Leveraging Developer Discussions to Improve Design Accessibility

by

Giovanni Viviani

B., Università della Svizzera Italiana, 2014

M.Sc., University of British Columbia, 2016

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

October 2022

© Giovanni Viviani, 2022
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

**Leveraging Developer Discussions to Improve Design Accessibility**

submitted by Giovanni Viviani in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

**Examinining Committee:**

Gail C. Murphy, Professor, Computer Science, UBC
*Supervisor*

Reid Holmes, Professor, Computer Science, UBC
*Supervisory Committee Member*

Ron Garcia, Professor, Computer Science, UBC
*University Examiner*

Carson Woo, Professor, Commerce and Business Administration, UBC
*University Examiner*

**Additional Supervisory Committee Members:**

Giuseppe Carenini, Professor, Computer Science, UBC
*Supervisory Committee Member*
Abstract

Since the inception of software engineering, the design of a software system has been recognized as one of its most important attributes. A software system’s design determines many of its properties, such as maintainability and performance. One might expect that there is a common and well-established understanding about what software design is and is not. Such an understanding is not evident in the literature, where design has been described in many ways such as large-scale architecture and low-level design patterns, to name just a few. At the same time, an understanding of design is also needed to maintain system properties as changes to the system are made. When developers lose track of the overall design, the system may not conform to its intended properties. Unfortunately, many systems do not have up-to-date design documentation and approaches to recover design often focus on how a system works by extracting structural and behavioural information rather than why it was designed to work like that. In this thesis, we propose an automated approach to extract design information from written discussions between developers. The aim is to make this information accessible to developers, helping them understand the design of a system and make better design decisions. First, we present an interview study we conducted to understand what researchers and practitioners consider as software design. These interviews revealed five recurring topics that can help inform what software design truly represents. We then introduce a classifier able to locate paragraphs in discussions, which we call design points, that pertain to design. Results show that this classifier is able to locate design information with high accuracy even in systems that it was not trained on. We describe a study conducted with software developers that shows that newcomers to a project, when provided with design points relevant to a programming task, are able to
interpret and use the design information to consider additional design alternatives. We fi-
nally discuss an early exploration into the use of semantic frames to identify useful design
points.
Lay Summary

The design of a software system dictates many of its properties. Unfortunately, all too often, design information is poorly recorded, hampering the future evolution of the system. Existing approaches to recover design focus on recovering descriptions of how the software works, ignoring the rationale behind the decisions made when developing it. This thesis aims to extract design information from written discussions between developers as a means of recovering latent design rationale. We discuss interviews with practitioners and researchers to learn how they view software design. We use the findings to describe the development of an automatic tool to locate design information in developer discussions and conduct a study to determine if the recovered information is understandable and useful to developers. We finally describe the application of natural language processing techniques to identify which design information is useful.
Preface

All of the research presented in this thesis is the original intellectual product of the author, Giovanni, with guidance and mentorship from my PhD advisor, Gail C Murphy. Most of this work has been published in various venues.

The work presented in Chapter 3 was published as a paper titled *What really is software design?* [150] at the 2022 *International Conference on Software Analysis, Evolution and Reengineering (SANER)* conference. The paper was also presented at the same conference in March 2022. The study included in this chapter was approved by the University of British Columbia’s Research Ethics Board, certificate #H20-01146.

The work presented in Chapter 4 was done in collaboration with Michalis Famelis, Xin Xia and Calahan Janik-Jones. I remained the lead investigator for the entire project. This work was published as a manuscript titled *Locating Latent Design Information in Developer Discussions: A Study on Pull Requests* [153] in the *IEEE Transactions on Software Engineering* Journal in 2019. This manuscript was also presented as a journal first presentation at the 2022 *International Conference on Software Engineering (ICSE)*. A portion of this work also appeared earlier in a paper titled *What design topics do developers discuss?* [152] at the 2018 *International Conference on Program Comprehension (ICPC)*.

The study included in the work presented in Chapter 5 was approved by the University of British Columbia’s Research Ethics Board, certificate #H21-03850.

The work presented in Chapter 6 was published as a paper titled *Assessing Semantic Frames to Support Program Comprehension Activities* [86] at the 2021 *International Con-
ference on Program Comprehension (ICPC). This work was done in collaboration with Arthur Marques, who is the first author of the paper. The work was split evenly between the two of us.

Finally, the work presented in Chapter 7 includes parts of two publications. The first is a paper titled *The structure of software design discussions* [151] published at the 2018 International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE): this work was done in collaboration with Calahan Janik-Jones and Michalis Famelis. The second one is a paper titled *What design topics do developers discuss?* [152], published at the 2018 International Conference on Program Comprehension (ICPC): this work was done in collaboration with Calahan Janik-Jones, Michalis Famelis and Xin Xia.
# Table of Contents

Abstract ......................................................... iii  
Lay Summary .................................................. v  
Preface ......................................................... vi  
Table of Contents ........................................... viii  
List of Tables ................................................ xii  
List of Figures ............................................... xiv  
Acknowledgments ............................................. xvi  

1 Introduction ................................................ 1  
  1.1 Recovering Design Information ......................... 2  
  1.2 What is Software Design ................................. 3  
  1.3 Locating Design Information from Developer Discussions 4  
  1.4 Using Recovered Design Information .................... 7  
  1.5 Identifying Useful Design Points ....................... 9  
  1.6 Thesis Overview ........................................ 9  
  1.7 Thesis Contributions .................................. 10  

2 Related Work ................................................ 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Refining the Location of Latent Design</td>
<td>103</td>
</tr>
<tr>
<td>7.2 Extracting Design from Design Points</td>
<td>104</td>
</tr>
<tr>
<td>7.2.1 Topic Identification</td>
<td>104</td>
</tr>
<tr>
<td>7.2.2 Discussion Structure</td>
<td>107</td>
</tr>
<tr>
<td>7.2.3 Frame Semantics</td>
<td>114</td>
</tr>
<tr>
<td>7.3 Design Points in Other Discussion Forms</td>
<td>114</td>
</tr>
<tr>
<td>7.4 Natural Language Processing</td>
<td>116</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>117</td>
</tr>
<tr>
<td>Bibliography</td>
<td>120</td>
</tr>
<tr>
<td>A Pull Requests Annotation Codebook</td>
<td>138</td>
</tr>
<tr>
<td>B Questionnaire on the Usefulness of Design Points</td>
<td>143</td>
</tr>
</tbody>
</table>
List of Tables

Table 3.1 Information about interview study participants .............................................. 23
Table 4.1 Statistics for the annotated pull requests ......................................................... 39
Table 4.2 Labels applied to paragraphs from pull requests ............................................. 42
Table 4.3 Annotation distribution .................................................................................... 43
Table 4.4 Features used to identify design points in a paragraph ..................................... 45
Table 4.5 Rationale behind each feature used to identify design points in a paragraph ...... 46
Table 4.6 AUC score for the proposed classifiers ............................................................... 51
Table 4.7 Results of the Scott-Knott ESD test on the features used .................................. 53
Table 4.8 Results of the comparison between the classifier and the gold standard .............. 57
Table 5.1 Identifiers for each participant in both groups. .................................................. 68
Table 5.2 Solutions chosen by participants based on group and task ............................... 68
Table 5.3 Rationale labelling results .................................................................................. 70
Table 5.4 Categorization of the responses on the understanding of design points .......... 71
Table 5.5 Categorization of the responses on the usefulness of design points ................. 73
Table 5.6 Representation of the participants sorting of design points ............................. 74
Table 6.1 Datasets for annotation and correctness studies .............................................. 82
Table 6.2 Summary of the annotation of frames ............................................................... 85
Table 6.3 Excerpt of results from annotation procedures ................................................. 86
Table 6.4 Results of correctness study ............................................................................. 91
Table 6.5 Results of robustness study .............................................................................. 92
Table 6.6  Datasets for the robustness study .........................  94
Table 6.7  Useful and not useful design points groups .................  98
Table 6.8  SEFrame frames found in design points ..................... 100
Table 7.1  Results of the coding process ............................... 106
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Depiction of the development process</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Example of a pull request on Github</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Example of a paragraph containing design information</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Comment from a pull request discussion</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Sample questionnaire followed by the interviewer</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Phases that are considered part of a standard development cycle</td>
<td>26</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Example of a paragraph containing a design point</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Example of a paragraph without a design point</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Labelled snippet of a discussion from a pull request</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Diagram showing the classifier training process</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Example of a paragraph containing a design point</td>
<td>57</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Questionnaire task</td>
<td>65</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Followup question on the questionnaire task</td>
<td>66</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Explanation of design points given to the participants</td>
<td>67</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Example of design point shown to Group II</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Example of design point shown to Group I</td>
<td>72</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Example of frames and frame elements</td>
<td>79</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Distribution of frames over sampled sentences</td>
<td>83</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Example of a sentence and the associate frame</td>
<td>90</td>
</tr>
</tbody>
</table>
Figure 6.4  Validation frames for the robustness study .......................... 95
Figure 6.5  Causation frame identified in a design point .......................... 99
Figure 6.6  Awareness frame identified in a design point .......................... 99
Figure 7.1  A design point identified in the Node.js project ..................... 103
Figure 7.2  Discussion snippet from Rails pull request #11407 ................... 110
Figure 7.3  Discussion snippet from Rails pull request #14519 ................... 111
Acknowledgments

I would like to acknowledge that UBC’s Point Grey Campus is located on the traditional, ancestral, and unceded territory of the Musqueam people.

I want to thank the members of my supervisory committee for the help they have provided me over the years. Special thanks must go to my supervisor, who has been an extremely supportive and understanding mentor from the start. Gail always made sure to support me and not only my work, even when I was troubled by external problems. Eight years in grad school is a very long time, but I could not have asked for someone better to guide me through it.

Similarly, I want to also thank all the members of the Software Practices Lab at UBC for being wonderful colleagues. In particular, I would like to thank Arthur and James for their support and encouragement in finishing my degree.

I need to particularly thank my parents and sister for their undying support of me. It is not easy to live so far from your family, even more when a pandemic prevents you from seeing each other for years, but I could always take comfort in knowing that they believed in me.

I want to thank all my friends, who now are spread all over the world. Many people entered my life throughout the years I have spent in Vancouver, and their support has also been fundamental in my success. There are too many of you to list here, but in particular I want to acknowledge Anna, Anna Maria, Hayley, Neil, Puneet, and Yasha for the positive impact they have had on me, and for all the help they have provided me over the years.
Finally, last but not least, I would like to thank Fel, Nom, and Zoey, who I have met online and who have been there to support me while I wrote this dissertation. Even though we have not had the chance to meet in person, they were always there to offer encouragement.
Chapter 1

Introduction

The design of a software system provides guidance on how a system should be built and why it is built as it is. As software evolves, developers continuously make design choices, which are reflected in updates to project artifacts. As the updates are made, design information becomes embedded in artifacts, such as source code, and discussions. Over time, design information is thus present across multiple artifacts as Figure 1.1 depicts.

The distribution of continually changing design in and among artifacts makes it difficult for developers to maintain knowledge of design information. The involvement in a project of multiple teams, each composed of multiple developers, further causes system design knowledge to become fragmented, leading to a situation in which developers do not know who holds the knowledge about the design of a particular part of the system [23]. Despite many notations and approaches for expressing design, design is often not represented explicitly over time [120], resulting in what has been referred to as evaporation of a project’s design [118]. Even when explicit design documents are produced by the developers, these documents are often not kept current with the evolving system [96] and information loss still happens in the form of design erosion [54].

There can be severe repercussions when needed design information is not available to developers due to fragmentation, evaporation or erosion. Lack of design awareness as
changes are made can result in further degradation of design and an aging of the software through a process described as *ignorant surgery* [104]. Software aging can cause repercussions such as loss of performance or an increase in defects for the system. In addition, when design information is not explicit or not current, newcomers to a project find it difficult to contribute to the system [131, 134]. In open source projects, delays in making contributions and failed contributions can slow growth and impact the health of the project [31].

### 1.1 Recovering Design Information

To address the loss of design information about a project, many researchers have considered ways to recover design information from project artifacts. A popular type of artifact from which to recover design is the source code. Given that the source code for the project represents the system, design extracted from this source is by definition relevant and up-to-date. The reverse engineering [28] approaches that have been developed center around inferring structural (e.g., [99]) or behavioural (e.g., [29]) designs from code and execution software artifacts. While these approaches have proven to be useful to recover information
about how a system works, they often fall short of explaining why certain design choices were made.

More recently, researchers have considered how design information is embedded in developer discussions recorded in various project artifacts, such as issue reports, pull requests and mailing lists [24, 147]. Concerns related to design are often raised during pull request discussions [147], and are in fact crucial factors for deciding whether to accept pull requests [52]. This suggests that these discussions include information that goes beyond how a system works. This thesis builds on written recorded discussions between developers as a new source for embedded and latent information about the design of a system.

1.2 What is Software Design

A pre-condition to recovering design information from developer discussions is to understand what developers consider as design information. One might expect given the ubiquitous references to software design in software engineering teaching and research that there is a common and well-established understanding about what software design is and is not. Unfortunately, there are no agreed upon definitions of software design. Perhaps, even though definitions are elusive in the research literature, there might be a common understanding about software design across practitioners. However, no literature currently captures such a common understanding.

To provide a basis for improving the recovery of design from developer discussions, this thesis reports on an exploratory interview study to determine if software engineering researchers and practitioners hold a common view of what software design is [150]. The study involved interviews of 16 participants from both academia and industry. The interviews were transcribed and coded to search for the presence of common topics that multiple interviewees discussed. Five major recurring themes were identified from the analysis, which describe design as:

1. belonging to multiple phases,
2. being about making decisions,
3. being dependent on context, 
4. involving communication, and 
5. benefiting from experience.

Two of these themes discuss when design occurs, namely during multiple phases (#1) and when communication occurs (#4). Two of the themes describe aspects about design, namely being affected by the context of development (#3) and the experience of those involved (#5). The last theme tells us about how to potentially recognize design, namely when decisions are made (#2). These results motivate the focus of this dissertation on recovering design information and decisions from the regular communication that happens between developers.

