, i.e. $\alpha = P_s \times \frac{C_f}{C_a}$. We therefore obtain the following equation:

$$\frac{ExpectedValue}{|Failure|} = recall \times (\alpha - \frac{1}{precision}) \tag{5.1}$$

In principle, it might be possible to estimate $P_s$ and $\frac{C_f}{C_a}$ empirically from past project data. However, such project specific data is rarely available. In the absence of this data, model users can estimate these numbers based on the general findings from the empirical studies reported in the Section 3.3.2, and assess the value of a prediction model for a range of $\alpha$ values rather than a single point. Considering product failures in the Firefox project, for example, if a model user estimates the ratio of the cost of failure to the cost of action to be between 50 and 100, and estimates the probability of success in preventing such failure by additional upfront analysis to be between 0.1 and 0.3, then the $\alpha$ values of interest will be between 5 (50x0.1) and 30 (100x0.3).

### 5.1.2 Baselines and Prediction Objectives

The advantage of assessing predictive models using equation 5.1 as opposed to more commonly used techniques such as an f-score is that the expected value has a clear meaning for the project and the parameters to be estimated for computing it are at least in principle measurable. We argue that it is more meaningful to ask model users to provide estimates for $P_s$ and $\frac{C_f}{C_a}$ than asking them estimate the relative weights of precision and recall.

To understand equation 5.1 we can look at how the expected value per failure varies with $\alpha$ for some prediction models with known precision and recall, as shown in Figure 5.1.

A perfect predictor would be one that generates alerts for all failures and generates no false alerts. Its precision and recall are both 1. We can observe that even for a perfect predictor, the expected value is positive only if $\alpha = P_s \times \frac{C_f}{C_a} > 1$. This means that for any upfront activity to have a positive value the ratio between the cost of the failure it may prevent and its own cost must be higher than the inverse of its probability of success in preventing the failure. For example, if additional upfront requirements analysis may prevent some failure type with an estimated probability of success of 0.1, the cost of this activity (e.g. the number of man-hours it takes) must be at least 10 times smaller than the cost of late correction of the failure it intends to prevent.

The most pessimistic predictor is one that generates alerts for all feature requests. Its

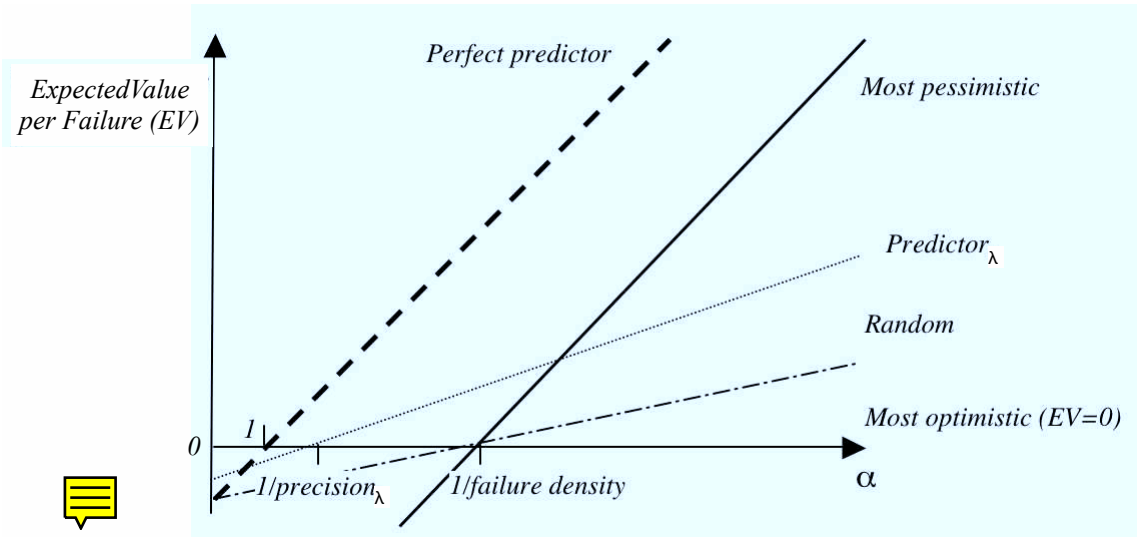



Figure 5.1: Expected Value Per Failure as a Function of $\alpha$

recall is equal to 1 and its precision to the failure density. The most optimistic predictor is one that never generates any alerts. Its recall is 0, precision 1, and expected value is therefore always null. The random predictor is one that randomly generates alerts for feature requests using the failure density for past feature requests as the probability of alert. We can observe from equation 5.1 that the best of these three baseline predictors is the most optimistic predictor when $\alpha$ is less than the inverse of the failure density; and it is the most pessimistic predictor when $\alpha$ is more than the inverse of the failure density. The random predictor is always outperformed by one of these two. This means that, unlike other failure prediction models that compare themselves against a random predictor [Nagappan et al., 2006] [Cleland-Huang et al., 2009] [Wolf et al., 2009], the baselines in our context are the most optimistic and most pessimistic predictors.

Figure 5.1 also shows the expected values for a predictor $\lambda$ whose precision is higher than the failure density. Such a predictor outperforms the most optimistic baseline when $\alpha > \frac{1}{precision_\lambda}$ and it outperforms the most pessimistic one when $\alpha < \frac{1}{1-recall_\lambda} \times (\frac{1}{FailureDensity} - \frac{recall_\lambda}{precision\lambda})$[†]. When $\alpha$ is outside of this range - either below or above it - the most optimistic or pessimistic predictors have better expected values. This provides

[†]These constraints are derived from Equation 5.1 by finding values for $\alpha$ such that the expected value for a predictor $\lambda$ is higher than the expected value for the most optimistic and most pessimistic baseline predictors, respectively.

a quantitative justification for the intuition that for a given set of failure predictions with imperfect recall and precision, if the relative cost of failure and the probability of success of additional upfront analysis in preventing the failure are low, then it is more cost-effective not to do any additional analysis; whereas if the cost of failure and probability of success are high, then it is more cost-effective to perform additional analysis on all feature requests.

If a predictor has a precision that is less than the failure density it is always outperformed by the most optimistic or most pessimistic predictors. Our objective when developing prediction models will therefore be to generate predictors whose precision is higher than the failure density and whose range of $\alpha$ values for which it outperforms the most optimistic and most pessimistic predictors is as large as possible.

## 5.2 Predicting Failures

The class of machine learning techniques that apply to our problem, known as classification algorithms, first construct prediction models from historical data and then use these models to predict classifications for new data. In our case the historical data used to generate prediction models is past feature requests, while the new data consists of feature requests that are about to be assigned or rejected on which we wish to predict future failures.

We have developed a tool-supported framework that generates alternative prediction models from historical data sets. These models vary according to the characteristics of feature requests discussions they take into account for predicting failures and the classification algorithms they rely on for constructing prediction models.

### 5.2.1 Generating a Prediction Model

To generate a single prediction model a user of our framework must specify the following inputs:

- The feature request management database to be used, in which the project where we wish to predict failures is held.
- The failure type to be predicted from Section 3.3.1.
- The classification algorithm to be used. Examples include the Linear Regression and M5P-Tree algorithms [Witten and Frank, 2005].
- A *predictive attribute* of a feature request discussion used to train a predictive model. Examples include the timing between posts, the textual content of the discussion, the number of code patches submitted to the discussion thread, or the number of participants.
- An estimated value for $\alpha$ used by our framework to make trade-offs between precision and recall in predictive models, and to compare the expected value of these models to the baselines for validation.

Using these inputs, our automated failure prediction framework follows the process shown in Figure 5.2 performing the following steps to construct a failure prediction model:

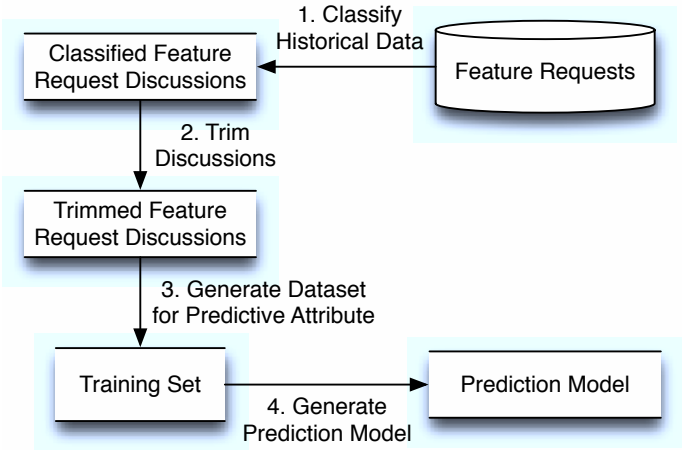



Figure 5.2: Framework for Generating a Prediction Model

**1. Classify Historical Data**: A large dataset is extracted from the feature request management forum. This data in its raw form includes the textual content of discussions, their structure, their associated meta-data and the history of changes made to this meta-data. To build a prediction model for rejection reversal failures we extract all feature

requests that have been rejected at some point in their lifetime, and for the other four failures we extract all those that have at some time been assigned for implementation. Each feature request is automatically identified as a positive or negative instance of failure with respect to the rules defined in the Section 3.3.1.

**2. Trim Discussions**: Posts within each feature request are trimmed to the point in its life-cycle where we want to make predictions. This will be just before it was rejected for the rejection reversal failure, and just before it is assigned for the other four types of failure.

**3. Generate Dataset for Predictive Attribute**: The trimmed discussions are then transformed into a data set that contains the relevant information about the attribute of interest and in a format that can be processed by classification algorithms. For example, if the attribute of interest is the number of participants in the discussion, this set will generate a set of tuples <feature request, number of participants, classification>.

**4. Generate Prediction Model**: The data set is then used to generate a prediction model using the selected classification algorithm. If the classification algorithm is one that directly classifies a feature request as a failure or not, such at the Decision Table and Linear Regression algorithms, then no further steps are needed. Other classification algorithms such as the Naive Bayes and M5P-Tree algorithms assign to each feature request a numerical score between -1 and +1 indicating whether the feature is more likely to be a failure (if its score close to 1) or not (if close to -1). In these cases, we still need to set a threshold such that all feature requests whose score exceeds the threshold generate a failure alert. Our framework sets this threshold automatically by estimating the model's precision, recall, and expected value (using 10 fold cross validation as described in the next subsection) for a range of threshold values and selecting the threshold that yields the highest expected value. For example, Figure 5.3 shows how the recall, precision and expected value vary with this threshold for a set of predictions made on product failures in the Firefox project and an estimated $\alpha$ of 3. In this case a threshold of 0.36 will be chosen. In cases where a prediction model cannot provide a positive value the threshold is automatically set so as to generate no alerts, thus mimicking the most optimistic baseline.

Conversely, in cases where the highest value is provided by generating all alerts the most pessimistic baseline is mimicked.