1.3 Locating Design Information from Developer Discussions

Developer discussions that could be a source for design information occur in multiple forms in software projects, including in issue reports, pull requests and mailing lists. In earlier work, Brunet et al. showed how nearly 25% of issue reports contain design information [24]. A closely related concept to issues are pull requests. For example, on the GitHub platform, issues and pull requests are closely tied, with the main difference being that pull requests have one or more Git commits associated to them. Moreover, Tsay et al. have reported that pull request discussions often include explanations and defenses of design decisions by developers [147].

Figure 1.2 shows an example of a pull request discussion on Github. A pull request discussion is composed of an initial message, that usually introduces the code change proposed and provides some motivation, and a sequence of messages posted by the same or other contributors. These discussions can contain a large number of messages: for example, the pull request displayed in Figure 1.2 contained 53 comments. Some of these messages include information about choices made about how the code is or should be written. To

[1]https://github.com/
Figure 1.2: Example of a pull request on Github from the Node.js project
This figure presents an example of a paragraph containing design information, from pull request #4077 of the Node.js project. In this message, the author discusses the removal of the `existsSync()` function and how throwing an error would be a breaking change for the existing API. These choices represent decision points about design. We refer to these choice points as *design points*: a piece of a discussion relating to a decision about a software system’s design that a software development team needs to make.

To determine the extent of design points in software projects, my collaborators and I annotated 10,790 paragraphs from the Node.js[^2], Ruby on Rails[^3], and Rust[^4] projects. This annotation activity resulted in the identification of 2,378 paragraphs as design points with a Fleiss’s Kappa Score of 0.52, commonly understood as moderate agreement [43, 69]. Our annotation revealing that 22% of the paragraphs contained design information echoes previous results from Brunet et al. [24], who showed that nearly 25% of issue reports contain design information.

We then used this dataset to develop a classifier that is able to automatically localize design points in pull requests discussions across different systems. This classifier is based on 19 features divided in 4 categories:

- **Process features** based on the author of the paragraph (e.g. whether the author is a core project member);

[^2]: https://github.com/nodejs/node
[^3]: https://github.com/rails/rails
[^4]: https://github.com/rust-lang/rust
• **Position features** based on the location of the paragraph within the discussion (e.g. the position of the comment in the discussion);

• **Text features** based on textual characteristics of the paragraph (e.g. the number of words in the paragraph);

• and **Content features** based on the vocabulary of the paragraph (scores calculated based on the textual content of the paragraph).

The classifier was evaluated in two different ways: 1) through an analytical cross-project evaluation, and 2) through a human evaluation. For the analytic evaluation, the classifier was evaluated on each of the three projects by training it on the other two. For the human evaluation, five computer science students annotated the presence of design points in five new projects that the classifier had not been trained on: *Bootstrap*[^5], *React*[^6], *FreeCodeCamp*[^7], *TensorFlow*[^8], and *Electron*[^9]. The resulting annotation created a gold standard against which the classifier trained on the 3 original projects was tested.

The two evaluations respectively reported an AUC [40] of 0.87 for the analytic (cross-project) evaluation and 0.79 for the human (gold standard) evaluation. These results show that the classifier is able to reliably localize the presence of design information within pull request discussions.

This classifier is the first to automatically identify design information embedded in developer discussions.

### 1.4 Using Recovered Design Information

Locating design points is useful only if it can be made accessible in ways that can help software developers perform their normal tasks. Let us consider the example shown in Figure 1.4. In this recent pull request, the author is saying that having the two functions

[^5]: https://github.com/twbs/bootstrap
[^6]: https://github.com/facebook/react
[^7]: https://github.com/freeCodeCamp/freeCodeCamp
[^8]: https://github.com/tensorflow/tensorflow
[^9]: https://github.com/electron/electron
fs.exists() and fs.existsSync() throw an error would be an acceptable option. At the same time, the design point in Figure 1.3 that belongs to a different pull request, disagrees with this notion and argues that throwing an error would be a breaking change. Imagine if developers had access to a tool that would notify them automatically of existing design points relevant to the task they are working on. In this example, the author would be notified of the existing design point, and would be able to take this additional information into account.

To investigate whether design points can be useful in this context, we ran a user study in which newcomers to a project were asked to pick an appropriate solution for a given development task from a set of possibilities. Each participant was presented with two tasks: in the first task, participants did not have access to design points; in the second task, participants were given access to a small set of potentially relevant design points.

The goals of this study were to investigate if newcomers to a project can understand the design points provided and choose those design points most suitable for the task. The study involved 15 participants. The results of this study indicate that newcomers to a project can interpret the information presented in design points, and this information can help them consider different design alternatives.
1.5 Identifying Useful Design Points

To further improve the quality of design points that are shown to developers, it would be beneficial if we could also differentiate which design points are likely to be more useful among those identified to be relevant to the task at hand. To differentiate useful design points, we investigated using frame semantics, a linguistic approach that aims to represent the intended meaning of the words in a sentence.

As the application of frame semantics to software engineering text has been limited, we began by investigating the applicability of semantic frame to software engineering text. We developed SEFrame, a tool that tailors semantic frames to software engineer text, and ran two evaluations to assess its correctness and robustness.

Finally, we explored the application of SEFrame to a subset of the design points used in the previous study (Section 1.4) that had been identified by participants as being the most useful.

1.6 Thesis Overview

This thesis explores whether:

*Design information about a software system embedded in written discussions between developers can be identified automatically and made available to developers in a form that allows them to make better design decisions as the system evolves.*

To demonstrate the usefulness, plausibility and novelty of extracting design information from developer discussions, Chapter 2 presents an overview of the existing literature on design recovery, the importance of design in software development, and the presence of design in developers communication.

Having identified that software design has many different meanings, Chapter 3 reports on an exploratory interview study conducted to understand what researchers and practitioners think design represents. This work was published in a conference paper and presented at
the IEEE International Conference on Software Analysis, Evolution and Reengineering in 2022[150].

In Chapter 4, the novel concept of a design point is introduced to represent design information in written discussions and describe the creation of a large annotated dataset of paragraphs from real pull requests. A novel application of a supervised classifier is also introduced, that is able to locate design information automatically in written discussions. This work was published in the IEEE Transactions of Software Engineering [153] and was presented at the IEEE/ACM International Conference on Software Engineering in 2021 as a journal-first presentation.

Design points can only become useful to a developer if a developer is able to apply the design information in a design point in the context of a current task. In Chapter 5, we report on the results of a user study we ran to investigate if developers could understand and make use of design points. We detail the design of the study and discuss the results obtained.

In Chapter 6, we discuss the applicability of frame semantics to software engineering text, and explore their use to identify useful design points. Part of this work was published in a conference paper and presented at the IEEE/ACM International Conference on Program Comprehension in 2021 [86].

The thesis concludes with a discussion in Chapter 7 regarding our findings and potential avenues for future work. Chapter 8 summarizes the dissertation and presents our conclusions.

1.7 Thesis Contributions

This thesis makes the following novel contributions:

1. It reports on five themes about software design that arose from an exploratory interview study with 16 participants from academia and industry. These themes can help situate research on software design.
2. It introduces the concept of a design point, to indicate a paragraph in a written developer discussion that contains information regarding a design decision made by developers.

3. It introduces and reports on a classifier that is able to locate automatically the presence of design information in pull request discussions.

4. It provides evidence of the understandability and usefulness of design points based on results of an user study with 15 developers.

5. It introduces SEFrame, a tool that tailors frame semantics to software engineering text.

In addition, the thesis has produced 3 datasets that may be useful to other researchers:

1. The transcripts of 16 interviews involving both researchers and practitioners, discussing what the term software design represents for them.

2. An annotated dataset consisting of 10,790 paragraphs from 34 pull requests from 3 GitHub Projects. The dataset is annotated for the presence of design information in the paragraph, out of which 2,378 (22%) were identified to contain design information.

3. The responses of 15 developers to a questionnaire about a hypothetical tool to recommend design points, including such information as the rationale for their choice of an ideal solution and information about whether and why design points were useful.

All the datasets are available at https://github.com/vivianig/dissertation-datasets.
Chapter 2

Related Work

Design is a concept that is integrally linked with software engineering. The conversations captured from the 1968 Nato Software Engineering Conference, which is seen as the birthplace of the field of software engineering, constantly and consistently refer to design as part of software engineering [100]. Despite this long history of design as part of software engineering, views on what software design is in the literature are quite divergent. For instance, design can be seen as the specifications [113] to which a development team must adhere, the process [44] in which the team takes part, as compliance to international† standards [1], as the choice of specific algorithms [49], or as a means to provide a shared vocabulary [45].

The breadth of views on software design have resulted in a breadth of literature on software design. In this dissertation, we build on existing work about how software developers view design. Thus, we begin a focused review of the existing literature on software design with a synopsis of how design is viewed based on empirical studies of software design in practice (Section 2.1). Recognizing that despite the multitude of ways to capture design as it occurs—such as through UML diagrams [19, 126] or architectural description languages (e.g., [47, 143])—design is often not captured, we review the kinds of design recovery techniques that have been created (Section 2.2). We describe that the bulk of the work in existing design recovery focuses on recovery from technical artifacts, such as source
code and execution traces. While these techniques can help discover how a system is constructed, they do not answer the question of why the system was constructed as it is. Why a system is designed in a specific way can be described as the rationale behind the design (Section 2.3). In this dissertation, we add to the techniques for determining why software design is as it is based on social artifacts, such as developer discussions. We review existing literature about design and developer discussions (Section 2.4) to provide background for the novel technique we present.

2.1 Studies of Software Design

Early studies of software design in action extracted insights from experience with system development. These insights were then framed into repeatable principles and processes with their efficacy and usefulness conveyed via case studies. Many of Parnas’ early papers took this form, such as the paper describing information hiding [103]. His later extension of this work demonstrated the ideas in the context of an actual flight system [106].

As software engineering began to mature as a field, researchers moved into empirical study of the processes associated with software design. For example, Curtis et al. interviewed individuals from 17 large projects primarily characterizing the processes used in designing these systems [32]. Drawing from many studies of software designers at work, Petre and Van Der Hoek have captured and described 66 ways that expert software developers design, such as ’Experts design throughout the creation of software’ [108]. Zannier et al. also use interviews to hone in on how developers make design decisions, finding that the more structured the design problem, the more rationale the decision making process, whereas when the problem was unstructured, the decision making became naturalistic [162]. The study we report on in Chapter 3 shares similarities with studies of the software design process. The study we conducted differs in exploring how developers think about design today. In comparison, Petre and Van Der Hoek focus on describing what a sub-class of developers do, namely those who are considered expert designers. Similar to Zannier et al., the study we conduct finds design decisions as a critical aspect of design; our study does not delve in-depth about decisions, rather we focus on using decisions as a means of identifying when design occurs.
Software engineering researchers have also studied design as an activity. For example, LaToza et al. study development teams at Microsoft, reporting on the activities they undertake, including designing [70]. At times, the study of design as an activity has focused on specific interaction points between developers rather than characterizing the activity overall. For example, Cherubini et al. discuss when and why developers use a whiteboard, finding that one of three reasons developers use visualizations is to design [27]. In Chapter 4, we consider that the activity of design is often externalized as discussion of decision in developer discussions. We describe how our pursuit of design as recorded decisions differs from other methods of design recovery that focus on extracting how a system works from technical artifacts. In contrast, we focus on identifying information about why a system was designed as it was. In pursuing information about why, or in other words rationale, we are attacking a problem identified by LaToza, after undertaking surveys and interviews at Microsoft, as one of the most serious problems developers face [70].

2.2 Design Recovery

To combat design evaporation [118] and design erosion [54], researchers have investigated many approaches to recover lost design information from software systems. Initial attempts to recover design information were focused on extracting design from source code, and were defined as reverse engineering. Reverse engineering has origins in hardware and was described by Rekoff as “the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system.” [116]. Some years later, Chikofsky et al. argued that it is possible to apply the same approaches to a software system to “gain a sufficient design-level understanding to aid maintenance, strengthen enhancement, or support replacement.” [28].

Most design recovery techniques focus on inferring either the system’s structure or its behaviour. Approaches that focus on the system’s structure attempt to recover information regarding what components are in the system and how are they connected to each other. Some approaches that have been developed are bottom-up. For example, the Rigi editor analyzes structural models extracted from the source (e.g., call graphs) and applies automatic clustering approaches to help identify software components and their connec-
tions [99]. Other approaches are top-down. For example, the software reflexion approach has developers specify a high-level component-like model and map it to a low-level model extracted from the system, allowing the developer to explore differences between their view and the reality of the system [95].

Approaches that focus on recovering descriptions of a system’s behaviour vary in their targets. Some approaches to recover behavioural descriptions focus on producing descriptions for other tools. For example, Bandera [29] generates finite-state models of Java code, which can be helpful to ease the verification of the system’s properties. Other approaches aim to support a developer’s understanding of a system. For example, Briand et al. [22] and others have explored the creation of UML sequence diagrams from execution traces of a system.

Techniques for inferring structural and behaviour design focus on source code and execution logs, or traces, as input. These techniques focus on transforming the static and execution information into descriptions of how the system meets its requirements. In this thesis, we are focused on why the system has been structured or behaves as it does. For example, a developer may choose between two alternative designs because of a need to focus on a non-functional requirement, such as performance. In this case, performance would be a rationale for why a certain design was pursued.

One question that arises is where information about why a system is as it is might found. Biggerstaff’s work provides some clues. He argued that design recovery differs from the simple act of reverse engineering in needing domain information and that reaching design recovery likely requires “includ[ing] more information than the analyst might find in the code alone” [17]. The ‘more information’ called for by Biggerstaff might be found in other artifacts produced as part of software development. Traceability research provides a basis for exploring such other information as it helps to build links between different types of artifacts, such as between code and documentation [9], and between code and commits/issues [128]. Alternatively, one might monitor the development process itself to acquire and record rationale information as it occurs. For example, Myers et al. monitored designers working using computer-aided design software to acquire rationale information [98].
2.3 Design Rationale

The need for developers to understand why software has been designed in a certain way—design rationale information—has been long recognized [93]. Researchers have thus considered ways to proactively and retroactively determine the design rationale for parts of a system.