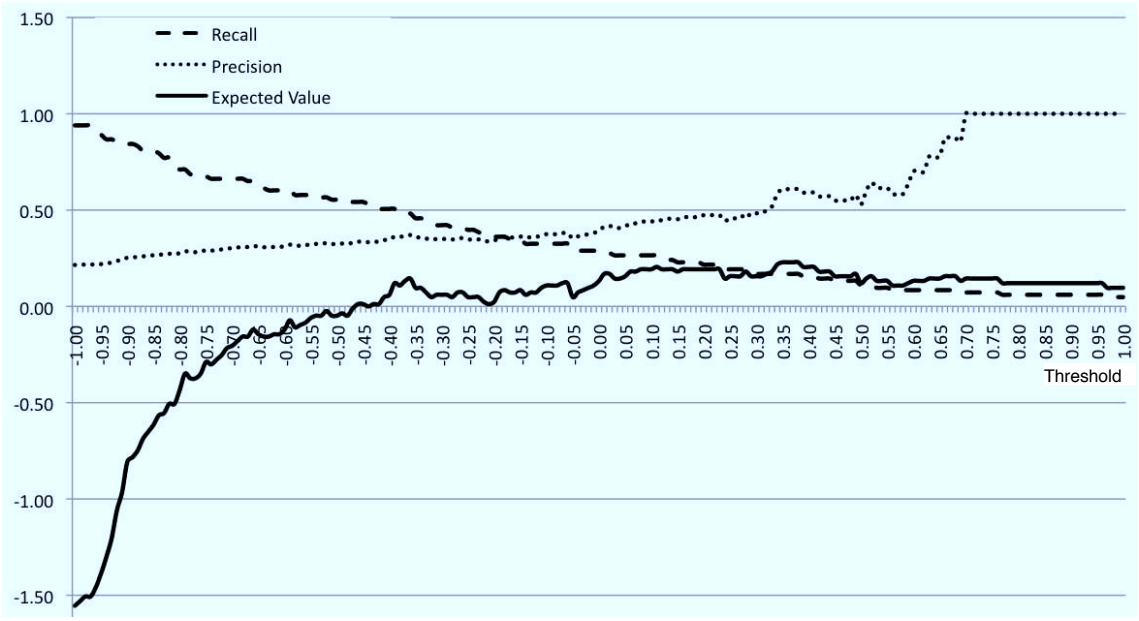



Figure 5.3: Recall, Precision and Expected Value as a Function of Alert Threshold

### 5.2.2 Evaluating a Predictive Model

A standard technique for evaluating prediction models consists in performing a 10-fold cross validation [Witten and Frank, 2005]. The data set is split into a training set composed of 90% of feature request selected at random and a testing set composed of the remaining 10%. A prediction model is built using the training set, and its precision and recall computed for the testing set. This experiment is repeated 10 times with different training sets and testing sets. The precision and recall of the prediction model obtained from the full data set are then estimated as the mean of the precision and recall for the 10 experiments. Equation 5.1 in Section 5.1 is then used to evaluate the prediction model's expected value from its precision, recall, and assumed $\alpha$ value.

## 5.3 Reducing the Risk of Failure

When taking a decision to assign a specific feature for implementation or reject it, if a developer or project manager knows that a failure is likely to occur they can perform more upfront requirements analysis to reduce this risk before taking the decision. We now discussed how this could be done for each of the five failure types defined in Section 3.3.1.

Removed feature and product failures result from unwanted behaviour caused by an implemented feature. To reduce the risk of such failures additional upfront analysis can include checks to see that all behaviour specified by a feature request is really wanted by stakeholders, and whether it conflicts with existing or planned features. Extra attention can also be given to the development of the feature to make sure that it implements its specification correctly.

Scrapped implementation failures can be mitigated by conduct feasibility studies and further analysis into whether a feature is really wanted by stakeholders. If a feature is found to be infeasible or in conflict with stakeholder goals then it can be rejected earlier, freeing up the time that would have been spent coding implementations.

Rejection reversal failures result from poor decisions to reject a feature. Developers can reduce the risk of such failures by eliciting input other stakeholders, or asking stakeholders already in the discussion to strengthen their rationale for rejecting a feature. Project managers can also monitor feature requests that have a high risk of this failure to check whether developers are objectively considering stakeholders' needs.

Stalled development failures can have the risks associated with them reduced if developers do a feasibility analysis on a feature to check if there may be barriers to implementing a feature and attempt to address these at an early stage. Project managers can also assign more developers to features that are likely to have this type of failure, or monitor these features and provide encouragement to developers if progress is not made on development.

## 5.4 Intueri: An Implementation

We have developed a tool, Intueri, that allows users to generate and evaluate predictive models using the framework described in this Chapter. Intueri's front-end is developed in Flash enabling it to be run in a web browser, while it's back-end is powered by ActionScript making use of PHP to communicate with other components and to extract, load and save data. Data can currently be extracted from Bugzilla-based feature request management systems and is converted to an XML format. The generation of prediction models makes use of the open source Weka libraries [Hall et al., 2009], which provide an interface to many classification algorithms. Results are currently stored and analysed in Matlab.

Figure 5.4 shows how data is stored and processed in the tool to realise the prediction framework and validation described in the previous Section. Screenshots from this process can be found in Appendix B.

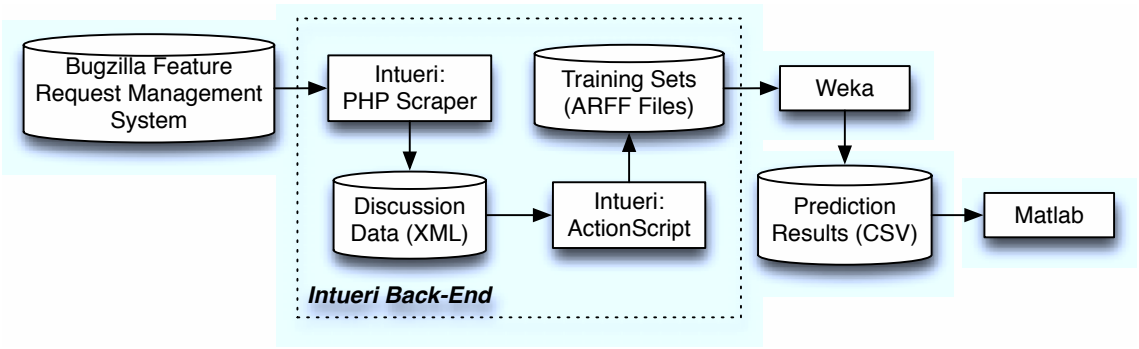



Figure 5.4: Data Flow in Intueri

The tool is available online[1] alongside the data we have used for the prediction experiments in Section 5.5. This includes the PHP files used to extract raw data from feature request management systems and automatically identify failures, raw discussion extractions from feature request management systems in XML format, the training sets that can be given as input to the WEKA tool to generate prediction models, and the full results of our experiments.

[1] `http://sre-research.cs.ucl.ac.uk/Intueri/`

## 5.5 Prediction Experiments

We have applied our early failure prediction framework to six large scale open source projects. The questions that we wished to answer with our experiments are the following:

1. What classification algorithms generate the most valuable prediction models?
2. Can feature request failures of the types defined in Section 3.3.1 be predicted from early discussions before the feature request is either assigned or rejected? What expected value can a project hope to obtain from actioning such predictions? Are some of failure types more susceptible to being predicted from early discussions? Are certain types of projects more susceptible to early failure prediction approach?
3. What attributes of early feature requests discussions, if any, can be used as reliable predictors of later failures? Do some attributes perform better for certain types of failures and across projects and failure types?

The full set of experiments can be found online, including the predictive models generated, the datasets used to generate them, the evaluation of these models, and the tool used to conduct the experiments[2].

### 5.5.1 Experimental Setup

Prediction models were generated and evaluated for a set of real large scale projects. We have not included the Eclipse project in these experiments because some failures were incorrectly identified using the automated failure detection approach (as discussed at the end of Section 3.3.2); and therefore the validity of failure prediction for this project is questionable. The following inputs were used to generate and evaluate prediction models using the process defined in Section 5.2.1:

- The six feature request management databases that held the projects: Apache, Firefox, KDE, Netbeans, Thunderbird and Wikimedia.

[2] `http://sre-research.cs.ucl.ac.uk/Intueri/`

- For each project prediction models were generated for each of the 5 failures defined in Section 3.3.1.

- Alternative models were generated using the Decision Table, Naive Bayes, Linear Regression and M5P-Tree algorithms, which constitute a good representation of the different types of classification algorithm [Witten and Frank, 2005]. The Decision Table and Naive Bayes algorithms are the least computationally complex, but have been known to perform well for less computational effort. Decision table constructs a simple decision tree to predict failures from the prediction attributes; while Naive Bayes computes a median value for each predictive attribute from historical failed and successful feature requests, and matches a new feature request's attributes to the case that they lie most closely to. Linear Regression, meanwhile, performs a full correlative analysis by constructing a set of best-fit linear functions from historical data to predict failures from attributes. Lastly, the M5P-Tree algorithm creates a decision tree to group feature requests with attributes that result in similar probabilities of failure, and then generates a linear regression prediction model for each leaf node. The computational complexity of M5P-Tree is usually less than that of Linear Regression since each node typically generates a simpler correlative function.

- The following 13 attributes were to generate a prediction model for each failure type in each project:

  - Attributes relating to discussion participants: the number of participants, the number and percentage of posts by the person who reported the feature request, the number and percentage of posts made by the person who is assigned to develop the feature request.

  - Attributes relating to the structure and development of the discussion: The total number of posts, the number of words in each post, the number of words in the whole discussion, the number of code contributions submitted to the feature request's thread, the time elapsed between posts, and the total time elapsed in the discussion.

  - Textual attributes: Bag of words and term frequency in document frequency

(TFIDF), which are two approaches for finding patterns in the textual content of discussions. Bag of words assigns an attribute to each unique word in a feature request with a value corresponding to the number of occurrences of the word. TFIDF, meanwhile, uses the same process but multiplies this value by the 'uniqueness' of each word (the inverse of the number of occurrences of the word across all feature requests in the historical dataset).

- For each failure type we estimated set of parameters and corresponding alpha value for the cost-benefit equation used to evaluate a prediction model from Section 5.1.1. These can be found in the header of Table 5.1 of the results, and a discussion of the errors in estimation of these parameters is given at the end of Section 5.5.2.

### 5.5.2 Results

| | **Product Failure** | **Abandoned Implementation** | **Rejection Reversal** | **Stalled Development** | **Removed Feature** |
|---|---|---|---|---|---|
| $\frac{C_l}{C_a} * P_s = \alpha$ | 10*0.3 = 3 | 50*0.5 = 25 | 10*0.5 = 5 | 10*0.5 = 5 | 20*0.2 = 4 |
| **Apache** | - | Code Submissions | Total Word Count | Code Submissions | - |
| **Firefox** | Posts in Discussion | *(Most Pessimistic)* | *(Most Optimistic)* | Code Submissions | *(Most Optimistic)* |
| **KDE** | - | Posts by Reporter | Bag of Words | Bag of Words | *(Most Optimistic)* |
| **Netbeans** | *(Most Optimistic)* | TF-IDF | Total Word Count | Percent by Assignee | Bag of Words |
| **Thunderbird** | Code Submissions | Bag of Words | *(Most Optimistic)* | Bag of Words | *(Most Optimistic)* |
| **Wikimedia** | *(Most Optimistic)* | Bag of Words | Num of participants | Bag of Words | *(Most Optimistic)* |

Table 5.1: Prediction Models with Best Expected Values

To answer the first question, we have evaluated the performance of each classification algorithm for all failure types and all of 13 prediction attributes on the data set for the Firefox project only. The experiment revealed that prediction models generated by the Decision Table and Naive Bayes classification algorithms failed to yield meaningful predictions. This might be expected as these algorithms perform a comparatively low amount of correlative analysis on data. The M5P-tree and Linear Regression algorithms, meanwhile, consistently produced similar results in terms of expected value. In all cases, however, the less computationally complex M5P-tree algorithm generated prediction models signif-


Figure 5.5: Expected Value per 100 Feature Requests from Best Predictive Models

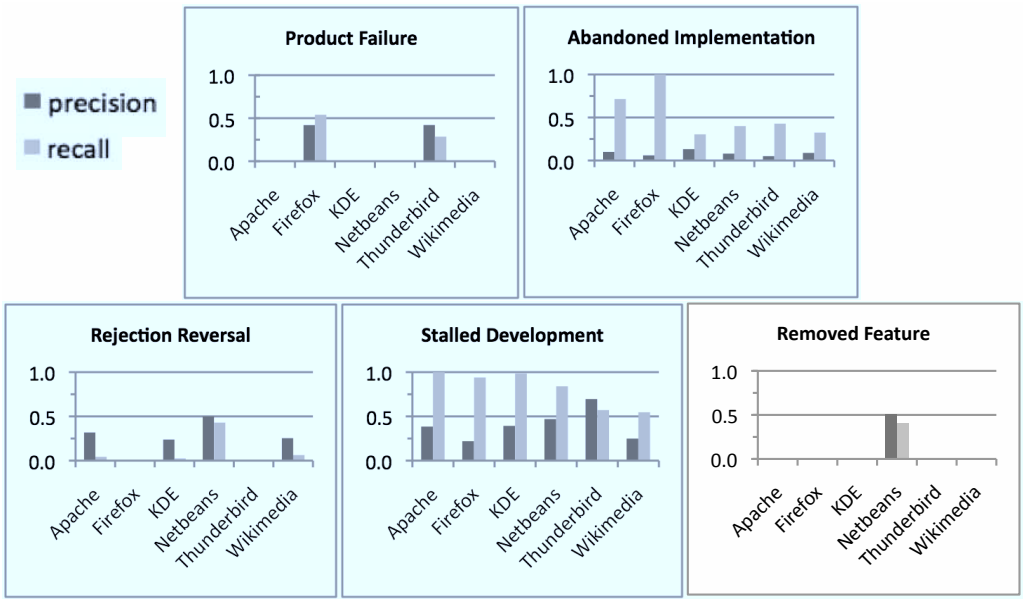



Figure 5.6: Precision and Recall from Best Predictive Models

icantly faster than Linear Regression - in the order of minutes as opposed to hours. For remainder of the experiments we therefore only used the M5P-tree algorithm.

To answer the second set of questions we have generated and evaluated prediction models for all failure types and predictive attributes in all six projects. Table 5.1 summarises the results by presenting for each project and failure type the prediction model that yielded the most expected value. That which gave the most expected value for product failure predictions in Firefox, for example, was generated using the 'number of posts in the discussion' attribute. The table also shows the estimated $\alpha$ value we have used for setting the alert threshold and computing the expected value. When the best prediction model is shown as being the most pessimistic or most optimistic this means that there was no predictive model was generated which performed better than these baseline predictors. In such cases, as you will recall from Section 5.2, the alert threshold for prediction models is automatically set so that it behaves as the best baseline predictor. Figure 5.5, meanwhile, shows the expected values per 100 feature requests for the predictive models, and Figure 5.6 shows their respective precision and recall.

The results show that it is possible to generate early failure prediction models that provide positive expected value to a project. Rough estimations of the real value expected from actioning predictions can also be made. As you will recall from Section 5.1 the cost of an action was fixed to 1 in our equation, and we can therefore multiply the expected value by an estimation of the real cost of an action in dollars or man-hours to quantify it. For example, if for product failures in Firefox the cost of failure is estimated to be 10 times that of an action and the probability of success is 30% then acting on failure predictions with a recall of 0.54 and a precision of 0.42 our equation gives us an expected value of $0.54 \times (0.3 \times \frac{200}{20} - \frac{1}{0.42}) = 0.33$ per failure. Multiplying this result by an estimated cost of action of 10 hours gives us a quantified value of 3.3 hours saved per failure.

The results suggest that on these six projects our approach is more effective at predicting stalled development and abandoned implementation failures than the other failures. A possible explanation for this is that these failures have higher densities than the others and that they occur earlier (closer to the time of prediction), than removed feature and

product failures.

While the Netbeans project could benefit the most from the prediction models constructed, there is no general case for some projects being more susceptible to early failure predictions than others. The results do not confirm whether projects that update their meta-data more consistently such as the Firefox project (as discussed in Section 3.3.2) would gain more from early failure predictions.

The recall and precision of the models demonstrate how the cost-benefit equation is used to make the trade-off between these measures by maximising the expected value with respect to the failure type. For abandoned implementation failures, for example, preference is given to recall over precision since the high estimated $\alpha$ value weights the benefits of taking action to avoid values to be far more than the costs. The expected value can also show us where prediction models might be able to provide value to a project in cases where a low precision and recall are achieved, such as for rejection reversal failure predictions in the KDE project.

The $\alpha$-values we have used for the different failure types have been estimated by ourselves and represent our best informed guesses based on general empirical studies of software projects [Boehm and Papaccio, 1988] [van Lamsweerde, 2009] [McConnell, 2004] [Boehm and Turner, 2003] [McConnell, 2004]. In the absence of empirical validation specific for each project these values are certainly subject to debate. A benefit of our evaluation framework is that it is possible to assess how deviation between the estimated value for $\alpha$ and its real (unknown) value will impact the expected value of a prediction model. Figure 5.7, for example, shows how the expected value for different product failure prediction models in Firefox vary with $\alpha$ (in which $\alpha$ was set to 3 to determine the alert threshold). We can see from the figure that the prediction model based on the number of posts in the discussion still performs better than the most pessimistic predictor if the real value for alpha goes up to about 6.5, but above that point the most pessimistic predictor (i.e. the one that suggests performing additional upfront analysis on all feature requests) yields a higher value. Similar graphs resulted from the other cases in which predictive models yielded more expected value than the baselines.

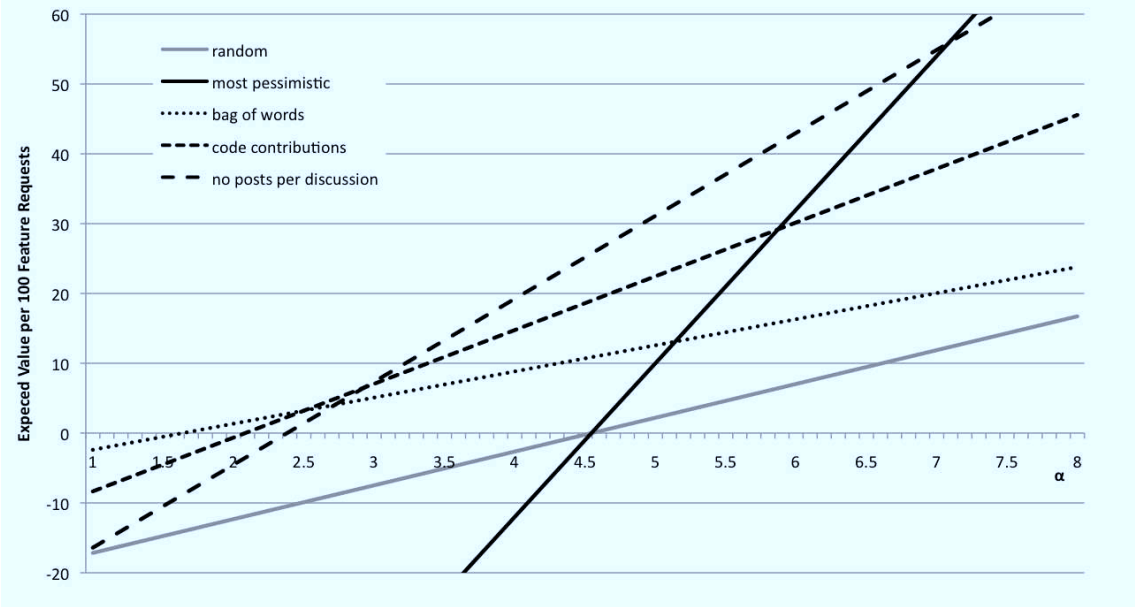



Figure 5.7: Expected Value vs. $\alpha$ for Firefox Product Failure Predictions

Table 5.2 helps us to answer the third set of questions. For each of the 13 predictive attributes the number of projects in which a prediction model was generated that outperformed the baselines is shown for each failure type. The numbers shown in brackets are the average expected value of these models. The code submissions predictive attribute, for example, generated stalled development prediction models that performed better than the baselines in 5 of the 6 projects studied, and the average expected value of these models was 59.61. Expected values should not be compared across failure types due to the differences in their units.

Interestingly, the predictive attributes that performed consistently well on all failure types for both projects are the two text-based attributes: bags-of-words and TFIDF. This suggests that analysing the actual content of the discussion, even at a very rudimentary level, could provide more reliable predictions than analysing attributes such as the number of persons involved in the discussion and the discussion length. The actual words that have the highest influences on whether a feature request is classified as a failure or not are surprisingly simple words such as “would”, “like”, and the product name, e.g. “Firefox”. Unfortunately, this doesn’t provide useful insights on how to write better feature requests to reduce the true risks of failures. One should not expect to obtain useful insights from

| Predictive Attribute | Product Failure | Abandoned Implementation | Rejection Reversal | Stalled Development | Removed Feature | Total Viable Models |
|---|---|---|---|---|---|---|
| Num of participants | 1 (0.82) | 1 ( 7.20) | 3 (0.21) | 4 (53.86) | 0 (0.00) | 9 |
| Posts by reporter (#) | 1 (0.07) | 4 (10.78) | 1 (0.01) | 2 (43.58) | 0 (0.00) | 8 |
| Posts by reporter (%) | 2 (0.87) | 1 ( 7.20) | 2 (0.02) | 4 (52.92) | 0 (0.00) | 9 |
| Posts by assignee (#) | 2 (1.09) | 1 ( 7.20) | 0 (0.00) | 4 (56.61) | 0 (0.00) | 7 |
| Posts by assignee (%) | 0 (0.00) | 1 ( 9.63) | 1 (0.05) | 4 (56.65) | 0 (0.00) | 6 |
| Posts in discussion | 2 (1.18) | 4 ( 9.33) | 2 (0.60) | 4 (55.49) | 0 (0.00) | 11 |
| Total word count | 2 (0.75) | 4 (10.04) | 1 (0.82) | 5 (55.16) | 0 (0.00) | 11 |
| Word count per post | 2 (0.87) | 5 (13.38) | 3 (0.28) | 2 (42.41) | 1 (0.01) | 13 |
| Code submissions | 2 (1.35) | 2 (13.28) | 0 (0.00) | 5 (59.61) | 0 (0.00) | 9 |
| Time elapsed per post | 0 (0.00) | 5 (10.51) | 3 (0.13) | 6 (53.78) | 1 (0.04) | 15 |
| Total time elapsed | 1 (0.73) | 1 ( 7.20) | 1 (0.35) | 3 (52.08) | 1 (0.04) | 7 |
| Bag of words | 1 (0.86) | 6 (15.55) | 2 (0.73) | 6 (63.36) | 1 (0.26) | 17 |
| TF-IDF | 1 (0.36) | 4 (14.60) | 3 (0.89) | 6 (61.86) | 1 (0.20) | 14 |

Table 5.2: Which Predictive Attributes Perform Well For Which Failure Types?

predictive models using very simple natural language characteristics related to word occurrences. Richer language-based analysis based on sentence structures and the presence of specific keywords and phrases (e.g. typical ambiguous phrases or typical phrases that reveal the presence of rationale such as "so that" or "in order to") may give better and more useful failure prediction models. This is an interesting avenue for future research.

No predictive attribute always outperformed the others for a specific failure type in all the projects studied. When generating a predictive model for a new project, therefore, a range of predictive attributes could be evaluated as we have done in our experiments to find that which gives the highest expected value.

A much wider range of attributes than the ones we have used in our experiments could be tested for early failure predictions. Some of these attributes could be more complex than the ones we have used here, such as for example an analysis of the communication structure between stakeholders [Wolf et al., 2009], the roles of the users involved in a discussion, or the system components affected by a feature request. We have also performed experiments where we generated and evaluated prediction models from combinations of

the attributes and found that the results were not significantly improved and in some case even performed worse than when the attributes were taken in isolation. Combining these attributes more intelligently, however, such as via the use of a feature selection method [Guyon and Elisseeff, 2003] could yield better performance.

### 5.5.3 Threats to Validity

#### Construct Validity

In this context construct validity refers to whether the theoretical concepts in the experiments are interpreted and used correctly.

The accuracy of the expected value measure is dependent on the inputs to the cost model. In this study estimation is subjective for the cost of a failure relative to the cost of upfront requirements analysis, and the probability that this analysis will avoid a failure. We have shown, however, that expected value remains positive for a wide uncertainty in these estimations thus mitigating against this threat.