The origins of research about design rationale can be found in the work of Rittel, who created the Issue-Based Information System (IBIS) [68], as a way to proactively record design rationale. Unfortunately, the proactive gathering of design rationale information was not found to be efficient or effective. Developers and designers attempting this type of approach faced multiple barriers and received no immediate payoff [60]. To ease the burden of describing rationale, McCal et al. proposed an approach to automate the process using an argumentative grammar [88]. This approach proved more successful, but its evaluation was limited to three transcripts and no further investigation was made to understand at what scale it could work.

An alternative approach to proactively asking developers to express rationale is to retroactively look for design rationale information where it is naturally occurring. As described earlier in the thesis, a naturally occurring location of design rationale information is in socially-oriented artifacts captured as part of a system development. This thesis investigates a retroactive approach to determining design rationale by focusing on socially-oriented artifacts in the form of recorded discussions between developers.

2.4 Design in Developer Discussions

In recent years, software engineering researchers have begun to pay more attention to the social context in which software is developed. These contexts can provide additional information that would not be otherwise accessible, as “software developers reveal their thought processes most naturally when communicating with other software developers” [127].

Some discussions between developers are naturally captured, such as in mailing lists,
issues and pull requests. Barcellini et al. explored how design appears in the python-dev mailing list [15] by manually categorizing a conversation of 3800 lines of text. Brunet et al. show discussions on issues can be bountiful source of design information [24]: they manually coded 1,000 discussions and trained a classifier to identify which discussions on issues include design, finding that roughly 25% of the discussions were about design. Zanaty et al. ran a similar study by training a classifier on a sample from 2,817 manually classified review comments, and found that 73% of design-related comments provide constructive suggestions [160]. Work by Tsay et al. shows that pull requests discussions also include design [147]; moreover, these discussions are often a crucial factor in deciding whether to accept pull requests [52].

The availability of information about social contexts through recorded developer discussions has facilitated the investigation of design rationale recovery, a goal that would have been unlikely achievable from only technical artifacts [121]. Rogers et al. began by investigating the use of text mining based on linguistic features to identify rationale in existing documents [123]. Later on, they further developed their approach and applied it to a corpus of bug reports, achieving results comparable to human annotators [124]. Liang et al. proposes a graph-based approach to language patterns to identify rationale in design documents based on a three-layer model connecting issues, design solutions, and artifacts [77]. Bhat et al. developed a machine learning approach to extract implicit design decisions from JIRA issues, a well known issue tracker [16]. More recently, Lester et al. investigated more complex approaches using evolutionary algorithms to identify rationale in bug reports and transcripts of design discussions [72]. These approaches tend to focus on locating design rationale either at the document or at the sentence level. The work presented in this thesis differs by targeting paragraphs as they are more likely to contain a full thought than sentences, while remaining at significantly smaller scale than an entire document. In addition we focus exclusively on pull request discussions instead of issues or bug reports.

Of course, records of developer discussions are not limited to mailing lists, issues and pull requests. Developers are also extensive users of instant messaging (IM) applications and social media [58]. IM is a phenomenon that has recently resurged, both within com-
panies and open source communities, slowly taking over the role traditionally held by emails [39, 63, 136]. Researchers are beginning to look to these artifacts as targets to identify design information: for example, Alkadhi et al. [7] investigate the use of machine learning approaches to reliably detect the rationale behind design decisions in IRC channels; Lima et al. employed data-mining techniques to identify lost knowledge within instant messages [78]. For this thesis we opted to exclude these form of discussions from the scope of the investigations, as IM discussions lack the structure that is present in pull requests discussions and would introduce additional challenges. Nonetheless, we don’t exclude the possibility of extending this work in this direction, and we discuss the topic more in depth in Chapter 7.

Developers also participate in discussions that may not be recorded, such as in-person discussions or whiteboard sessions. When researchers study this context, they are studying the development of design in its “natural setting”, often aiming to describe the activities involved in the process of design [107]. Fewer studies of this nature. One notable example is Rodeghero et al. who analyzed recordings of discussions between developers and their clients to automatically generate requirements in the form of user stories [122]. This work shows that it is possible to investigate oral discussions, even though it poses significant challenges. For example, it is usually not possible to retroactively obtain discussions as they are often not recorded, therefore needing to run long field studies to record new discussions. While this kind of discussion is not covered by this thesis, our work does not preclude the possibility to expand in that direction.

### 2.5 Summary

Design has permeated software engineering since its inception, and it is necessary to understand its origin and evolution before attempting to capture it. We began this chapter by covering how researchers have investigated and identified design, both as a process and as an activity. Section 2.1 describes how design has been traditionally studied in the context of understanding more how a software system works rather than why it was design to work like that, upon which we find our motivation to focus on the study of the latter.
Having covered how design has been identified by researchers, the next step moved the focus from design itself to the task of maintaining it and recovering it. Design recovery has been proven to be necessary because of the natural erosion of the knowledge over time, but also because of a lack of consistency among developers in writing down this kind of information at the moment of its inception. Section 2.2 provides a broad overview of the many approaches that have been developed over the years, from the initial applications of reverse engineering approaches from hardware to software, to the more recent attempts to localize not only the design, but its rationale too.

The recovery of rationale information has particularly benefited from the expansion of design research to include the social contexts of software development. In Section 2.4 we described how researchers have been leveraging this new source of rich design information. In this section we position our work in the composition of the state of the art of design recovery from developer’s discussions.

In the next chapter we take a step back toward Section 2.1 and discuss our investigation of how developers and researchers think about design.
Chapter 3

What Really is Software Design?

As described in Chapter [1], although there is broad agreement that the development of software includes design, there is less consensus about what software design is. One might think that a definition for design would be found in descriptions of the widely-known Unified Modeling Language (UML) [19, 126], which enables the recording and visualization of the design of a system. Although the terms design and to design appear frequently in the texts about UML, no definition is provided for the terms. Similarly, the well-known design patterns book [45] also does not define the term design.

Perhaps the lack of an accepted definition of software design is not a problem. After all, research occurs about design (e.g., [141]) and software systems are developed successfully each day. However, without accepted uses of the term, misunderstandings can occur in a software development that reduces the productivity of the developers, such as asking for the design of a part of a system from a team, being surprised at the response and having to schedule meetings to reach understanding. Without accepted uses of the term, what is taught as design in educational settings can differ from what design is in practice. And without accepted uses of the term, it can be challenging for researchers to locate and build on the results of others. For this thesis, a lack of a common definition also makes it challenging to determine what design information can or should be sought from developer discussions.
It is possible that even though definitions are elusive in the literature, there is a common understanding about software design across practitioners, across software engineering researchers and across both groups that could be used to inform the research in this thesis and beyond. To determine whether this common understanding exists, we conducted an interview study to investigate how various developers and researchers view design; this study method allows the exploration of perspectives of participants without presuming particular views.

The chapter begins with a description of the approach to the interviews and to the analysis of the data collected (Section 3.1). Next, themes resulting from the analysis of the data are presented and discussed (Section 3.2 and Section 3.3). The chapter concludes with a description of threats to validity of the study results (Section 3.4) before a summary of the results is presented (Section 3.5).

3.1 Methodology

To gain insight into how software design is viewed today, we interviewed individuals from both academia and from industry. Academics were included in the participant group as these individuals teach software design and conduct research to further software design methods and processes. Industrial participants were included as these individuals create, evolve and interact with software design as part of their work. This section describes the participant recruitment approach, the interview format and how the collected data was analyzed.

3.1.1 Recruitment

The recruitment of academic participants focused on individuals who have published on software design. We queried The IEEE and ACM databases for every paper published between January 1st, 2016 and December 31st, 2020 (5 years) where the terms “software design” or “software architecture” appeared in the title or in the name of the proceeding. The term 'software architecture’ was included as architecture is often considered a form of design (e.g., [2]) and there are specific conference series on software architecture (e.g.,
IEEE Conference on Software Architecture-ICSA\textsuperscript{1}. After removing duplicates, this query resulted in 437 papers, from which we selected every author whose name appeared two or more times, for a total of 194 authors. We attempted to recover an email contact from each author from their website. Eventually, we sent 193 recruitment emails, one for each retrieved email contact. In thirteen cases, the email bounced back due to an invalid address.

The recruitment of industrial participants followed a similar approach. Individuals were contacted who have been authors in the Software Engineering in Practice (SEIP) track of the International Conference of Software Engineering (ICSE)\textsuperscript{2}, as these individuals have demonstrated specific knowledge or interest in software design. We crawled the proceeding of the SEIP track or the years 2016-2020 (five years) and selected authors not affiliated with an academic institution. This resulted in 146 practitioners, whom we sent the same recruiting email sent to academics. In 28 cases, the email bounced back due to an invalid address.

### 3.1.2 Participants

Out of the 298 emails successfully sent, we received 21 responses, roughly 7%. Out of these 21 emails, 16 responses were for individuals associated with academia and 5 from individuals associated with industry. Not all of these responses resulted in a participant for the study, as some respondents stopped replying or were unable to arrange a convenient time for the interview.

These recruitment efforts resulted in 11 participants: 9 academics and 2 practitioners. To expand the list, two approaches were used. First, we reached out to known contacts in industry, who provided us with recommendations of other potential participants who were then contacted. This resulted in two additional practitioners. Second, academics who were recommended by academic participants were contacted, resulting in 3 additional academic participants. For these three additional academic participants, we first verified whether they would have appeared in our queries if we had extended them to a larger window of time.

\textsuperscript{1}https://icsa-conferences.org
\textsuperscript{2}http://www.icse-conferences.org/
Table 3.1: Information about interview study participants

<table>
<thead>
<tr>
<th>Gender</th>
<th>16 Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 Men</td>
<td>5 Women</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>12 Academics</th>
<th>4 Practitioners</th>
</tr>
</thead>
<tbody>
<tr>
<td>Professors</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Students</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Researchers</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Developers</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Continent</th>
<th>2 Asia</th>
<th>4 Europe</th>
<th>7 North America</th>
<th>1 Oceania</th>
<th>2 South America</th>
</tr>
</thead>
</table>

The interview study thus comprised interviews of 16 participants: 12 academics and 4 practitioners. As depicted in table 3.1, the participants ranged from graduate students studying design to senior researchers and from younger practitioners to those with many years of industrial experience. The participants were spread across 4 countries and 5 continents.

### 3.1.3 Interview Method

After completing a web-based consent form, a video call was scheduled with a participant. All interviews were held using the Zoom platform. The interviews followed a semi-structured approach: the interviewer introduced the topic and had a dialogue with the participant. Figure 3.1 shows the prepared list of questions used to help ensure the dialogue moved forward and considered important points. For questions that involved identifying examples of design concepts, the interviewer was careful to keep the questions open-ended (e.g., without proposing any examples), to ensure the interviewee’s own thoughts were captured.

Each interview lasted between 20 minutes and an hour. All interviews were recorded and then manually transcribed, removing any reference that could trace back to the participant. Transcripts can be found at [https://github.com/vivianig/dissertation-datasets](https://github.com/vivianig/dissertation-datasets)
3.1.4 Coding

To better analyze the interview transcripts, we split the transcripts into smaller units. Initially, we considered using each participant response to a question or a statement of the interviewer as a single block of text. In multiple cases, the responses given by participants were long, and touched multiple points. To not lose any information, we opted to further divide these blocks in smaller ones, each one containing one or more sentences. These blocks were formed by grouping sentences in a way that each block could be summarized to a single point or statement. Following this step, the 16 transcripts resulted in 351 blocks. To analyze this data, we made use of the Trello website\(^4\). We organized the data with a list representing each participant, and a card representing each block. For the remainder of this section, the blocks are referred to as cards.

Not all of the cards contain comments about design, as the participants may have started talking about other topics before the interviewer re-directed the discussion back to design. For this reason, we executed another pass over the cards, eliminating any card that did not discuss design. This step resulted in a set of 234 cards.

\(^4\)https://trello.com/
The study authors then proceeded with an open coding approach[50] with the goal of identifying trends in the participant responses about software design. This approach has been used in other similar work[62].

To improve consistency between the two coders (the study authors), the 16 participants were divided in three groups: G_1 had 5 participants, G_2 had 5 participants and G_3 had 6 participants. Each one of us first independently coded G_1 and then the codes from each were compared, discussed and consolidated. We then independently coded G_2 before comparing, discussing and consolidating, and so on. During the process, we focused on the ideas behind the categories emerging from the cards, rather than focusing on the cards being assigned to the same themes. This step resulted in 24 categories. To further condense the results, we did a second round of open coding, this time categorizing the 24 themes into meta-categories, which we refer to as themes for clarity. This final step resulted in 5 themes.

3.2 Themes

Five themes emerged from the data analysis: 1) design is practiced by all developers and everywhere, 2) design involves decision making, 3) design depends on context and software evolution, 4) design involves communicating information to other developers, and 5) developers benefit significantly from design experience. The themes are presented in this order which represents the most (to least) in terms of number of participants making comments related to the theme. Quotes from the participants are used to help ground the themes: participant identifiers starting with ”A” represent academic participants whereas participant identifiers starting with ’I’ represent industrial participants.

3.2.1 Multiple Developers and Levels

Many individuals first encounter the concept of software design during their education. In many cases, this education will be based on a software engineering textbook (e.g., [132]). Many of these textbooks represent design as a phase in a software project’s development cycle. Figure 3.2 provides an abstract representation of the kinds of pipelines that appear
Figure 3.2: Phases that are considered part of a standard development cycle in textbooks.

Eleven of the participants appealed to the textbook notion of design as a phase, not to re-enforce it, but instead to argue it. They described that design is not a phase itself, but rather happens throughout each phase of a project’s development.