The failure prediction approach relies on the automatic identification of failures using the rules defined in Section 3.3.1, which poses a threat to validity if some failures are incorrectly identified. The manual verification of failures in 100 automatically identified feature requests for each type of failure (discussed at the end of Section 3.3.2) mitigates against this risk, and we have not included the Eclipse project in the experiments due to the poor reliability of automatic failure identification. This threat, however, is one that should be taken into consideration when considering the early failure prediction approach for a feature request management system.

Inconsistent updates in the meta-data of feature requests leads to many failures not being identified in historical datasets (as discussed at the end of Section 3.3.2). While prediction models constructed with missing data are consistent, we expect that more value could be obtained from predicting failures in projects with a strong culture of consistently updating their meta-data.

### Internal Validity

In this context internal validity refers to whether there are flaws in the design of the study, and whether the results presented follow from the data.

The internal validity of the experiments could be called into question since the same set of feature requests are used to generate and validate predictive models. Could, for example, the selection of an alert threshold that yields the highest expected value as per step 4 of model generation in Section 5.2.1 be selecting a model that only performs well on the exact dataset it is built upon? Could the size of the datasets and their failure densities be inadequate for generating a model that can be statistically expected to perform equivalently on a similar dataset?

Sanity checks were performed to confirm the internal validity of the evaluation approach using blind training and testing sets. Figure 5.8 shows the results of such an experiment on stalled development failures in the KDE project. From an original dataset of 1,800 feature requests 1,000 were chosen at random to build and evaluate a prediction model $\lambda$. The first column in Figure 5.8 shows the expected value, and the precision and recall of prediction model $\lambda$ obtained in the same fashion as the results in Section 5.5.2. The remaining 800 feature requests were then split into 10 datasets, and prediction model $\lambda$ was used to generate predictions for each set. The expected value, precision and recall for these experiments are shown in the following 10 columns, followed by an average of these results across the 10 test sets.

An almost identical expected value, precision and recall to the ten fold cross validation result were obtained on the test cases. This blind testing of the failure prediction approach on a set of feature requests not used to generate a prediction model suggests internal validity.

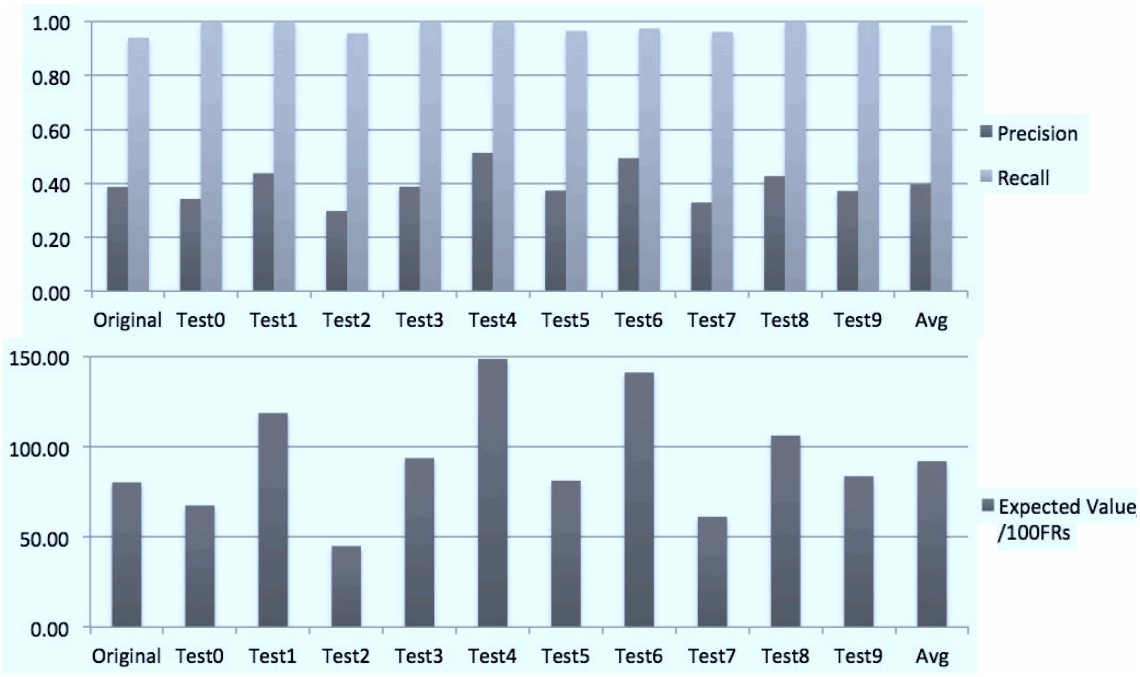



Figure 5.8: Blind Testing for Stalled Development Failures in the KDE Project

### External Validity

In this context external validity refers to whether the results of these experiments apply to the wider context of feature request management systems.

The entire dataset of six real large-scale feature request management systems were used in the validation of the early failure prediction approach, strongly suggesting external validity. Concept drift (in which the classification of feature requests that fail changes over time) may affect the benefits of the approach when the prediction models are put to practical use in feature request management systems. Users' behaviour will change of its own accord and in reaction to the introduction of the early failure prediction system, rendering prediction models less accurate. An online learning technique [Widmer and Kubat, 1996] could be employed to counter this effect.

The experiments reported assume independence of failures. In practice, however, performing upfront analysis on a set of feature requests predicted to have a failure of type $F_1$ could effect the rate of failure of type $F_2$; thereby altering the precision, recall and

expected value of a prediction model for failure $F_2$. An assessment of the direction, size, and implications of these effects could be made by measuring subsequent changes to all failure densities over time after one set of failure predictions has been acted upon.

## 5.6 Related Work

There is a large volume of work on predicting failures at later stages of development: techniques have been proposed to predict the location of code defects from source code metrics [Nagappan et al., 2006, Zimmermann et al., 2007] and the textual content of code [Zeller et al., 2011], to predict build failures from the communication structure between system developers (who communicated with who) [Wolf et al., 2009], and to predict the system reliability from test case results [Musa, 2004]. In contrast, we study the extent to which it is possible to predict failures from discussions much earlier in the development process before a decision is made on whether to implement a feature. Further, our automated predictions extend the usual product failures to cover process level failures.

Failure predictions earlier in the life-cycle can be made using a causal model that aims at predicting the number of defects that will be found during testing and operational usage based on a wide range of qualitative and quantitative factors concerning a project (such as its size, the regularity of specification reviews, the level of stakeholder involvement, the scale of distributed communication, programmer ability, etc.) [Fenton et al., 2008] [Madachy and Boehm, 2008]. In contrast, we aim at predicting failures in projects that may have a less disciplined approach than those for which this causal model has been designed, and we aim to be able to identify which specific feature requests are most at risk rather than predicting the overall number of failures. Further, the use of a classification algorithm is fully automated and does not require the human expertise needed to build a causal model.

There is a large body of work on predictive techniques that, like our approach, use a cost model for making trade-offs between precision and recall when the loss associated to false positive and false negatives are asymmetric [Granger, 1969] [Kaufhold et al., 2006]. These

papers consider general techniques for developing and evaluating predictive models in this context. Our work is an application of such techniques for early failure predictions on feature requests. Our approach uses the cost model *a-posteriori* to find an optimal trade-off between precision and recall. In some contexts, methods that take cost into account during the construction of the prediction model have been shown to perform better than a-posteriori approaches [Abrahams et al., 2005]. Such a technique might be used to try to improve the failure prediction results presented.

A loose comparison of our results can be made to failure predictions later in the software development life-cycle using the measures of precision and recall. The precision and recall measure we obtain in our experiments (ranging approximately between 0.3 and 0.9 recall and between 0.2 and 0.7 precision) are of the same order as those obtained by the later predictions, such as in Wolf et al.'s paper predicting build failures from the communication structure in bug tracking systems at build time with a recall between 0.55 and 0.75 and a precision between 0.5 and 0.76 [Wolf et al., 2009]. We cannot, however, draw solid conclusions from such comparisons as they are based on studies involving different projects, failure types and failure densities.

The attributes used to generate predictive models have been used in similar contexts. It has been suggested that textual structuring of a bug or feature description can be an indicator of later defects [Bettenburg et al., 2008], and that the attributes relating to the social network of stakeholders has been used to predict build failures [Wolf et al., 2009]. While the textual predictors performed well in our results, those relating to the actors contributed to discussions did not. Using more complex attributes on the social aspects of collaboration, such as whether discussion participants have contributed many times before, may yield better results.

## 5.7 Chapter Summary

In this Chapter we have presented a tool-supported framework that constructs and evaluates early failure prediction models using data that already exists in feature request

management systems. This framework can be used to predict the process and product failures we have defined in Section 3.3 at the point in time when a developer or project manager makes a decision to reject a feature or assign it for implementation.

We have presented a novel cost-benefit model which allows us make a quantified estimate of whether acting upon a set of failure predictions will provide value to a project. This contrasts related work where precision and recall are used to assess the quality of predictions. Further, the model suggests that the random baseline used to benchmark predictive models in related work never provides more value than the most optimistic and most pessimistic baselines.

We have presented experiments assessing the expected value, recall and precision of predictive models generated using the early failure prediction approach on six large scale projects. These results suggest that a project's costs could be reduced by performing additional upfront requirements analysis on feature requests predicted to fail for many of the projects and failure types, showing promise for the approach. The most effective prediction models were generated with the M5P-tree algorithm using predictive attrbiutes based on the textual content of early discussions, and the types of failure most susceptible to prediction were found to be Stalled Development and Abandoned Implementation.

The approach described in this Chapter is a promising solution to a key challenge in feature request management systems: Determining where, how, and how much upfront requirements analysis should take place. The approach provides valuable information suggesting which features are at risk of which types of failure, and thereby where and how upfront requirements analysis effort should be focused. Further, the output from the cost-model provides information on whether it is worth the effort to perform this additional upfront analysis.

# 6 Conclusion

*In this thesis we have proposed and evaluated two distinct tool-supported solutions to key challenges in feature request management systems. Firstly, a structured defect-oriented argumentation framework that encourages stakeholders to fully explore their needs, and makes these needs more understandable to developers. Second, an early failure prediction framework that informs developers which feature requests require additional upfront requirements analysis, what types of analysis to perform, and whether performing this analysis will reduce costs in a project. Moreover, these approaches are grounded on studies into the types of requirements defects and failures specific to feature request management systems that provide new information on the errors that occur in these systems.*

## 6.1 Contributions

An increasing number of software development projects rely on online feature request management systems to elicit, analyse and manage users' change requests [Bird et al., 2008] [Damian, 2004a]. Requirements defects and failures in feature requests lead to inadequate functionalities being developed, costly changes, and wasted development effort. Consequently, there are strong benefits if early on in a feature requests' life cycle requirements defects can be detected and resolved, and the likelihood of later failures occurring were known.

We have explored and studied defects and failures in feature request management systems. We define a taxonomy of process and product failures that occur in feature requests and can be automatically traced from existing historical meta-data. A study of these failures across seven large scale projects shows that they occur in abundance and incur significant costs. A study of defects identified by stakeholders in the Firefox project suggests they differ from those of existing taxonomies, and are especially focused on identifying inadequacies in suggested features.

We have presented a tool-supported defect-oriented argumentation framework that aims to help stakeholders express and explore their needs, and help developers better understand these needs. A preliminary evaluation of the framework suggests that it requires little additional effort to use, and that it captures the key discussion concepts on whether to implement a given feature.

We have presented a tool-supported framework for generating and evaluating early failure prediction models from historical feature request data. Project managers and developers can use these predictions to reduce project costs by performing additional upfront requirements analysis on feature requests with a high probability of failure before taking a decision on whether to implement a feature.

We have defined a cost-benefit model for evaluating whether the benefits of acting upon a set of early failure predictions will outweigh the costs - thereby determining whether value to a project can be expected from the use of a given failure prediction model. This provides a more meaningful evaluation of prediction models than the standard measures of recall and precision. The cost-benefit equation also reveals that benchmarking against two baselines that make all positive and all negative predictions is more meaningful than the random predictor (often used baseline in failure prediction research).

We have presented experiments on predicting failures in seven large scale projects, which strongly suggest the early failure prediction approach to be a useful strategy for guiding upfront requirements analysis in a feature request management system and reducing the software development costs. The results also indicate the types of failure that may be more susceptible to early predictions and the characteristics of feature request discussions that act as good predictors.

## 6.2 Limitations and Perspectives

### 6.2.1 The Do-Argue Framework

A preliminary evaluation defect-oriented argumentation framework suggests that it encapsulates the key concepts of early requirements discussions on whether to implement a feature, and requires minimal additional effort to use. To fully validate the framework, however, an evaluation of user adoption and whether the software developed better suits stakeholders' needs is required. It would be difficult to convince stakeholders in a real project to adopt the framework, and we therefore decided to explore a more promising approach to decision making in feature request management systems that became apparent during the development of DoArgue; the early failure prediction framework.

When designing DoArgue we focused on minimising the additional effort required to use the framework in a feature request management system. Stakeholders do still, however, need to annotate discussions which could lead to low adoption when put to use in a real project. The structuring of discussions could also exacerbate this problem since open, free discussion is at the core of development in feature request management systems. A machine learning approach could be used to automate, or partially automate, the process of annotating discussions by finding phrases that are key indicators of concepts in the DoArgue framework.

The research could be carried forward by further validating and developing DoArgue with the use of controlled experiments in a real feature request management system. Further iterations of development should aim to maximise adoption (measured via stakeholder participation levels), and the frameworks' benefits (which could be measured through a survey asking stakeholders whether they feel that their needs are better understood and how well the finished product fits their needs). The design rationale community have extensively researched solutions to the problem of structuring complex collaborative discussions to make them more effective and understandable. Further development of the DoArgue framework could potentially transfer the lessons learned from design rationale research into the wider context of stakeholder-lead requirements engineering.

### 6.2.2 The Early Failure Prediction Framework

We have presented an extensive validation of the early failure prediction framework on seven large scale projects which strongly suggests that prediction models can be generated that would provide value to projects. The tool implementation, Intueri, could now be integrated into a feature request management system so that further evaluations can assess how well it performs in real projects.

The issue of concept drift (in which the strength of predictors change over time) is likely to decrease the accuracy of prediction models; users' behaviour will change naturally and in response to the presence of the prediction framework. Online learning techniques [Widmer and Kubat, 1996] should be explored and integrated into the approach to counter this effect.

The technique identifies correlations between the characteristics of feature request discussions and failures, but it does not explain the causes of failure. For example, the predictive model generated for Netbeans Stalled Development failures decreases the probability of failure as the percentage of posts made by the developer assigned to a feature request decreases. Interpreting such a result involves a high degree of speculation - it could be that the risk of failure decreases as the assigned developer contributes less, or as more stakeholders contribute to a discussion. Further, a direct causal relationship is not necessarily implied. With care given to this caveat further analysis of prediction models could provide a basis for exploring best practice recommendations in requirements discussions.

Predicting failures based on the requirements defects present in a feature request would be an interesting extension to this work that could provide information explaining the causes of failure. One way to make information about requirements defects available would be to use the outputs from natural language requirements analysis tools [Gnesi et al., 2005] [Kiyavitskaya et al., 2008] [Verma and Kass, 2008]. Another approach would be to use manually tagged defects from the DoArgue framework presented in Chapter 4.

We suspect that significant improvements in the performance of prediction models can still be achieved. This could be done by using a richer set of predictive attributes than the

restricted set that has been successfully used in this thesis. These include the presence of rationale in a feature request discussion (which could be indicated by common intentional phrases), advanced natural language processing techniques [Sebastiani, 2002], the architectural components potentially affected by a feature request, or the roles, expertise, and communication structure of the persons involved in the discussion. Techniques also exist for manipulating and cleaning a dataset before passing it to a classification algorithm to generate a prediction model that could improve accuracy, such as for example feature selection [Guyon and Elisseeff, 2003] whereby predictive attributes that are not powerful predictors are removed. Integrating a cost-model into a classification algorithm has been shown to improve its benefits [Fan et al., 1999]; consequently integrating the cost model from Section 5.1 into a classification algorithm, as opposed to using it to select an alert threshold, might also yield more expected value.

The cost model from Section 5.1.1 could be extended to validate failure prediction models in terms of the benefits of acting upon a set of failure predictions. Predictive modelling research typically performs evaluations based on the precision and recall measures; without directly evaluating whether the cost of actioning a set predictions will outweigh the benefits. Adapting the cost model to other types of predictive modelling in software engineering would be of significant value to researchers by grounding their validation in an economic context.

The five process and product failures that can be automatically identified in a body of feature requests could be put to many alternative uses by the data mining and machine learned research community. The presence of process failures, for example, could be used to predict system reliability (as an alternative to existing predictors such as test-case results [Musa, 2004] and source code metrics [Zimmermann et al., 2007]). Failure could also be used as a measure of software development process maturity in feature request management systems.

We have shown the early failure prediction framework to be a practical, useful tool for providing information on which feature requests are in need of attention and whether performing requirements analysis on them will reduce project costs. The approach could

be applied to any software development process where historical data is available on failures that can be traced back to requirements activity, providing important information quantifying the benefits of performing specific requirements analysis actions.

# A Firefox Feature Request

Figure A.1 shows the meta-data of feature request 171702[1] in the Firefox project at the point in time when it had been integrated into the product. Figures A.2 through A.4 show the chronological discussion thread for this feature request and are annotated with a cloud bubble where changes were made to the feature request's data to reflect its progress though the life-cycle defined in Section 3.3.1. Finally, in Figure A.6 shows the history of changes made the feature request's meta data over its development.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=171702

First Last Prev Next  This bug is not in your last search results.

**Bug 171702 - Add support for a <toolbar> to the right of the menu bar** Last Comment

| | | | |
|---|---|---|---|
| **Status:** | VERIFIED FIXED | **Reported:** | 2002-09-30 07:54 PDT by djk |
| **Whiteboard:** | | **Modified:** | 2006-11-13 07:25 PST (History) |
| **Keywords:** | | **CC List:** | 15 users (show) |
| **Product:** | Firefox | **See Also:** | |
| **Component:** | Toolbars | | |
| **Version:** | unspecified | **Crash Signature:** | |
| **Platform:** | x86 Windows 98 | | |
| **Importance:** | -- enhancement with 2 votes (vote) | | |
| **Target Milestone:** | Phoenix0.4 | | |
| **Assigned To:** | David Hyatt | | |
| **QA Contact:** | toolbars | | |
| **URL:** | | | |
| **Duplicates:** | ~~172516~~ (view as bug list) | | |
| **Depends on:** | | | |
| **Blocks:** | | | |
| | Show dependency tree / graph | | |

| Attachments | | |
|---|---|---|
| **Creates an extra toolbar to the right of the menu bar.** (8.96 KB, patch) 2002-10-03 10:34 PDT, Darren Salt | *no flags* | Details \| Diff \| Splinter Review |
| **Function to (quickly) find UI elements of a given type; depth-limited** (490 bytes, text/plain) 2002-10-06 04:03 PDT, Darren Salt | *no flags* | Details |
| Add an attachment (proposed patch, testcase, etc.) | | Show Obsolete (1) View All |

Figure A.1: Example Feature Request: Meta-data

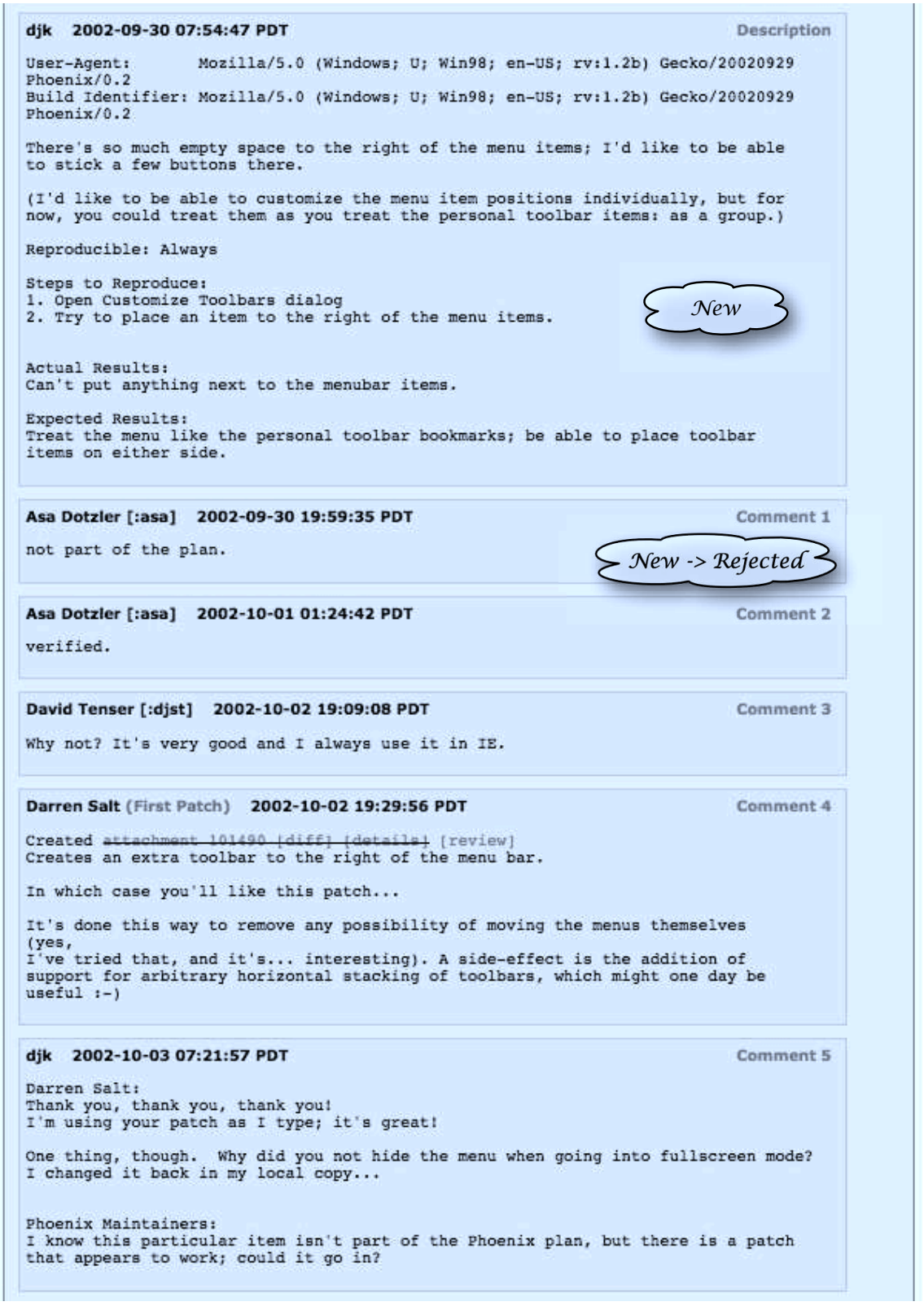



Figure A.2: Example Feature Request: Opened and Rejected

**David Tenser [:djst] 2002-10-03 09:57:54 PDT** Comment 6

Please check this in, it's great! I want to be able to have the menu, the
toolbar and the address field on the same row. It's the way I'm used to have it
in IE, so why should Phoenix not let me do it? Very tempted to reopen...

**Darren Salt (First Patch) 2002-10-03 10:34:45 PDT** Comment 7

Created attachment 101551 [diff] [details] [review]
Creates an extra toolbar to the right of the menu bar.

Bug fixes, basically, but this is the full patch.

* No longer leaves the menu bar visible in full-screen mode.
* No longer causes Phoenix to hang on closing the toolbars customisation window

  if there are empty toolbars.