“[Although] lots of people [...] talk about design as a phase in software development, we [...] argue that design happens in other phases, it happens in maintenance, it happens through the development of product lines, through debugging, through all sort of places, because at any of those point there is a need to step back and reassess.” - A8

Unsurprisingly, if design occurs throughout various phases of development, the focus and the level of detail considered in the design activity varies.

“There are high-level design and low-level design kind of things.[...] High level we have things like use cases, architecture [...] Then on the low level, if the component has some backend database you come with some schema diagram, if you want to use a new class or a new method, then how do you do it [...]” - I4

Participants also expressed that design is the product of multiple people and not of a specific member taking the role of designer.

“You have [a] project leader, you have software architects, interaction designers... I think they are just terms used that are fairly common in industry, and they all do design-kinda work, so they’re all designers in my mind.” - I1

Although the majority of participants agreed with design occurring throughout development, there were dissenting views by participants who saw design as happening only during specific phases of development. Not all of these views align with the textbook notion
of software design.

“Everything that’s after you understand the requirements, until you start typing the code, should be in the design” - A7

“Writing the code is where I do the design.” - A9

3.2.2 Decision Making

While the first theme informs that software design occurs frequently during software development, it does not provide many clues about how to recognize what software design is.

Outside of software engineering, the word design may conjure up the work of an architect planning for the construction of a building or a structure. In a similar way, 10 participants talked of design as planning the software and constructing a blueprint of the system.

“To design at the end is a plan, that I can choose to instantiate and create real thing from, or it can remain a plan.” - I1

“Software design is like the blueprint of the system you want to develop. If you compare it to constructing a building, the blueprint of the building I would consider that corresponding to what we call software design.” - A11

The creation of a plan is the act of making many decisions. Participants described design in terms of making decisions that will have an impact.

“Any decision you take which will impact different aspects of the software development cycle such as what libraries are you going to use, which framework/architecture, how long it will take and so on” - A3

Decision making is, by definition, tied to choosing between multiple options. Some participants were clear in stating that good practice when it comes to designing is to keep options open and understand what alternatives are available.

“There are certain things that are practices that we see, like ‘keeping options
While describing design in terms of planning the software, participants further explained their belief that the software’s design is determined upfront, effectively building the foundation of the system.

“Software architecture is about the base of the system, those things that you cannot change easily” - A2

Some participants did not agree that design was about decision making. Instead, some participants argued the opposite, namely that design emerges throughout the various phases of the development cycle.

“Design is something that emerges from the system to someone who uses the system or gets an overview of the system” - A4

### 3.2.3 Evolution and Context

The previous two themes highlight two points on which participants seemed to disagree. In the first theme, some participants did not agree that design belongs to every phase. In the second theme, some participants argue that design is determined upfront, while others describe that it emerges throughout the development process. The third theme that emerged from my interviews provides reasons behind these disagreements as 7 participants talked about context affecting design.

Three factors were frequently mentioned with the first being the scale of the project. Participants argued that the scale can significantly affect design, not only in the form of complexity, but also its importance and the way it is approached.

“I think design is a word, a terminology that is strongly associated with the scale and the scope of the software. If the software gets more complex, you have to design more things. If the software is more localized, the design will only be very details stuff” - A7

Not all software is the same, therefore we can expect the design to also differ depending
on the type of project: for example the design decisions or concerns regarding REST APIs [76] would be very different compared to the ones regarding microservices [59]

“Depends on what kind of software. [...] For example a mobile app, if it runs only on the phone, what you can design is only how you can layer your code. When the software grows bigger, there are many things that you can design. [...] For example, if your software depends on multiple devices to talk to each other, then how they communicate.” - A7

Finally, a last major context highlighted by multiple participants is the size of the company or team working on the design, but also its culture, that can affect how much importance is given to design:

“I worked at some small companies, like 10 of us, and we still didn’t run design docs, just stand up at a whiteboard. There are definitely difference based on company size, larger companies will have more formal approaches, but my guess is that small companies would have even less formal approaches to it.” - I2

3.2.4 Communicating Design Information

The fourth topic that emerged from the interviews focuses on how the design is recorded and communicated, based on the comments of 5 participants.

When it comes to record and visualize design information, UML is usually the most well known format as it provides multiple formal standards to record and visualize any kind of design information, each one specialized to represent the salient aspect of that specific type of information.

“Yeah. I think if you talk about it many people will agree that, well you tend to think about UML diagrams as design, or objected oriented principles, information hiding, generalization and stuff like that. So I think that, or I hope that most people would agree that a UML diagram would be some kind of design of a software.” - A9
Unfortunately, UML has its limitations. One of the main complaints is that it is too formal and developers prefer to quickly sketch a diagram less formally when they need one.

“So UML is the classic version of academics, is totally different than in practice. [...] the reality in practice is that you draw something that is kinda like a class diagram but then you throw a data flow on top and it doesn’t really matter as long as the person you’re talking to understood” - I2

We note that the academic participants commented more favorably about UML than the industrial participants.

Participants highlighted it can often be easier to communicate design information through verbal communication rather writing it. One of the main reasons for this is that having a discussion allows one to choose what information to communicate and in which way, depending on who the recipient is:

“The reason I said it’s a conversation is because the diagrams are not always the same right, so depending who I am talking to I might do a higher or lower level view of the architecture, I might drill into particular things or I may not even show all parts of the system, but rather focus on certain parts at the start” - I1

Participants also mentioned that, while communicating design in this way, it is often easier to use informal sketches rather than complex notations like UML. These sketches are used as a tool to help the discussion, but do not contain information that can be directly used as design documentation:

“Yes but it’s hard. I have a decent collection of pictures of whiteboards. And people take those pictures, but actually the content on these whiteboards is very sparse. So unless it servers as reminder of what we talked about, it actually doesn’t contain what we did.” - A1

As a result, this information is often not transcribed somewhere, and even when it is, it is not necessarily kept up to date because the developers involved already hold the informa-
tion in their memory:

“*When the developers meet, I’ve seen they talk about design and work visually on them, but it’s true that this information doesn’t get to the repositories and documentations.*” - A5

3.2.5 Experience

The previous theme mentioned the issue of how design information is often transmitted verbally without being recorded. This final theme, supported by 4 participants, informs about the importance of experience and experienced developers.

Experienced developers are vital for the creation of good design. Their strength is in the knowledge that they’ve accumulated over time that informs their decisions:

“*The reason an experienced engineer is more valuable is not because they have written design, but because they remember these things. A lot of the learning that you do it gets recorded in your head, so you remember that you tried that another time and you know it’s not going to work.*” - I1

This kind of knowledge can’t be acquired just by reading documentation. Instead developers learn by doing or by working with more experienced developers.

“*It’s learning by doing, learning by having made a bad design mistake once and never make that mistake again, it’s also by having someone else review your design, that’s why we do design reviews.*” - I2

Even within the context of an ability to learn design, some individuals emerge over time as better designers, indicating that individuals saw it as possible to compare designs even if they find it difficult to articulate what makes one design ‘better’ than another.

“*At the same time there are people in teams who over time emerge as better designers*” - A14
3.3 Discussion

The five themes that emerged from the analysis of interview data indicate the range of what participants think about when they reflect on the concept of 'software design'. In this section we present a series of reflections inspired by the themes and the ways they conflict, and offer some observations that can help guide design research forward.

**Design is everywhere and design is nowhere.** As described in theme one, many of the participants commented that design is prevalent throughout many software development activities. In this way, software design is *everywhere* in a software development. On the other hand, participants also noted the widespread use of informal formats and/or channels to communicating design information as described in them four. In this way, software design is *nowhere* because it is often not possible to find recorded design information that can easily be located and understood after the moment the design is discussed.

This dichotomy helps explain why it is so difficult to get a handle on what design is and provides evidence for Petre et al.’s descriptions of software design in terms of multiple dialogues [109]. It also highlights the need for more research considering when and how design occurs during development. Additional ethnographic approaches, such as those taken by Cherubini et al. [27], may be one approach to gaining additional insights into design in practice. Alternatively, empirical approaches might target helping developers identify and record decisions as indicated by theme two as a means to capture design. These decision points could then be revisited through interviews of developers to better understand and characterize design. This complementary approach might provide new insights into design.

**Design is disseminated rather than taught.** According to theme five, junior developers learn how to create good design by practicing and receiving help from their senior colleagues. This notion hints at junior developers joining teams without design experience. One of the participants in fact recalled their experience as follows:

“Senior people will have a good understanding of what software design is, but younger people might struggle. When we were in college, we had some lecture
about design but it wasn’t as covered as coding. So I’ve learned by following seniors, reading their design documents and participating in discussions” - 14

Perhaps this was just the experience of one individual. However, it resonates with the importance of mentorship in general in software development. For instance, Begel and Simon observed problems new software developers had at Microsoft with communicating design, suggesting that the education of software developers might need to include more opportunities to be mentored by practicing software developers in the context of course work [8].

Another participant made the following comment, which reinforces this lack of education about critical aspects of design:

“Do students really learn how to make good design decisions when faced with a specific issue? Do we teach people how to think about design? The answer is typically “no, we don’t teach that”. If you ask the question “do we know how to teach people how to think about design?”. Think about all the papers you read, the answer is likely no.” - A2

Improving the teaching of design will require additional research to understand better approaches to facilitate the acquisition of sufficient design knowledge by junior developers. A possible direction could be to shift the focus from the design itself to the rationale behind it, as theme five highlighted how an experienced designer’s knowledge comes from having seen over the years what kind of designs were successful and reflecting on what caused them to be ’good design’.

Alternatively, a more practical approach could involve the development of tools to provide junior developers access to archives of design information related to the project that they are working on, allowing them to acquire knowledge of the rationale behind design decisions of the system at the same time that they are learning about the system itself. These tools could then be evaluated to understand if developers are able to acquire knowledge about design more quickly.

Vocabulary for design is weak. Participants indicated that design occurs in many dif-
different forms across many different phases of development (theme one). Participants also indicated that design depends on the context of development: more complex software requires more design (theme three). Moreover, as one participant directly explained, the kind of design depends on the role of individuals in the development:

“you have project leader, software architects, interaction designers [...] so they’re all designers in my mind.” - These are good examples of people covering different examples of design” - A1

Though it is possible to correlate some kinds of design that occur between participants, overall, they struggled to describe different kinds of design, often referring to particular kinds of diagrams, whether formal or informal, as in:

“High level we have things like use cases, architecture [...] Then on the low level, if the component has some backend database you come up with some schema diagram and things like that.” - I4

Further study of software design may require building a more consistent vocabulary for design kinds, activities and processes. Getting to a more consistent vocabulary may require investigating what kinds of design exists, introducing terms or vocabulary for the designs and then empirically investigating design in practice building on this new vocabulary. Having consistent vocabulary to refer to various aspect of software design may also be useful to support communication between developers, as different parties could now understand immediately what specific kind of design an exchange is about.

**Identifying design.** Theme two describes how design is about decision making was discussed by many participants (10 out of 16 or 62.5%). This theme provides a concrete path towards identifying design information. While describing ‘what’ design might be, the theme does not indicate where such information might appear. Theme four helps provide a target for ‘where’ design might be found, namely in forums that support communication between developers.
3.4 Threats to Validity

Internal and Construct Validity

As an interview study, the major source of data collected was the responses of participants to a set of semi-structured interview questions about design. The data may be incomplete if a participant was not prompted in a way to get a full view of their perspectives on design. The perceptions they relate may also not be fully indicative of their experiences with software design. As interviewer, we mitigated these threats by engaging with participants in a dialogue about design, exploring their perspectives in depth.

The participants were recruited in two ways: first by inviting individuals who had published on software design, and second as a snowball from other participants and the authors own contacts. Both of these recruitment mechanisms were approved in the ethics for the study. The use of snowballing and personal contacts may cause some views of software design to be over-represented. The diversity of participants’ experience and geographical location helps to mitigate this threat.

The academics interviewed outnumbered the practitioners by a significant amount. Additionally due to our recruitment approach our practitioners participants have been involved with academic research. For these reason our results may be biased toward an academic point of view. We mitigated this threat by labelling each card in the coding process depending if it came from an academic or a practitioner participant to help highlight comments coming from practitioners.

The themes identified through open coding may have been impacted by the coders own views of software design. By making sure that both coders individually coded the data and then compared results, the impact of individual views is reduced. The data set on which the coding was performed is available for others to inspect the results[^5].

[^5]: A copy of the transcripts can be found at https://github.com/vivianig/dissertation-datasets
External Validity

An interview study method was used to allow an in-depth exploratory study of a breadth of perspectives about software design. An interview study design limited the number of participants, impacting the generality of the results.

3.5 Summary

Software design has always been recognized as an important aspect of software development. Unfortunately it has become common practice to use the term to refer to quite different concepts, without specifying which one is being considered. This chapter presents an investigation study aimed at understanding how developers and researchers view design.

The interview study presented involved 16 participants, from both academia and industry, and focused on the topic of software design. The analysis revealed the complexity of how design is viewed. Five major recurring themes were found in the responses of participants, describing design as: 1) belonging to multiple phases, 2) being about making decisions, 3) being dependent on context, 4) involving communication and 5) benefiting from experience. These observations help validate other reports about software design, not all of which refer to specific data.

These empirical results suggest various avenues to further study the concept of software design. The breadth of software design as seen by our participants call for additional empirical study of design activities to better understand their breadth and to start to develop a vocabulary for enabling clear and consistent study and communication of design concepts. The results of the study also provide a basis for identifying 'what' indicates design and 'where' design information might be found, namely in decisions that are recorded and discussed in communications between developers.
Chapter 4

Locating Design in Pull Request Discussions

Our interviews with software engineering practitioners and researchers (Chapter 3) suggest that design information might be present when decisions are discussed by developers. Fortunately, many discussions between developers now occur in a recorded form, such as in comments attached to issue reports or pull requests. Other researchers have independently shown that these recorded discussions do include design information [24, 147]. In this chapter, we introduce design points as a means of identifying design in recorded developer discussion, and describe the creation of a data set that identifies design points in 10,790 paragraphs from 34 pull requests from three different open source projects. We also introduce a classifier that can locate when a design point occurs in a paragraph of a pull request discussion. Finally, we evaluate the efficacy of the classifier in locating latent design information in developer discussions occurring in pull requests.