**Asa Dotzler [:asa] 2002-10-04 09:37:33 PDT** Comment 8

Darin, this sounds like a fine mozdev project. I encourage you to get set up
over there and use their infrastructure rather than ours for the development of
this extension.

This issue is closed. The Module Owner has made a decision and this request will
not be implemented in Phoenix.

(preemptive warning for anyone coming from mozillaZine where it was suggested
that enough people reopening or voting for this bug might help it get
incorporated into Phoenix: Don't. The Module Owner has said this won't happen.
Reopening this feature request will be considered abuse of our system which
could lead to your Bugzilla account being disabled. This isn't a democracy and
the Module Owner has spoken.)

**David Tenser [:djst] 2002-10-04 09:59:18 PDT** Comment 9

It would be a lot easier for us to understand why this fine patch will not get
checked in if The Module Owner could give us another reason than "not part of
the plan".

**David Hyatt 2002-10-04 11:07:06 PDT** Comment 10

(1) Performance - customizing of menus would be very slow and would dramatically
impact New Window time, since unlike toolbar widgets they have a lot of sub-content.

(2) Mac - XUL is still a cross-platform language, and the <menubar> needs to be
kept as a separate entity. If we moved <menu>s into the palette and just built
up the <menubar> like a <toolbar>, then we violate that rule.

**djk 2002-10-04 11:32:22 PDT** Comment 11

Since I don't fully understand the attached patch or XUL, I may be wrong, but:
I think the patch only creates a new toolbar to the right of the menubar. I
don't believe that it enables menu editing or moving the menu itself.

If I understand David's critisism of this request, those are the two things
stopping this bug.

I'm sorry for including those two items in my initial bug comment.
All I really want is what the patch implements: another toolbar to the right of
the menu.

If I'm wrong about this, I'll just shut up and keep on patching my own builds. :-}

Figure A.3: Example Feature Request: Stakeholder Negotiation

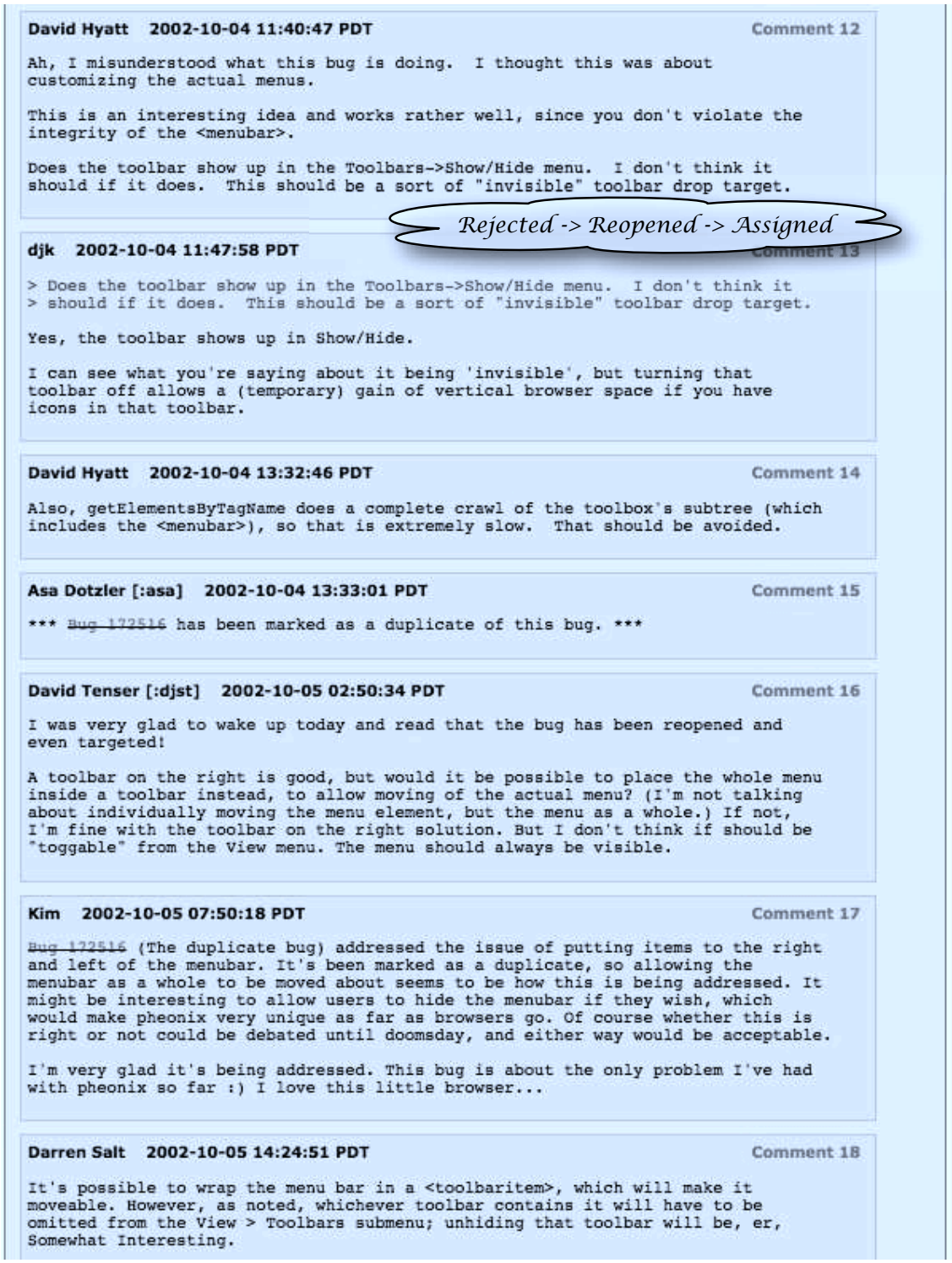



Figure A.4: Example Feature Request: Re-opened and Assigned

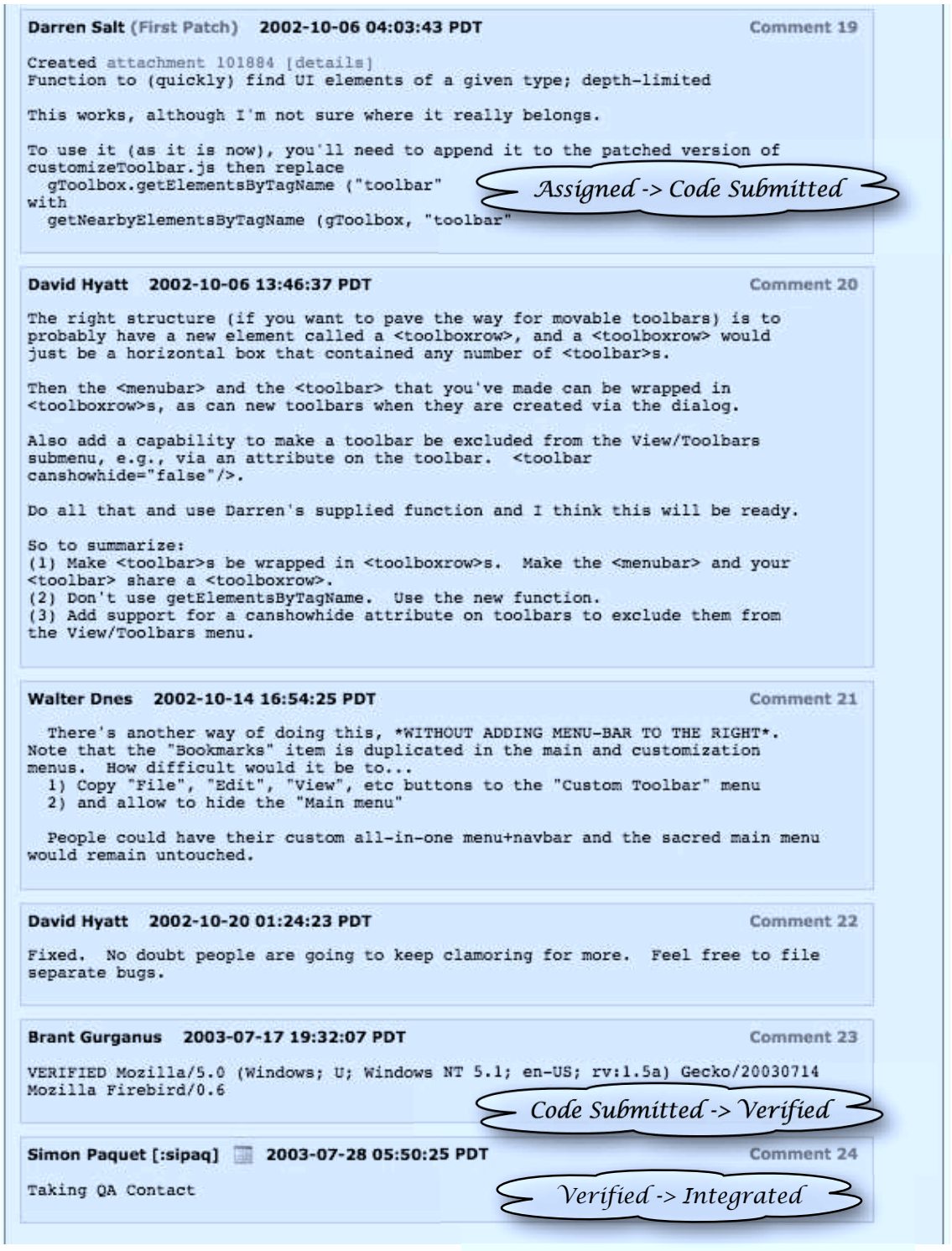



Figure A.5: Example Feature Request: Implemented and Integrated

| Who | When | What | Removed | Added |
|---|---|---|---|---|
| asa | 2002-09-30 19:59:35 PDT | Status | UNCONFIRMED | RESOLVED |
| | | Resolution | --- | WONTFIX |
| | | Last Resolved | | 2002-09-30 19:59:35 |
| asa | 2002-10-01 01:24:42 PDT | Status | RESOLVED | VERIFIED |
| djst | 2002-10-02 19:09:08 PDT | CC | | david.tenser |
| linux | 2002-10-03 10:34:45 PDT | Attachment #101490 Attachment is obsolete | 0 | 1 |
| moz | 2002-10-03 12:28:55 PDT | CC | | moz |
| hyatt | 2002-10-04 11:40:47 PDT | Status | VERIFIED | UNCONFIRMED |
| | | Resolution | WONTFIX | --- |
| hyatt | 2002-10-04 11:41:58 PDT | Status | UNCONFIRMED | ASSIGNED |
| | | Ever confirmed | | 1 |
| | | Target Milestone | --- | Phoenix0.3 |
| asa | 2002-10-04 13:32:58 PDT | CC | | magpie |
| hyatt | 2002-10-06 13:58:31 PDT | Summary | menu bar needs to be a target for customize toolbars | Add support for a <toolbar> to the right of the menu bar |
| hyatt | 2002-10-06 19:01:58 PDT | Target Milestone | Phoenix0.3 | Phoenix0.4 |
| hyatt | 2002-10-20 01:24:23 PDT | Status | ASSIGNED | RESOLVED |
| | | Resolution | --- | FIXED |
| | | Last Resolved | 2002-09-30 19:59:35 | 2002-10-20 01:24:23 |
| brant | 2003-07-17 19:32:07 PDT | Status | RESOLVED | VERIFIED |
| bugzilla | 2003-07-28 05:50:25 PDT | QA Contact | asa | bugzilla |
| mnyromyr | 2003-07-29 12:28:54 PDT | CC | | kd-moz |
| bugzilla | 2006-11-13 07:25:01 PST | QA Contact | bugzilla | toolbars |

Figure A.6: Example Feature Request: Change History

# B Intueri Screenshots

The Figures in this Appendix show screenshots from the process of generating prediction models using Intueri - the tool implementation of our prediction framework described in Chapter 5. Figure B.1 shows the raw discussion data extracted from a bugzilla feature request management system in XML format. Figure B.2 shows the Intueri front-end used to prepare training sets from the raw discussion data. Figure B.3 shows a training set in the ARFF file format which can be given to the WEKA tool to generate prediction models. Figure B.4 shows a matlab analysis of a set predictions taken from the output of WEKA.

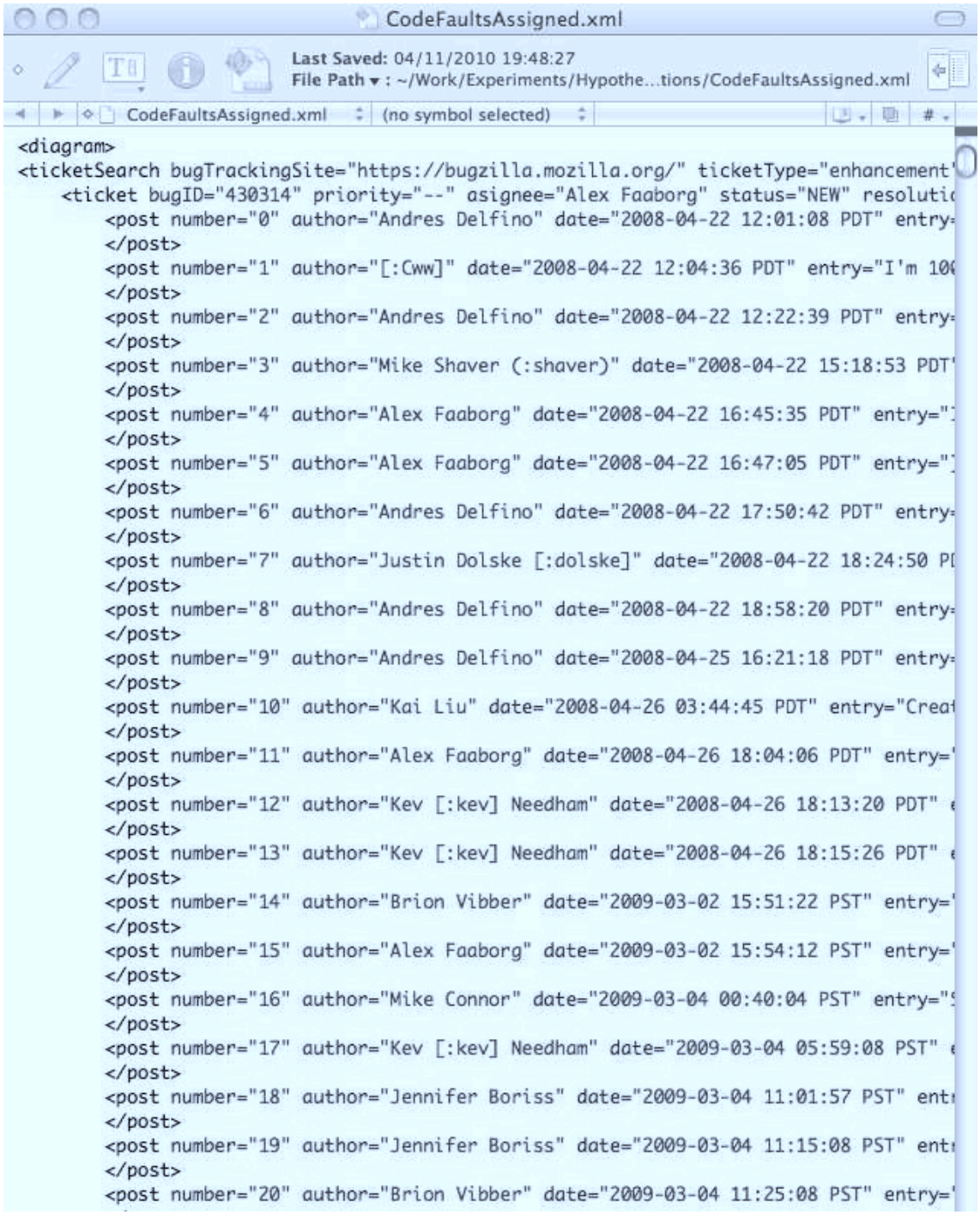



Figure B.1: Raw Feature Requests in XML

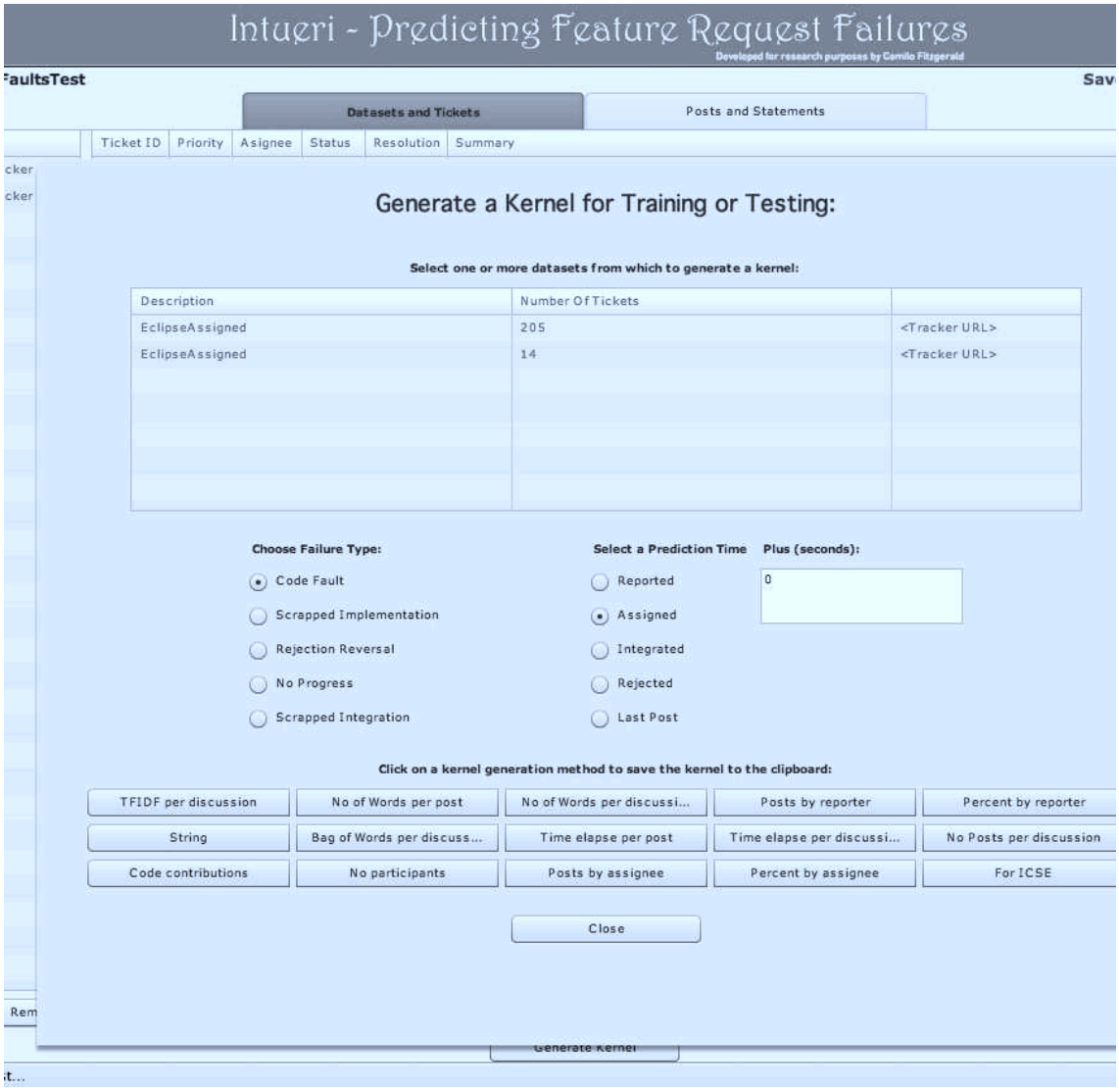



Figure B.2: Intueri Front-end


Figure B.3: ARFF File Training Set

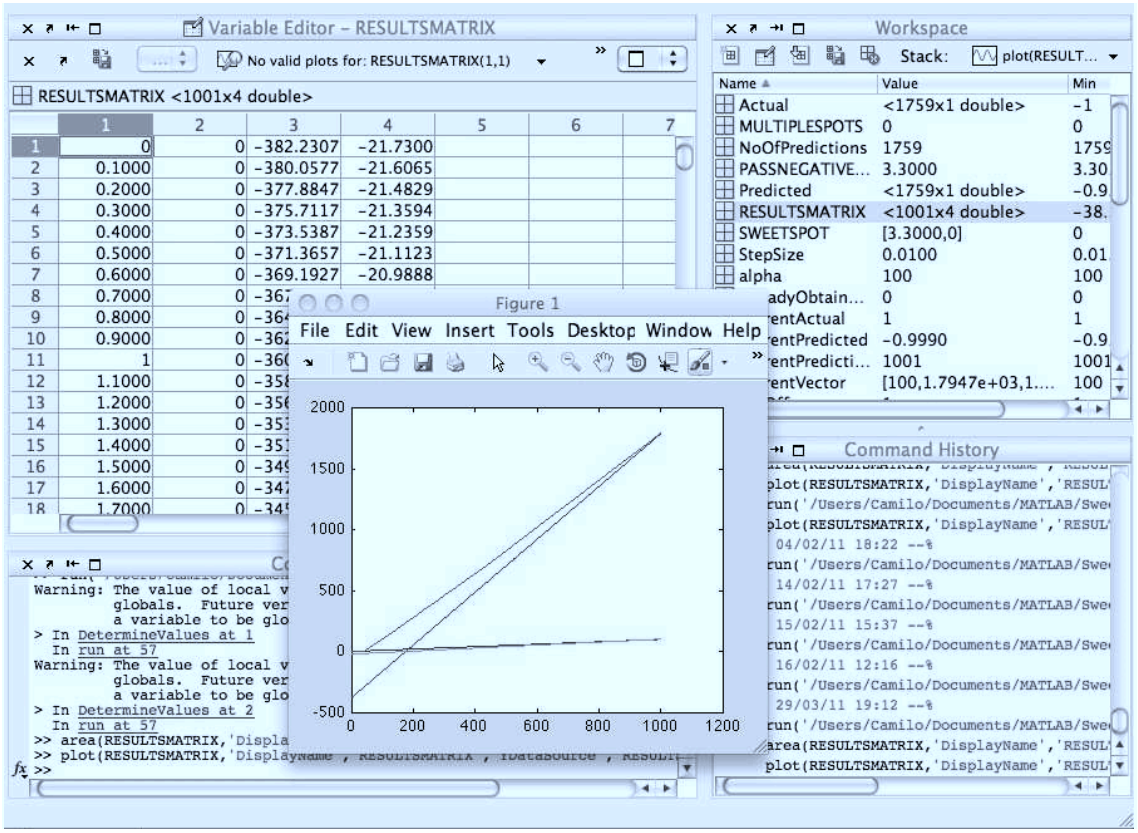



Figure B.4: Matlab Evaluation of Prediction Models

# Bibliography

[Abeti et al., 2009] Abeti, L., Ciancarini, P., and Moretti, R. (2009). Wiki-based requirements management for business process reengineering. In *Proceedings of the ICSE Workshop on Wikis for Software Engineering*, pages 14–24.

[Abrahams et al., 2005] Abrahams, A. S., Becker, A., Fleder, D., and MacMillan, I. C. (2005). Handling generalized cost functions in the partitioning optimization problem through sequential binary programming. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 3–9, Washington, DC, USA. IEEE Computer Society.

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA. ACM.

[Bell and Thayer, 1976] Bell, T. E. and Thayer, T. A. (1976). Software requirements: Are they really a problem? In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 61–68, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Berry et al., 2005] Berry, D. M., Damian, D., Finkelstein, A., Gause, D., Hall, R., and Wassyng, A. (2005). To do or not to do: If the requirements engineering payoff is so good, why aren"t more companies doing it? In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, page 447, Washington, DC, USA. IEEE Computer Society.