4.1 A Design Points Dataset

Instead of presupposing a single notion of design, we opt to focus on separating design information from other kinds of information present in a discussion. For this separation,
we introduce the concept of a *design point*, which we define as:

*a piece of a discussion relating to a decision about a software system’s design that a software development team needs to make.*

Design points include a wide variety of design, encompassing structural, functional and non-functional aspects of the system. Although broad, the concept of design points ensures we are not prematurely limiting the range of design information we consider.

To study design points, we require a dataset that identifies design points in developer discussions. We consider pull request discussions for creating this dataset because pull requests have been shown to include design information [24]. To understand what characteristics might indicate the presence of design points in discussion, we apply an iterative, manual annotation process to pull request discussions from three open source systems. We focus on pull requests with lengthy discussions to maximize the likelihood that design is discussed, as opposed to a pull request that might be quickly accepted by a development team with little discussion. We focus on the *Node.js*[^1], *Ruby on Rails*[^2], and *Rust*[^3] projects because they are large open source projects of different kinds, underlying technologies, and levels of maturity, thus allowing us a diverse perspective on design.

We created an annotation guide to identify design points by first having two individuals (myself and a collaborator) independently annotate the same pull request discussion looking for paragraphs containing design points. Each individual created an annotation guide to document annotations used, steps taken and assumptions made. Next, we perform a consolidation phase, where the independently created annotation guides are merged into a single codebook. During the consolidation phase, we opted to restrict the search of design points to paragraphs. On one hand, working with entire comments is too coarse, since different paragraphs within a comment sometimes discuss different topics. On the other hand, working with single sentences is too fine, as a design point can easily span over multiple sentences. The codebook is available in Appendix A.

[^1]: https://github.com/nodejs/node
[^2]: https://github.com/rails/rails
[^3]: https://github.com/rust-lang/rust
Following the consolidation phase, we annotated four pull requests and compute the Cohen’s Kappa Coefficient across these annotations, which resulted in a value of 0.52. Given this “moderate agreement” level [43, 69] in the annotations, we proceeded to apply the annotation process to more pull requests.

In all, we annotated a total of 34 pull requests: 14 from Node.js, 10 from Rails and 10 from Rust. Of these pull requests, 18 had been successfully merged into the project; 16 had been rejected by project developers. None of the pull requests used in the first phase to create the codebook were included in the final set. The annotated pull request dataset consists of 10,790 paragraphs of which 2,378 contained at least one design point, for a total of 2,475 design points. We summarize the collected data in Table 4.1. For each project, we list the number of paragraphs (#Paragraphs), the number of design points (#DPs), and the percentage of paragraphs containing at least one design point (% Paragraphs with DPs). The results match the finding of Brunet et al. that, in a larger study of 102,122 discussions, found out that roughly 25% of discussions in a project are about design [24].

To provide a sense of this dataset, Figure 4.1 shows an example of a paragraph with a design point from pull request #12422 from the Node.js project. Figure 4.2 provides an example of a paragraph without a design point from the same pull request.

The mere presence or absence of a design point does not provide much context about the design point. For this reasons we opted to also label three kinds of additional information about each paragraph: the developers’ level of expertise, the form of the language used to express a design point, and the logical relationships between design points. We elaborate on each of these kinds of information below.

### Table 4.1: Statistics for the annotated pull requests

<table>
<thead>
<tr>
<th>Pull Request</th>
<th>#Paragraphs</th>
<th>#DPs</th>
<th>% Paragraphs with DPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js</td>
<td>3,963</td>
<td>985</td>
<td>24%</td>
</tr>
<tr>
<td>Rails</td>
<td>3,201</td>
<td>722</td>
<td>22%</td>
</tr>
<tr>
<td>Rust</td>
<td>3,626</td>
<td>770</td>
<td>20%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10,790</strong></td>
<td><strong>2,475</strong></td>
<td><strong>22%</strong></td>
</tr>
</tbody>
</table>
This PR means that we’re unable to safely rely on the existing error handling semantics, namely not swallowing errors by default or affecting post-mortem debugging. Again, I can’t stress how critical the existing error handling semantics are regarding operating our Node stack in the critical Netflix streaming path at scale. It’s imperative to us that the Node runtime continues to work with the existing error handling best practices. After all, we’re relying on Node in some of our most critical systems here at Netflix, where reliability and debuggability are our top priority.

**Figure 4.1:** Example of a paragraph containing a design point from pull request #12422, Node.js project

@yunong I appreciate you speaking up on this matter, but I’ve replied to Julien’s comment above as well as yours at nodejs/CTC#12 (comment). As noted above, that’s a better place for more general concerns about improved Promise support in Node core.

**Figure 4.2:** Example of a paragraph where a design point is not present from pull request #12422, Node.js project

First, given that in a software development project, developers typically have areas of the code in which they have more expertise than others, we wanted to capture information about the role of the paragraph author in the project. This label has three possible values: whether the paragraph author was the owner of the pull request, whether the author was a core project member, or playing another role. Additionally, a second label informs whether the author was invited to the discussion, which was only possible if the author was not the pull request owner.

**Figure 4.3** shows a paragraph from Rails pull request #505 annotated with this information: each annotation starts with ;##. The line marked ;##D54 marks the paragraph below has a design point, identified with ID 54, with a short summary of the design point. The next two lines are information about the author of the comment: the line marked ;##ROLE PM indicates the paragraph was penned by a project member whereas the line marked ;## INV T indicates that the project member was invited to the discussion. (Ta-
As I mentioned earlier, we should simply add an option for users who want to use PATCH to be able to use it, and move on with our lives. We can address the defaults in some far distant release once we actually have real experience with the benefits.

Figure 4.3: Discussion snippet from Rails pull request #505. This paragraph contains a design point and the enhanced labels are shown.

Table 4.2 summarizes the labels annotated.

To give a sense of the final dataset, Table 4.3 reports on the distribution of labels across the paragraphs. Interestingly, the roles associated with design points are roughly split between the three categories: Owner (27%), representing the users that opened the pull request, Member (38%) representing users that are identified as core members of the project, and Other (35%), representing users without any affiliation to the project.

Only a minority of paragraphs containing design points were generated by users invited (INV) in the discussion (37.3%). This seems to indicate that the actors participating in design discussion are both core members of the projects and generic contributors, not officially recognized as part of the team, who join the discussions on their own accord.

The second information considers the ways in which developers use language in the discussions. We chose to label whether a design point takes the form of: 1) an assertion of a solution, 2) an open question, or 3) a closed question, which generally takes the form of an enumeration. In Listing 4.3, the design point takes the form of a solution (i.e., ;##FORM SOL). Table 4.3 shows that the majority (76.6%) of design points appear as Solutions, followed by Questions (21.4%). We found very few (2%) cases of Enumerations (2%).

---

4 The codebook can be found with the dataset at https://www.cs.ubc.ca/~vivianig/annotation.zip
5 The definition of a core member can be found in the codebook in Appendix A.
Table 4.2: Labels applied to paragraphs from pull requests

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>D[ID][Summary]</td>
<td>Identify a design point using a unique number and a descriptive summary.</td>
<td></td>
</tr>
<tr>
<td>ROLE[Role]</td>
<td>Define the role of the author of the paragraph containing the design point.</td>
<td>Pull request Owner, Core Project Member, Other</td>
</tr>
<tr>
<td>INV[Invited]</td>
<td>Indicate if the author of the design point was invited to the discussion (i.e., was tagged in the discussion before her first comment).</td>
<td>True or False</td>
</tr>
<tr>
<td>FORM[Form]</td>
<td>Define the form the design point takes inside the paragraph.</td>
<td>Solution, Open Question or Enumeration</td>
</tr>
<tr>
<td>REL[Relationship]</td>
<td>Indicate whether the design point is related to a previously posted design point.</td>
<td>Elaborates, Generalizes or Reframes another design point, or is New</td>
</tr>
</tbody>
</table>

indicating that there was little evidence of developers brainstorming multiple solutions when posting a comment.

Finally, we were interested in whether there is logical structure in a discussion [151]. We used the REL label to capture the logical structure of the discussion by making explicit the relationships between design points. Because design points in an asynchronous discussion should only be related to design points previously added, this annotation takes into consideration only previous design points. Some design points may have no relationships to previously seen design points, and are labeled “new”. Design points may “elaborate” on a previous point and respond to concerns or expand on premises, or “generalize” previous design points and come to a conclusion. Finally a design point can “reframe” another design point if it highlights different aspects of it, without generalizing or elaborating it. In Listing 4.3 the design point annotated elaborates a previous design point paragraph that is labeled as D43.
Table 4.3: Annotation distribution

<table>
<thead>
<tr>
<th>Annotation Dimension</th>
<th>Label</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE</td>
<td>Owner</td>
<td>669</td>
<td>27%</td>
</tr>
<tr>
<td></td>
<td>Member</td>
<td>949</td>
<td>38%</td>
</tr>
<tr>
<td></td>
<td>Other</td>
<td>857</td>
<td>35%</td>
</tr>
<tr>
<td>INV</td>
<td>True</td>
<td>924</td>
<td>37%</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>1551</td>
<td>63%</td>
</tr>
<tr>
<td>FORM</td>
<td>Solution</td>
<td>1897</td>
<td>77%</td>
</tr>
<tr>
<td></td>
<td>Question</td>
<td>530</td>
<td>21%</td>
</tr>
<tr>
<td></td>
<td>Enumeration</td>
<td>48</td>
<td>2%</td>
</tr>
<tr>
<td>REL</td>
<td>New</td>
<td>1065</td>
<td>43%</td>
</tr>
<tr>
<td></td>
<td>Elaborate</td>
<td>889</td>
<td>36%</td>
</tr>
<tr>
<td></td>
<td>Generalize</td>
<td>74</td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td>Reframe</td>
<td>447</td>
<td>18%</td>
</tr>
</tbody>
</table>

As Table 4.3 shows, we found that large numbers (43%) of design points are either unrelated to others or Elaborate on previous design points (36.2%); one interpretation of these values is that a large number of design points have at least one followup. Reframes are less common (18.1%) and Generalize are rare (3.9%).

Data detailing the annotation of the 10,790 paragraphs and all the remaining data used in Sections 4.1 is available at https://github.com/vivianig/dissertation-datasets.

4.2 Locating Design Points Automatically

To be able to access and leverage design points in written developer discussions, one needs to be able to automatically locate them. With this goal in mind, we consider two research questions:

**RQ1** Can we effectively locate design points automatically in discussions on pull requests?

**RQ2** Which features are most important to locate design points?
Building on the dataset described in the previous section, we developed a machine learning approach for locating which paragraphs in a pull request discussion contain a design point. We begin by describing the features that are potential indicators of the presence of a design point in a paragraph (Section 4.2.1). We then describe the approach we took to build and evaluate various classifiers (Section 4.2.2) before presenting the results (Section 4.2.3). In evaluating classifiers, the focus was on the first research question (RQ1) which asks whether design points can be effectively located automatically. In this context effective means whether the design points can be located accurately and whether the classifier can apply across different software projects. A classifier that locates design accurately, but requires training for every new project is not effective, as it would not be applicable to real-world scenarios. To ensure effectiveness, the classifier was evaluated on projects that were not used for training it.

4.2.1 Features

The manual annotation process found a variety of information of interest to characterize a design point, such as the role of the author of the comment containing a paragraph. We combine this information with the knowledge of the literature of similar techniques to define 19 features on which to investigate a machine learning approach. Table 4.4 summarizes the 19 features that we investigate, grouped in 4 dimensions: process, position, text, and content, while Table 4.5 describes the rationale for each feature.

Process Dimension:

Similar to discussions in other contexts, such as emails [37], pull request discussions involve multiple individuals in different roles. For example, a participant may be heavily involved in the project and considered a core developer [159] or the individual may have been asked to join the discussion because someone already involved in the discussion believes their expertise is needed. We observed how these process aspects varied per paragraph of discussion through the annotations we performed (Section 4.1). Based on this information, we investigate three process features: whether the paragraph author is a project member (isProMem), whether the paragraph author was invited to the discussion
Table 4.4: Features used to identify design points in a paragraph

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Feature</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>IsProMem</td>
<td>Boolean</td>
<td>Whether the author is a core project member</td>
</tr>
<tr>
<td></td>
<td>IsInvited</td>
<td>Boolean</td>
<td>Whether the author was invited to the discussion</td>
</tr>
<tr>
<td></td>
<td>IsOriginal</td>
<td>Boolean</td>
<td>Whether the author submitted the pull request</td>
</tr>
<tr>
<td></td>
<td>NumCom</td>
<td>Integer</td>
<td>Number of comments posted by the author in the discussion</td>
</tr>
<tr>
<td>Position</td>
<td>PosInPr</td>
<td>Integer</td>
<td>Position of the containing comment in the discussion</td>
</tr>
<tr>
<td></td>
<td>PosInCom</td>
<td>Integer</td>
<td>Position of the paragraph in the containing comment</td>
</tr>
<tr>
<td>Text</td>
<td>NumOfWords</td>
<td>Integer</td>
<td>Number of words in the paragraph</td>
</tr>
<tr>
<td></td>
<td>HasCapWord</td>
<td>Boolean</td>
<td>Whether the paragraph contains any capitalized words</td>
</tr>
<tr>
<td></td>
<td>HasShould</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “shall/should”</td>
</tr>
<tr>
<td></td>
<td>HasHow</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “how”</td>
</tr>
<tr>
<td></td>
<td>HasWhat</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “what”</td>
</tr>
<tr>
<td></td>
<td>HasWhy</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “why”</td>
</tr>
<tr>
<td></td>
<td>HasCan</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “can/could”</td>
</tr>
<tr>
<td></td>
<td>HasMay</td>
<td>Boolean</td>
<td>Whether the paragraph contains the word “may/might”</td>
</tr>
<tr>
<td></td>
<td>NumQues</td>
<td>Integer</td>
<td>Number of questions in the paragraph</td>
</tr>
<tr>
<td>Content</td>
<td>NBScore</td>
<td>Real Number</td>
<td>Naive Bayes score of the paragraph</td>
</tr>
<tr>
<td></td>
<td>NBMScore</td>
<td>Real Number</td>
<td>Naive Bayes multinomial score of the paragraph</td>
</tr>
<tr>
<td></td>
<td>COMPScore</td>
<td>Real Number</td>
<td>Complement Naive Bayes score of the paragraph</td>
</tr>
<tr>
<td></td>
<td>RFScore</td>
<td>Real Number</td>
<td>Random forest score of the paragraph</td>
</tr>
</tbody>
</table>