[Bettenburg et al., 2008] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA. ACM.

[Bird et al., 2008] Bird, C., Pattison, D., D'Souza, R., Filkov, V., and Devanbu, P. (2008). Latent social structure in open source projects. In *Proceedings of the 16th ACM SIG-*

*SOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA. ACM.

[Boehm and Turner, 2003] Boehm and Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Boehm et al., 1995] Boehm, B., Bose, P., Horowitz, E., and Lee, M. J. (1995). Software requirements negotiation and renegotiation aids. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 243–253, New York, NY, USA. ACM.

[Boehm et al., 1988] Boehm, B., Papaccio, P., Inc, T., and Beach, R. (1988). Understanding and controlling software costs. *IEEE Transactions on software engineering*, 14(10):1462–1477.

[Boehm and Papaccio, 1988] Boehm, B. W. and Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477.

[Briand, 2008] Briand, L. C. (2008). Novel applications of machine learning in software testing. In *Proceedings of the 8th International Conference on Quality Software*, volume 0, pages 3–10, Los Alamitos, CA, USA. IEEE Computer Society.

[Brykczynski, 1999] Brykczynski, B. (1999). A survey of software inspection checklists. *SIGSOFT Software Engineering Notes*, 24(1):82.

[Carver et al., 2008] Carver, J., Nagappan, N., and Page, A. (2008). The impact of educational background on the effectiveness of requirements inspections: An empirical study. *IEEE Transactions on Software Engineering*, 34(6):800–812.

[Cheng and Atlee, 2007] Cheng, B. H. C. and Atlee, J. M. (2007). Research directions in requirements engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 285–303, Washington, DC, USA. IEEE Computer Society.

[Cleland-Huang et al., 2010] Cleland-Huang, J., Czauderna, A., Gibiec, M., and Emenecker, J. (2010). A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 155–164, New York, NY, USA. ACM.

[Cleland-Huang et al., 2009] Cleland-Huang, J., Dumitru, H., Duan, C., and Castro-Herrera, C. (2009). Automated support for managing feature requests in open forums. *Communications of the ACM*, 52(10):68–74.

[Conklin and Begeman, 1988] Conklin, J. and Begeman, M. L. (1988). gibis: a hypertext tool for exploratory policy discussion. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, CSCW '88, pages 140–152, New York, NY, USA. ACM.

[Damian, 2004a] Damian, D. (2004a). An exploratory study of facilitation in distributed requirements engineering. *Requirements Engineering*, 8(1):23–41.

[Damian, 2004b] Damian, D. (2004b). RE challenges in multi-site software development organisations. *Requirements Engineering*, 8(3):149–160.

[Decker et al., 2007] Decker, B., Ras, E., Rech, J., Jaubert, P., and Rieth, M. (2007). Wiki-based stakeholder participation in requirements engineering. *IEEE Software*, 24(2):28–35.

[Fagan, 1976] Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211.

[Fan et al., 1999] Fan, W., Stolfo, S. J., Zhang, J., and Chan, P. K. (1999). Adacost: Misclassification cost-sensitive boosting. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 97–105, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Fenton et al., 2007] Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., and Mishra, R. (2007). Predicting software defects in varying development lifecycles using bayesian nets. *Information and Software Technology*, 49(1):32–43.

[Fenton et al., 2008] Fenton, N., Neil, M., Marsh, W., Hearty, P., Radliński, Ł., and Krause, P. (2008). On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537.

[Finkelstein and Fuks, 1989] Finkelstein, A. and Fuks, H. (1989). Multiparty specification. *ACM SIGSOFT Software Engineering Notes*, 14(3):185 – 195.

[Fitzgerald et al., 2011] Fitzgerald, C., Letier, E., and Finkelstein, A. (2011). Early failure prediction in feature request management systems. In *Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference*, RE '11, pages 229–238, Washington, DC, USA. IEEE Computer Society.

[Fitzgerald et al., 2012] Fitzgerald, C., Letier, E., and Finkelstein, A. (2012). Early failure prediction in feature request management systems: an extended study. *Requirements Engineering*, 17:117–132.

[Freeman, 1984] Freeman, R. (1984). *Strategic Management: A stakeholder approach*. Pitman, Boston.

[Freimut et al., 2005] Freimut, B., Denger, C., and Ketterer, M. (2005). An industrial case study of implementing and validating defect classification for process improvement and quality management. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, METRICS '05, pages 19–, Washington, DC, USA. IEEE Computer Society.

[Gnesi et al., 2005] Gnesi, S., Lami, G., Trentanni, G., Fabbrini, F., and Fusani, M. (2005). An automatic tool for the analysis of natural language requirements. *International Journal of Computer Systems Science and Engineering*, 20(1):53–62.

[Granger, 1969] Granger, C. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica: Journal of the Econometric Society*, pages 424–438.

[Guyon and Elisseeff, 2003] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. (2009). The weka data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

[Hayes, 2003] Hayes, J. H. (2003). Building a requirement fault taxonomy: Experiences from a nasa verification and validation research project. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, page 49, Washington, DC, USA. IEEE Computer Society.

[Herbsleb and Moitra, 2001] Herbsleb, J. and Moitra, D. (2001). Global software development. *Software, IEEE*, 18(2):16–20.

[Kaufhold et al., 2006] Kaufhold, J., Abbott, J., and Kaucic, R. (2006). Distributed cost boosting and bounds on mis-classification cost. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, pages 146–153, Washington, DC, USA. IEEE Computer Society.

[Kiyavitskaya et al., 2008] Kiyavitskaya, N., Zeni, N., Mich, L., and Berry, D. M. (2008). Requirements for tools for ambiguity identification and measurement in natural language requirements specifications. *Requirements Engineering*, 13(3):207–239.

[Klein et al., 2006] Klein, G., Moon, B., and Hoffman, R. (2006). Making sense of sensemaking 1: Alternative perspectives. *Intelligent Systems, IEEE*, 21(4):70–73.

[Klein and Iandoli, 2008] Klein, M. and Iandoli, L. (2008). Supporting collaborative deliberation using a large-scale argumentation system: The mit collaboratorium. *MIT Sloan Research Paper No. 4691-08. Available at SSRN: http://ssrn.com/abstract=1099082 or http://dx.doi.org/10.2139/ssrn.1099082.*

[Knauss et al., 2009] Knauss, E., Brill, O., Kitzmann, I., and Flohr, T. (2009). SmartWiki: support for high-quality requirements engineering in a collaborative setting. In *Proceedings of the ICSE Workshop on Wikis for Software Engineering*, pages 25–35.

[Lamsweerde, 2009] Lamsweerde, A. (2009). Reasoning about alternative requirements options. In *Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos*, pages 380–397. Springer-Verlag.

[Laurent and Cleland-Huang, 2009] Laurent, P. and Cleland-Huang, J. (2009). Lessons learned from open source projects for facilitating online requirements processes. In *Proceedings of the 15th International Working Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ '09, pages 240–255, Berlin, Heidelberg. Springer-Verlag.

[Laurent et al., 2007] Laurent, P., Cleland-Huang, J., and Duan, C. (2007). Towards automated requirements triage. In *Proceedings of the 15th IEEE International Conference on Requirements Engineering*. IEEE Computer Society.

[Lim et al., 2011] Lim, S. L., Damian, D., and Finkelstein, A. (2011). Stakesource2.0: using social networks of stakeholders to identify and prioritise requirements. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1022–1024, New York, NY, USA. ACM.

[Macdonald and Miller, 1999] Macdonald, F. and Miller, J. (1999). A comparison of computer support systems for software inspection. *Automated Software Engineering*, 6(3):291–313.

[Mackenzie, 1984] Mackenzie, J. (1984). No logic before friday. *Synthese*, 58(2).

[Madachy and Boehm, 2008] Madachy, R. and Boehm, B. (2008). Assessing quality processes with odc coqualmo. In *Proceedings of the Software process, 2008 international conference on Making globally distributed software development a success story*, ICSP'08, pages 198–209, Berlin, Heidelberg. Springer-Verlag.

[McConnell, 2004] McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[Meyer, 2008] Meyer, B. (2008). Design and code reviews in the age of the internet. *Communications of the ACM*, 51(9):66–71.

[Musa, 2004] Musa, J. D. (2004). *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Authorhouse.

[Nagappan et al., 2006] Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, page 461. ACM.

[Nuseibeh and Easterbrook, 2000] Nuseibeh, B. and Easterbrook, S. (2000). Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46, New York, NY, USA. ACM.

[Ontañón and Plaza, 2008] Ontañón, S. and Plaza, E. (2008). An argumentation-based framework for deliberation in multi-agent systems. *Lecture Notes in Computer Science*, 4946:178–196.

[Parnas and Weiss, 1985] Parnas, D. L. and Weiss, D. M. (1985). Active design reviews: principles and practices. In *Proceedings of the 8th international conference on Software engineering*, pages 132–136, London, England. IEEE Computer Society Press.

[Porter et al., 1995] Porter, A., Votta Jr, L., and Basili, V. (1995). Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575.

[Riechert and Berger, 2009] Riechert, T. and Berger, T. (2009). Leveraging semantic data wikis for distributed requirements elicitation. In *Proceedings of the ICSE Workshop on Wikis for Software Engineering*, pages 7–13.

[Schmidt et al., 2001] Schmidt, R., Lyytinen, K., Keil, M., and Cule, P. (2001). Identifying software project risks: An international delphi study. *Journal of management information systems*, 17(4):5–36.

[Sebastiani, 2002] Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47.

[Shull et al., 2002] Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., and Zelkowitz, M. (2002). What we have learned about fighting defects. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, page 249, Washington, DC, USA. IEEE Computer Society.

[Shum, 1996] Shum, S. B. (1996). Design argumentation as design rationale. *The Encyclopedia of Computer Science and Technology*, pages 95–128.

[Shum, 2008] Shum, S. B. (2008). Cohere: Towards web 2.0 argumentation. In *Proceedings of the 2nd International Conference on Computational Models of Argument*.

[Shum et al., 2006] Shum, S. B., Selvin, A. M., Sierhuis, M., Conklin, J., Daw, M., Rowley, A., Juby, B., Michaelides, D., Slack, R., Bachler, M., Mancini, C., Procter, R., de Roure, D., Chown, T., and Hewitt, T. (2006). From gIBIS to MEMETIC: evolving a research vision into a practical tool. *Design Rationale Workshop: Design, Computing & Cognition Conference*.

[Srinivasan and Fisher, 1995] Srinivasan, K. and Fisher, D. (1995). Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, 21:126–137.

[Sun et al., 2010] Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S. (2010). A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM.

[van Lamsweerde, 2001] van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, page 249, Washington, DC, USA. IEEE Computer Society.

[van Lamsweerde, 2009] van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[Verma and Kass, 2008] Verma, K. and Kass, A. (2008). Requirements analysis tool: A tool for automatically analyzing software requirements documents. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 751–763, Berlin, Heidelberg. Springer-Verlag.

[Widmer and Kubat, 1996] Widmer, G. and Kubat, M. (1996). Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101.

[Witten and Frank, 2005] Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Wolf et al., 2009] Wolf, T., Schroter, A., Damian, D., and Nguyen, T. (2009). Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[Yang et al., 2008] Yang, D., Wu, D., Koolmanojwong, S., Brown, A. W., and Boehm, B. W. (2008). Wikiwinwin: A wiki based system for collaborative requirements negotiation. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, HICSS '08, pages 24–, Washington, DC, USA. IEEE Computer Society.

[Zeller et al., 2011] Zeller, A., Zimmermann, T., and Bird, C. (2011). Failure is a four-letter word: a parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 5:1–5:7, New York, NY, USA. ACM.

[Zhang and Tsai, 2003] Zhang, D. and Tsai, J. (2003). Machine learning and software engineering. *Software Quality Journal*, 11:87–119.

[Zimmermann et al., 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, page 9, Washington, DC, USA. IEEE Computer Society.