45
## Table 4.5: Rationale behind each feature used to identify design points in a paragraph

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Feature</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>IsProMem</td>
<td>The role of the author of the comment may influence the likelihood that the paragraph contains design point. A core project member [159] or an invited developer is more likely to have the expertise needed for involving in the pull request discussion.</td>
</tr>
<tr>
<td></td>
<td>IsInvited</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IsOriginal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NumCom</td>
<td>The number of comments relates to the level of the activity of the author, which can characterize the role of the author [122].</td>
</tr>
<tr>
<td>Position</td>
<td>PosInPr</td>
<td>Positional features were found to be useful for classifying discussions [97] [115]. We assume that paragraphs</td>
</tr>
<tr>
<td></td>
<td>PosInCom</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>NumOfWords</td>
<td>A larger paragraph is more likely to be informative and it is more likely to contain design point.</td>
</tr>
<tr>
<td></td>
<td>HasCapWord</td>
<td>Developers may use capitalized words to highlight the key points they want to express.</td>
</tr>
<tr>
<td></td>
<td>HasShould</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasHow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasWhat</td>
<td>As observed in Section 4.1, developers may use assertive or speculative language when discussing design. The words “should”, “how”, “what”, “why”, “can” or “may” are indicative of speculative language [73], and their presence in a paragraph may increase its likelihood of containing a design point.</td>
</tr>
<tr>
<td></td>
<td>HasWhy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasCan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasMay</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NumQues</td>
<td>The presence of a high number of interrogative sentences [73] indicates the presence of speculative language (cf. previous).</td>
</tr>
<tr>
<td>Content</td>
<td>NBScore</td>
<td>The textual content of the paragraph may indicate the presence of a design point. NBScore, NBMScore, COMPScore and RFScore are the likelihood scores of a paragraph containing a design point calculated based on the textual content of the paragraph by four classifiers [57] [149] [156] (see Section 4.2.1).</td>
</tr>
<tr>
<td></td>
<td>NBMScore</td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMPScore</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RFScore</td>
<td></td>
</tr>
</tbody>
</table>
(IsInvited), and whether the paragraph author is the pull request originator (IsOriginal). An additional fourth feature, NumCom, relates to the level of activity of the author of the paragraph in the discussion; this feature is similar to one used by Rodeghero et al. when building a classifier for turning conversations into user stories [122].

Position Dimension:

Machine learning-based classification has been applied to discussions to perform other tasks, such as extractive summarization (e.g., [97]). As well, the features used for general email discussions have also been found to be useful for text produced as part of the software engineering process (e.g., [115]). In a similar fashion, the features belonging to the position dimension were inspired by the positional features used by such classifiers: PosInCom measures the position of the paragraph in a comment, while PosInPR measures the position of the comment containing the paragraph in the entire pull request discussion.

Text Dimension:

The features of this dimension measure textual characteristics of a paragraph as indications for the presence of a design point. Building on features defined by Correa and Sureka, we consider the size of the paragraph (NumOfWords) and the presence of capitalized words (HasCapWord) [30]. Based on heuristics found to be useful by Li et al., we also consider the presence of words indicative of speculative language (e.g., HasShould, HasMay, etc.) [73]. Finally, we also measure the density of interrogative sentences directly, by counting the number of questions in a paragraph (NumQues) using the heuristic approach developed by Li et al. [73].

Content Dimension:

The vocabulary that an author uses in a paragraph can also indicate the presence of a design point. With the features in the text dimension, we considered simple vocabulary features, such as using specific speculative words. At the same time, we also want to consider whether a wider set of words might indicate the presence of a design point. Given the broad vocabulary used in any discussion, one cannot simply treat every word that appears
in a pull request discussion as a separate feature. In the data mining literature, the large amount of words indicates the problem of the curse-of-dimensionality [57]. To attack this problem, we follow previous studies (e.g., [149, 156]) by converting the words in a paragraph to a simple textual score, called a “content score”. The higher the value of the content score, the higher the chance that the paragraph will contain a design point.

The computation of the content score is based on the procedure depicted in Figure 4.4. First, the text in paragraphs is pre-processed using standard approaches, including removing stop-words (e.g., “and” and “the”) and stemming (e.g., reduce “reading” and “reads” to “read”). Second, for each paragraph in the dataset (Section 4.1), a token is extracted and a word frequency table is created, which records the number of times each token appears across all paragraphs. Third, to help avoid biasing the design point classifier (hereafter referred to as the ”main” classifier), we train three helper classifiers $C_1$, $C_2$, and $C_p$, each of which predicts a content score from the tokens of a given paragraph. Specifically, $C_1$ and $C_2$ are used to compute the content score for half of the paragraphs respectively in the training set $T$ used to train the main design point classifier, while $C_p$ is used to compute the content score for paragraphs in the test set $P$ of the main design point classifier (label C in Figure 4.4). This strategy of multiple classifiers avoids bias from the training sets [149]; otherwise, our models may lead to optimistic (unrealistic) values for the textual scores.

To compute the content score for paragraphs in $T$, the paragraph is split into two subsets $t_1$ and $t_2$ using stratified random sampling so that they each have the same distribution and number of paragraphs with and without design points. Then, $C_1$ is trained using $t_1$ and used to predict the content scores of the paragraphs in $t_2$ to produce $t'_2$ (label A in Figure 4.4). For each paragraph in $t_1$, $C_1$ computes the content score based on its word frequency table and whether the paragraph contains a design point or not. Conversely, $C_2$ is trained using $t_2$ and used to predict the content scores of the paragraphs in $t_1$ to produce $t'_1$ (label B in Figure 4.4). This creates the new training dataset $T' = t'_1 \cup t'_2$, where paragraphs are represented by their content score. In other words, the content score of each paragraph in the training set is generated from either $C_1$ or $C_2$.

To compute the content score for paragraphs in the test set $P$, we train the helper classifier
Figure 4.4: Diagram of how our approach works, including the calculation of the content scores using $C_1$, $C_2$, and $C_p$. The sections labelled as A and B compute the content scores that are used to train the classifier, while the section labeled C computes the score for the testing set. Finally, section D represents the main classifier that predicts the final result.
Specifically, $C_p$ is trained on $T$ and used to predict the content score for a new unseen paragraph from $P$, identified as $P'$ (label A in Figure 4.4). The difference with $C_1$ and $C_2$ is that when predicting a new content score, $C_p$ does not know whether the new paragraph contains a design point. In other words, the main design point classifier uses the content scores in $T'$ created by $C_1$ and $C_2$, and invokes $C_p$ to get the content scores of paragraphs not seen in the training set, as shown in label D in Figure 4.4.

We investigate four different algorithms for the classifiers, using the same algorithm for each of $C_1$, $C_2$, and $C_p$: a naive Bayes classifier [89], a naive Bayes multinomial classifier [89], a complement naive Bayes classifier [117], and a random forest classifier [21]. The implementation of these algorithms was provided by the default Weka distribution [56]. This approach leads to four features: $NBScore$, $NBMScore$, $COMPScore$, and $RFScore$.

The four classifiers can leverage the textual content of a paragraph in different ways. The use of different classifiers helps better characterize the textual content of a paragraph, combining the likelihood scores from different classifiers. This approach can also help dealing with the inherent biases of each classifier, such as the strong independence assumption of the naive Bayes classifier.

### 4.2.2 Approach

To determine whether a paragraph contains a design point, we opted to investigate four different classifiers: Random Forest (RF)\(^6\) [21], Naive Bayes (NB) [89], Support Vector Machine (SVM) [57] and K-Nearest Neighbor (KNN, with $K = 5$) [57]. The implementation of each algorithm was provided by the Weka distribution.

We use the Receiver Operating Characteristic Curve score [40] (AUC) to evaluate the performance of our prediction models. AUC scores range from 0 to 1, with 1 representing perfect prediction performance. A classifier using random prediction has an AUC score of 0.5; thus if the AUC score for a classifier is larger than 0.5, it performs better than random prediction. AUC measures the prediction performance across all the thresholds and it is

\(^6\)To reduce the effect of overfitting, in our study, the depth of the decision trees built in random forest is set to 10.

50
Table 4.6: Average AUC for classifiers built on all of the proposed features (Random Forest, Naive Bayes, SVM, and KNN), and content classifiers (Content\textsubscript{NB}, Content\textsubscript{NBM}, Content\textsubscript{COMP}, and Content\textsubscript{RF}) (mean ± Standard variance)

<table>
<thead>
<tr>
<th>Approach</th>
<th>AUC</th>
<th>Approach</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.87±0.06</td>
<td>Content\textsubscript{RF}</td>
<td>0.80±0.04</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.84±0.08</td>
<td>Content\textsubscript{NB}</td>
<td>0.73±0.04</td>
</tr>
<tr>
<td>SVM</td>
<td>0.67±0.15</td>
<td>Content\textsubscript{NBM}</td>
<td>0.67±0.09</td>
</tr>
<tr>
<td>KNN</td>
<td>0.77±0.10</td>
<td>Content\textsubscript{COMP}</td>
<td>0.56±0.12</td>
</tr>
</tbody>
</table>

Insensitive to cost and class distributions [112]. Lessmann et al. recommended that AUC should be used as the primary accuracy indicator to compare the performance of prediction models [71]. Romano et al. concluded that a prediction model with an AUC score above 0.7 is often considered to have adequate classification performance [125]. In a recent work, Tantithamthavorn and Hassan recommended that threshold-independent measures (e.g., AUC) should be used in lieu of threshold-dependent measures (e.g., F1-score), since F1-score is sensitive to the used probability threshold [142]. In this context, the AUC score measures the probability the classifier will rank a randomly selected paragraph with a design point higher than a randomly selected paragraph without a design point.

4.2.3 Results

To determine if the classifier can effectively locate design points in developer discussions (RQ1), we evaluate the performance of each of the four classifiers using a cross-project approach. In this approach, we train a prediction model using annotated pull requests from two projects and test the classifier using the pull requests in a third project. This approach helps consider the evaluation of the classifier under realistic conditions as it is conceivable that data from a small number of projects might be manually labeled (as done for the dataset presented in Section 4.1) as long as the classifier trained from such data is applicable across a range of projects.

To help determine if a simpler set of features might suffice for detecting design points, we also build predictions models based on term frequency scores computed by treating each
paragraph as a bag-of-words. Following Section 4.2.1, we leverage naive Bayes, naive Bayes multinomial, complement naive Bayes, and random forest to build the prediction models, and we refer to these four classifiers based on the content of paragraphs as content classifiers, and denote them as Content$^{NB}$, Content$^{NBM}$, Content$^{COMP}$, and Content$^{RF}$, respectively.

Table 4.6 shows the average and distributions of AUC for each of the four classifiers (RF, NB, SVM, and KNN) based on all of the features we proposed, compared with the four content classifiers on the 10,790 paragraphs from the 34 pull requests in our dataset.

The AUC values are averages from running and averaging over all combinations of cross-project validation. Overall, Random Forest (RF) based on all of the proposed features achieves the best results with AUC of 0.87. We confirmed the dominance of the RF approach using a Wilcoxon signed-rank test with Bonferroni correction; the improvement of RF based on all of the features over other approaches is statistically significant at the 95% confidence level. Compared with our proposed approach with the content classifiers, we still find that RF based on all of the features improves the statistical significance. In practice, RF based on all of the proposed features is the best choice for identifying design points in pull requests.

We believe this result is due to the nature of random forests. A random forest is constructed using a multitude of decision trees, each learned separately at training time, and outputs the mode of the prediction of each tree. By doing this, RF is able to overcome overfits and is robust against noises and outliers in the training dataset.

Our second research question (RQ2) asks which features are important in enabling the classifier to discern which paragraphs have a design point. To answer this question, we use a four step process.

**Step 1: Correlation Analysis.** We first look for collinearity among the features by using variable clustering analysis.

Out of the 19 features, there are 4 features belonging to two groups of variables that have correlations larger than 0.7. The two groups are $\{NBScore, RFscore\}$, and $\{NBScore, RFscore\}$. 
Table 4.7: Importance of the 17 features as ranked according to the Scott-Knott ESD test results. The second and third columns correspond to P-values, and Cliff’s Delta for the features. Statistically significance at confidence level of 95% and non-negligible effect size are in bold.

<table>
<thead>
<tr>
<th>Group</th>
<th>Feature</th>
<th>P-value</th>
<th>Cliff’s Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IsInvited</td>
<td>&lt;0.001</td>
<td>0.36 (Medium)</td>
</tr>
<tr>
<td>2</td>
<td>NumOfWords</td>
<td>&lt;0.001</td>
<td>0.56 (Large)</td>
</tr>
<tr>
<td>3</td>
<td>PosInPR</td>
<td>&lt;0.001</td>
<td>-0.15 (Small)</td>
</tr>
<tr>
<td>4</td>
<td>NBMScore</td>
<td>&gt;0.05</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>NumCom</td>
<td>&lt;0.001</td>
<td>-0.10</td>
</tr>
<tr>
<td>6</td>
<td>PosInCom</td>
<td>&lt;0.001</td>
<td>-0.17 (Small)</td>
</tr>
<tr>
<td>7</td>
<td>NBScore</td>
<td>&gt;0.05</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>HasCan</td>
<td>&lt;0.001</td>
<td>0.16 (Small)</td>
</tr>
<tr>
<td>9</td>
<td>NumQues</td>
<td>&lt;0.01</td>
<td>0.10</td>
</tr>
<tr>
<td>10</td>
<td>IsProMem</td>
<td>&gt;0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>11</td>
<td>HasShould</td>
<td>&lt;0.001</td>
<td>0.10</td>
</tr>
<tr>
<td>12</td>
<td>IsOriginal</td>
<td>&lt;0.001</td>
<td>-0.09</td>
</tr>
<tr>
<td>13</td>
<td>HasCapWord</td>
<td>&lt;0.001</td>
<td>0.13</td>
</tr>
<tr>
<td>14</td>
<td>HasWhat</td>
<td>&lt;0.001</td>
<td>0.07</td>
</tr>
<tr>
<td>15</td>
<td>HasMay</td>
<td>&lt;0.001</td>
<td>0.06</td>
</tr>
<tr>
<td>16</td>
<td>HasWhy</td>
<td>&lt;0.001</td>
<td>0.04</td>
</tr>
<tr>
<td>17</td>
<td>HasHow</td>
<td>&lt;0.001</td>
<td>0.05</td>
</tr>
</tbody>
</table>

COMPScore}. Randomly removing one feature from each group (RFscore and COMPScore respectively), we are left with 17 features.

Step 2: Redundancy Analysis. Having reduced collinearity among the features, we use R to determine redundant features, which do not have unique signal relative to the other features. With this analysis, we did not find any redundant features.

Step 3: Important Feature Identification. Next, we apply “out-of-bag” (OOB) estimation to estimate the internal error of a random forest used to estimate the internal error of a random forest classifier [155]. The main idea is to permute each feature randomly one by one and see whether the OOB estimates will be reduced significantly or not.

For each run of a 10 times 10-fold cross validation, we get an importance value for each
feature. To determine the features that are the most important for the whole dataset, we take the importance values from all 10 runs and apply the Scott-Knott Effect Size Difference (ESD) test [75, 142, 157].

Table 4.7 presents the importance of the 17 features as ranked according to the Scott-Knott ESD test. We find that IsInvited (whether the paragraph author is invited to the discussion), NumOfWords (number of words in the paragraph), PosInPR (position of the paragraph in the pull request), NumCom (number of comments the developer posted previously in the pull request), NBMScore (naive Bayes multinomial score of the paragraph), and PosInCom (position of the paragraph in the comment) are the six most important features that influence the random forest model. These features help most in discriminating paragraphs with design points from these without design points.

Step 4: Effect of Important Features. To understand the effect of the six most important features, we compare their values in paragraphs with design points and in paragraphs without design points. To analyze the statistical significance of the difference between the two groups of paragraphs, we apply the Wilcoxon rank-sum test [84] at 95% significance level. To show the effect size of the difference between the two groups, we calculate Cliff’s Delta\(^7\) [83], which is a non-parametric effect size measure. A positive effect indicates that a higher level of a feature corresponds to an increase in the likelihood of a paragraph containing design points, while a negative effect indicates that a higher level of a feature corresponds to a decrease in the likelihood of a paragraph containing design points. Table 4.7 presents the p-values and Cliff’s Delta for the 17 features. We notice that IsInvited, NumOfWords, PosInPR, and PosInCom show non-negligible effect size on the two groups of paragraphs with and without design points, while NBMScore shows negligible effect size. Based on the findings in Table 4.7, we conclude that:

- IsInvited has a medium positive effect: invited developers are more likely to propose Design Points.

- NumOfWords has a large positive effect: longer paragraphs have a higher chance to

\(^7\)Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size, respectively.
contain Design Points than shorter ones.

- *PosInPR* has a small, non-negligible negative effect: paragraphs posted earlier in the discussion have a higher chance of containing Design Points than those posted later.

- *PosInCom* has a small, non-negligible negative effect: developers tend to express Design Points at the beginning of a comment.

In summary, we extracted 19 features from pull requests, categorized in four dimensions: process, position, text and content. Experimental results show that random forest achieves the best performance with an average AUC value of 0.87. Moreover, statistical tests show that *IsInvited, NumOfWords, PosInPR, and PosInCom* are the four most important features to locate paragraphs with design points; these features are in the process, text, and position dimensions. It might beneficial to focus in the future on investigating these four features to both simplify and improve the classifier.

### 4.3 Developers and Design Points

An approach for locating design points automatically is much more useful if it can find design points in pull requests for projects on which it was not trained and that still agree with what a human developer would identify as a design point. To investigate whether the classifier we developed generalizes to pull requests from other projects and whether the design points automatically identified agree with a broader set of human developers than the three annotators used to build the classifier, we performed an experiment. The experiment consisted of creating a gold standard dataset based on pull requests from projects not used to develop the classifier and then comparing the results of applying the classifier to the dataset.
4.3.1 Gold Standard

To generate our gold standard dataset, we selected 5 among the most popular projects on Github: Bootstrap\footnote{https://github.com/twbs/bootstrap}, React\footnote{https://github.com/facebook/react}, FreeCodeCamp\footnote{https://github.com/freeCodeCamp/freeCodeCamp}, TensorFlow\footnote{https://github.com/tensorflow/tensorflow}, and Electron\footnote{https://github.com/electron/electron}. From each project, we selected the 10 most commented pull requests and randomly sampled 50 paragraphs for each project, to a total of 250 paragraphs.

We then recruited five Computer Science students to annotate the presence of design points in these paragraphs. Four of the students had between three and six years of experience working in software development, averaging 4.75 years; the remaining student had no significant experience in industry, but has been a teaching assistant for a software engineering course involving a large codebase. Each evaluator was assigned three samples from the dataset, or 250 paragraphs each. Overall, these assignments were made in such a way that each sample was assigned to three evaluators. Each evaluator was given the definition of a design point (Section 4) and asked to mark whether each paragraph contained a design point or not.

From these annotations, we formed a gold standard by selecting only the paragraphs for which all three evaluators agreed either did or did not contain a design point. The resultant gold standard consists of 181 paragraphs (out of 250 paragraphs) for which there was agreement by all annotators. 19 of these paragraphs were identified as design points and 162 as paragraphs without design information.

4.3.2 Classifier Evaluation

Having the gold standard as ground truth, we proceeded to evaluate the classifier against it. We applied the previously trained classifier to the 50 pull request discussions. The classifier was used to predict the presence of design information in every paragraph in the discussions, rather than just those from the gold standard, as the classifier uses information
Table 4.8: Results of the comparison between the classifier and the gold standard

<table>
<thead>
<tr>
<th>Paragraphs</th>
<th>181</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Points</td>
<td>19</td>
</tr>
<tr>
<td>True Positive</td>
<td>13</td>
</tr>
<tr>
<td>False Negative</td>
<td>150</td>
</tr>
</tbody>
</table>

AUC 0.81

Figure 4.5: Example of a paragraph containing a design point from pull request #12422, Node.js project

from an entire discussion and cannot be applied to paragraphs independent from their context. We then extracted whether the classifier predicted (or not) if each of the 181 paragraphs of interest were design points, and compared the resultant predictions to the gold standard.

Table 4.8 shows the results of this evaluation. Overall, the classifier was able to correctly identify 163 paragraphs out of 181 as either being design points or not. The classifier failed to identify correctly 18 paragraphs: 6 of which were missed as design points and 12 which were incorrectly identified as design points.

The classifier tends to create false positives - incorrectly identifying a paragraph as having a design point when it does not - when the language used is similar to design points but is really about implementation details. Figure 4.5 shows an example of a paragraph from the false positives. Note that all three annotators agreed this paragraph does not discuss design. The language used in this paragraph is similar to the one normally seen in design points (such as having a implicit question followed by some rationale), but this particular paragraph is discussing an implementation detail that is not a piece of design information.
The classifier also misses identifying some paragraphs as design points (false negatives); several of these missed design points consist of short sentences, which might be difficult cases for the classifier to identify features as indicating a design point.

The AUC score was 0.81, which is comparable to the results of experiments on the classifier reported earlier in this paper Section 4.2.3. At the same time the precision and recall were calculated respectively at 0.52 and 0.68, indicating that more work is required to improve the predictions of the classifier.

This result provides evidence that the classifier can generalize to pull requests from other projects.

4.4 Using Design Points

Locating design points is useful only if tools can be built to then extract and represent the design point information in ways that improve tasks performed by software development personnel. We describe three kinds of tools that could be built if design points can be located automatically.

As one kind of tool, design point paragraphs might be used to enhance current approaches for recommending reviewers of pull requests. Existing approaches focus on the expertise of reviewers based on the source code they have contributed to the system (e.g., [145, 161]). This information might be augmented with what kinds of design discussions the developers participate in so that if a particular review discussion begins to talk about that design consideration, the developer could be invited. An advantage over source code based recommenders for reviewers is more ability to provide recommendations within the current focus and topic of the review discussion.

Another possible tool would be to reconstruct the state of the design of the project. Open source software projects often feature thousands of pull requests over their lifespan: for example, the Rust project has had nearly 25,000 pull requests on GitHub. Learning about the design of the system by reviewing these pull requests is impossible for a newcomer. For an existing contributor, it is also hard to keep up with the state of the design based
on latent information in the pull requests. One possible approach is to investigate the extraction of design topics. In earlier work, we described what design topics might be determined from design points identified manually [152], such as *maintainability*, testing or performance. With an automated approach to design point location, design topics could be extracted from more discussions and used to create an easily accessible archive of design information.

Finally, links could be introduced automatically between pull requests to help developers traverse similar design topics in discussions. At present, developers can be unaware of relevant discussions. For instance, sometimes, pull requests are closed due to being duplicates of other pull requests, potentially losing access to valuable discussions [53]. At other times, the discussion on a pull request can transition over time to a topic already covered in detail in the past; contributors might not be aware of the other discussions. To aid developers in following and finding relevant discussions, a tool could be built to automatically reference previous discussions that might be relevant to the current pull request or topic being discussed.

### 4.5 Threats to Validity

To investigate latent design information in developer discussions, we created a dataset of design points (Section 4.1) and used the resultant dataset as a base for a supervised learning approach (Section 4.2). We then compared the results of the classifier to a gold standard formed from human considerations of discussions on projects on which the classifier had not been trained (Section 4.3). There are threats to validity that arise for each of these approaches.

*Internal and Construct Validity.* For the annotation process, we had a total of three individuals involved in labeling the existing design points (Section 4.1). A shared codebook was developed and used to annotated over ten thousand paragraphs with five labels. This annotation process helped inform the occurrence of various kinds of information that could be used in a supervised learning process. For the classifier, we relied on information, such as **ROLE** extracted automatically from the discussions. We leave the investigation of more
difficult to extract information, such as the language used and structure of design points, for future work (Section 4.4).

A threat related to the annotation process is that the Cohen’s Kappa Coefficient was measured at 0.52, indicating multiple instances of disagreement between the annotators. Nonetheless, this value is considered “moderate agreement” and we considered it satisfactory to continue.

We formed the gold standard used to evaluate the classifier (Section 4.3) with five students. To mitigate any risk of a lack of development knowledge in these students, we chose individuals with extensive coding and design experiences; four out of five had spent many years working as software developers.

External Validity The dataset annotated and used by the classifier comprises 10,790 paragraphs across 34 pull requests from 3 projects. By considering pull requests from 3 projects, we mitigate bias to any particular project.

With the classifier, we considered 19 features categorized into 4 dimensions. In choosing the features to investigate, we focused on features related to development process, discussion structure and the content of discussions to avoid features that might only apply to a specific project. Features related to the content dimension depend on the range of vocabulary represented by the projects on which the classifier is trained; further study is needed to determine how these features perform across a wider range of projects.

4.6 Summary

Developers discuss design issues both in-person [107] and in online discussions [24, 24]. In this chapter, we focus on one kind of online discussion, pull requests, and investigate how to locate the points of the discussion where developers discuss design. We introduced the concept of design points, defined as “a piece of a discussion relating to a decision about a software system’s design that a software development team needs to make”. We apply supervised machine learning to build a classifier that can locate design points automatically in pull requests at the level of paragraphs. This classifier can locate design points with high
accuracy (average AUC score is 0.87).

To demonstrate that the classifier locates design points beyond the dataset we created, we apply the classifier to pull requests on which it was not trained or tested and compared to a gold standard created by five students with development experience using five generic projects from GitHub. We found that the classifier was still able to locate design points in projects it was not trained on with high accuracy (average AUC score is 0.79).

This chapter shows that there is useful design information latent in on-line developer discussion and provides a means to locate this information at a coarse granularity. Future research can determine how to locate more specific and nuanced design information and investigate how to semantically model the information to produce even more useful tools for developers.
Chapter 5

Leveraging Design

Having shown that it is feasible to localize where design information occurs in one form of developer discussions, namely pull requests (Chapter 4), we turn to the question of whether the design points that are identified are useful to developers. For developers to make use of design points, they must be able to interpret the information in a design point and then apply the knowledge gained from the design point to a new task being performed on the system. To explore whether design points are a suitable medium for conveying design information from the history of the system to developers, we report on a user study that focuses on the usefulness of the information in a design point.

We focus in particular on the case of newcomers joining an open source project, as previous research has shown that understanding the architecture and design of a system is often a major barrier for someone to join a project [135]. Even with this challenge, a large amount of contributions to open source projects are made by newcomers [111, 135], who are often not aware of the design decisions that have been made in the past when they attempt to propose a code change [53], potentially causing code quality to deteriorate [105]. If newcomers are able to gain useful design information from design points, they may be able to improve the quality of their submissions and speed the incorporation of their contributions into the system. This scenario thus holds potential value for open source developments and simultaneously tackles a difficult audience as newcomers may have less
ability to interpret design information for a system which they lack deep familiarity.

This study focuses on three research questions:

- **RQ1**: Does access to design points change the behaviour of newcomers?
- **RQ2**: Do newcomers understand the information contained in design points?
- **RQ3**: Are design points able to help newcomers consider more alternatives when making design decisions?

The chapter begins with a description of the study methodology (Section 5.1). Next, we discuss the results of the study (Section 5.2) and threats to the validity of the study (Section 5.3). We conclude with a summary of our conclusions (Section 5.4).

A copy of the questionnaire used in the study can be found in Appendix B. All the data discussed in Section 5.2 is available at [https://github.com/vivianig/dissertation-datasets](https://github.com/vivianig/dissertation-datasets)

### 5.1 Methodology

To understand if design points can be successfully used to help the accomplishment of software development tasks, we perform an experiment that simulates newcomers to a project who are working on a contribution. The study involves posing two development tasks to the newcomer—one without design points and one with design points—and asking the newcomer to choose from a set of potential solutions. The newcomer is also invited to comment about their rationale for the choice, and when design points are presented, to comment on the usefulness of the information in the design points.

#### 5.1.1 Recruitment

We recruited participants through a mixture of recruitment emails and personal contacts. A recruitment email was sent to email lists for graduate students in the Department of Computer Science at the University of British Columbia. We also recruited additional participants through personal contacts.
To participate, an individual had to have at least two years of programming experience. We did not require professional experience as we wanted to simulate the experience of newcomers to a project, who often do not have professional experience[3]. Additionally, as each participant would be included in a raffle for a prize, to meet government requirements for a raffles, we had to limit recruitment to residents of Canada (excluding Québec).

We received 10 response from our recruitment emails. An additional 8 participants were recruited through personal contacts, for a total of 18 participants.

5.1.2 Participants

Sixteen of the recruited participants started the study. Of these sixteen, one stopped halfway through the study, while the remaining fifteen completed the study in full.

We were able to gather demographic information for only twelve of the fifteen participants (80%) as some participants had received an early version of the questionnaire that did not contain the demographic questions. Four of the twelve participants (33%) identified as women and eight (66%) identified as men. Eight of the participants (66%) identified as currently being students, while four (33%) did not identify as students.

All but two of the non students had not had previous professional experience as developers. In average, the students had 6.625 years of professional experience, with a median of 4 years, while the non-students had an average and median of 4.5 years of professional experience.

5.1.3 Study Format

The study was structured in the form of a questionnaire to ease participation.

The questionnaire began with a short preamble in which participants were asked to take the perspective of a newcomer developer who has just began contributing to the Node.js project. Participants were told that they would not have to write any code, but would be choosing from a set number of possible solutions for each of the two tasks they would be
Task One

Task Description

You are a developer who has recently started contributing for the Node.js project. You have found multiple issues opened by third party developers about Node.js throwing errors when attempting to open local files. You started investigating the reports and were able to replicate it correctly: based on your findings, Node.js is unable to correctly open files whose name contains foreign characters. As you know, Node.js uses a low level C library to handle I/O operations, and after a quick check you realized that it supports only UTF-8 as encoding.

Your goal for this task is to find a solution allowing Node.js to correctly handle cases in which files provided by a user are not encoded in UTF-8. Note that, in this context, “user” refers to the third party developer who’s code is providing the files, and not to the final users.

After some time reflecting on the task, you have come up with three possible solutions.

- **Solution 1:** Continue assuming that UTF-8 is the encoding being used and add error handling in the form of providing a message back to the user. This would guarantee that NodeJS continues working as expected, and notifies the user that there’s a problem with their file encoding. The downside is that the user may not know how to handle this problem.

- **Solution 2:** Continue assuming that UTF-8 is the encoding being used, but force convert any filename provided to UTF-8. Nodejs will continue operating as normal and will be able to handle files in other encodings too. Even though you can be certain that filenames will be converted to UTF-8 correctly, you are not certain if any other side effect may happen.

- **Solution 3:** Continue assuming that UTF-8 is the encoding being used, but offer an optional parameter allowing the user to specify which encoding the files are. Nodejs will continue operating as expected and will be able to handle files in other encodings too. The drawback is that the user may not known which encoding to provide or provide it in the wrong way.

Figure 5.1: One of the two tasks that participants saw first

shown. Participants were split into two groups using the randomizer option of the Qualtrics survey tool, resulting in one group of six participants and one group of nine participants.

Participants were shown the two tasks, one at the time, and asked to choose one of the three proposed solutions before answering a few questions. The tasks were created based on existing pull requests and issues from the Node.js project. Neither task had a “right” or “wrong” solution as different solutions present different trade-offs. Participants were
When deciding between different solutions, our choice can be based on tradeoffs that we have identified between the various options.

Did you consider any tradeoff when making your decision? And if so, which ones?

**Figure 5.2:** Following the first task, participants are asked if they considered any tradeoff in their decision

shown the tasks in different order depending on their group to avoid bias. The two tasks were created ad-hoc for this study. Instead of building tasks from scratch, we opted instead to leverage design points to find real problems that we could use to create more realistic tasks. In doing so we decided to limit the search space by focusing only on paragraphs from the 14 pull requests we had annotated from the Node.js project (Chapter 4, Section 4.1).

Figure 5.1 shows one of the tasks that participants saw as part of the study and the possible solutions. For the first task on which a participant worked, they were asked to select one of the three solutions and to explain why they chose it. In a second page, participants were shown the solution they chose and their previous response, and asked if they had taken into account any tradeoff when making their decision.

Before the second task was presented, a participant was presented with a definition of a design point (Figure 5.3), and instructed to imagine that, for the next task, they would have access to a tool that would provide them with a list of design points relevant to the task. Eight design points were then presented, all of which were from the dataset presented in Chapter 4, and had additionally been labelled as design point by the classifier described in Chapter 4. To help participants understand the design points, additional context was

---

2The entire questionnaire is available in Appendix B.
Figure 5.3: Explanation of design points given to the participants, including an example of how they would be shown to them.

The entire questionnaire is available in Appendix B.

5.2 Results

We consider each of the research questions in turn.

5.2.1 RQ1: Newcomer Behaviour with Design Points

The first research question (RQ1) considers whether the presence of design points changes the behaviour of developers. To explore this question, we consider which solution partici-
Table 5.1: Identifiers for each participant in both groups.

<table>
<thead>
<tr>
<th>Group I</th>
<th>Group II</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-1</td>
<td>II-1</td>
</tr>
<tr>
<td>I-2</td>
<td>II-2</td>
</tr>
<tr>
<td>I-3</td>
<td>II-3</td>
</tr>
<tr>
<td>I-4</td>
<td>II-4</td>
</tr>
<tr>
<td>I-5</td>
<td>II-5</td>
</tr>
<tr>
<td>I-6</td>
<td>II-6</td>
</tr>
<tr>
<td></td>
<td>II-7</td>
</tr>
<tr>
<td></td>
<td>II-8</td>
</tr>
<tr>
<td></td>
<td>II-9</td>
</tr>
</tbody>
</table>

Table 5.2: Solutions chosen by participants based on group and task

<table>
<thead>
<tr>
<th></th>
<th>Task A</th>
<th></th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solution 1</td>
<td>Solution 2</td>
<td>Solution 3</td>
</tr>
<tr>
<td></td>
<td>2 (33.3%)</td>
<td>1 (16.7%)</td>
<td>3 (50%)</td>
</tr>
<tr>
<td></td>
<td>I-3 I-5</td>
<td>I-6</td>
<td>I-1 I-2 I-4</td>
</tr>
<tr>
<td>Group I</td>
<td>2 (22.2%)</td>
<td>0</td>
<td>II-2 II-4 II-5 II-6 II-7 II-8 II-9</td>
</tr>
<tr>
<td></td>
<td>II-1 II-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group II</td>
<td>3 (33.3%)</td>
<td>6 (66.7%)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>II-4 II-6 II-7</td>
<td>II-1 II-2 II-3 II-5 II-8 II-9</td>
<td></td>
</tr>
</tbody>
</table>

Pants chose as the most suitable for the given task. We refer to the two tasks as Task A and Task B respectively and use the terminology of groups to identify the participant groupings that had either Task A or Task B first. Recall that the first task in each group did not have access to design points for the task. Thus, Group I received Task A without design points first and then Task B with design points, while Group II had Task B first without design points and then Task A with design points. Table 5.1 shows the identifier with which we refer to each participant.
Table 5.2 summarizes which solution the participants selected. Overall, participants in Group I were more evenly split, for each task, across all solutions. For Group I, at most 50% of the participants settled on one solution for either task. Group II was more consistent with the solutions they picked, with 77.8% picking the same solution for Task A and 66.7% for Task B. Participants in Group 2 also did not choose one of the solution at all, for each task.

This data does not show any strong trend that design points influenced the behaviour of the newcomers.

5.2.2 RQ2: Newcomers Understanding of Design Points

The second research question (RQ2) considers whether newcomers understand the content of design points and whether they have the same understanding of design points. To explore this question, we decided to label their responses to why they chose a particular solution for a task. Participants were asked one question about their solution:

*Using your experience and the information provided, which solution do you think would be best suited for the task? Why did you choose this solution?*

For the task for which design points were not available, participants were asked a second question:

*When deciding between different solutions, our choice can be based on trade-offs that we have identified between the various options.*

*Did you consider any trade-offs when making your decision? And if so, which ones?*

For each group and task, we labelled the responses to each of these questions. Table 5.3 summarizes the results of the labelling process. Each column describes how many occurrences of the same label were found in the responses, counting each label only once for each participant. We identified five labels that describe the rationales provided by the participants. Four of the labels appeared for rationales given for both tasks: namely, solutions
Table 5.3: Results of the labelling of what rationale participants used to justify which solution they chose.

<table>
<thead>
<tr>
<th>Task</th>
<th>Without Design Point</th>
<th>With Design Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group I</td>
<td>Group II</td>
</tr>
<tr>
<td>Task A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User-related</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Maintainability/Compatibility</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Correctness/Robustness</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Complexity</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Performance</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Task B

<table>
<thead>
<tr>
<th>Task</th>
<th>Without Design Point</th>
<th>With Design Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group II</td>
<td>Group I</td>
</tr>
<tr>
<td>User-related</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Maintainability/Compatibility</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Correctness/Robustness</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Complexity</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

being user-related, maintainability or compatibility related, correctness or robustness related or complexity related. A fifth label, rationale related to the performance of the code, appeared only in the responses about rationales for solutions to Task A.

Considering Task A, we can see how Group I, who considered the task without having access to design points, shared three labels for the rationales provided. Two additional rationales were discussed by one participant of Group II each, on top of all the ones discussed by Group I and other members of Group II. Considering Task B, Group II, who did not have access to design points, discussed four different rationales, whereas Group I, who had access to design points, focused exclusively on the rationale of maintainability and compatibility. This data suggests that for at least one of the tasks, Task B, the design points played a role in suggesting a rationale to the newcomers, all of whom understood the same rationale likely from the design points.

As part of the second task performed, we also asked participants:

Were you able to understand the information contained in the design points?
Table 5.4: Categorization of the responses the participants provided to the understanding question, depending if they viewed design points Positively, Neutrally, or Negatively

<table>
<thead>
<tr>
<th>Question</th>
<th>Task</th>
<th>Group</th>
<th>Positive</th>
<th>Neutral</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding</td>
<td>A II</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>B I</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Please explain.

Table 5.4 reports on the results of this question. We categorized the responses in three categories based on the comments by the participants: positive, neutral, and negative. This categorization was made based on the content of their responses: if a participant clearly responded to the question with a Yes or a No before explaining why, the response was categorized, respectfully, as positive or negative. Similarly if the participant did not use those two terms but their wording clearly associated with one of them (e.g., “I was able to follow the information” for yes or “not exactly” for no), then their responses were categorized in the same way. In the remaining cases, in which the response was not clearly positive or negative, the response was categorized as neutral [3]

Seven of the fifteen participants (46.7%) answered that they had been able to understand the information provided, while 5 (30%) answered neutrally and another 3 (20%) negatively. Group II in particular had a largely positive response, with five out of nine (55.5%) participants responding positively. Participant II-6 commented that “There were a few references/links and some jargon that I wasn’t 100% able to understand, but it was still much more useful than not having any information.”, while participant I-1 said that “Yes, but for most of them, I was only able to understand them with the additional context. A few were context-free and made sense without the context, but the context was important for most.”.

Three participants, two from Group I and one from Group II, did not find the design point information understandable. Participant II-9 stated that “No, I was confused about what they were referring to and how the text above and below gave the design point response”

[3] The labels can be found as part of the data available at https://github.com/vivianig/dissertation-datasets
Design Point #8

The benchmark don’t directly use process.hrtime() in the http case, but instead the op/sec reported by wrk in the stdout. The childStart stuff is just for reporting how much time the benchmark actually use. This is only used as a side metric, which I found useful for finding benchmarks that uses to many iterations.

not a lot of data is going through this so don't think it matters, but setEncoding() adds pretty hefty overhead to data events. I don't see why we can't just do chunk.toString() instead.

do you know the time difference between the final data event and the close event? doubt it's much, but want to remove any static in the test.

Figure 5.4: One of the design points, shown for task A to Group II, that refers to potential performance issues even though neither the task nor the solutions mention anything about performance

Design Point #1

I'm not sure I understand what is the difference with the EPERM error on Windows and EACCESS on UNIX.

Isn't one of the biggest gripes about getting rid of existsSync() that it gives them a simple Boolean response? Throwing is also a breaking change on a heavily used API. You could argue that it's a bug fix but I think people will still yell loudly.

No, it's that the others require excess options and stuff.

Figure 5.5: One of the design points, shown for task B to Group I, that discusses how the proposed changed would break a heavily used API

and I-2 echoed the sentiment: "No, many of them (standalone) are completely uninterpretable without the surrounding context”.

We discuss the issue of context for design points in Chapter 7.

5.2.3 RQ3: Newcomers Consideration of Alternatives from Design Points

The third research question (RQ3) considers whether design points help newcomers consider more alternatives when weighing potential solutions.

We consider again the labelling of the rationales described earlier.

As noted earlier for Task A and as shown in Table 5.3, one participant in Group II justified
Table 5.5: Categorization of the responses the participants provided to the usefulness question, depending if they viewed design points Positively, Neutrally, or Negatively

<table>
<thead>
<tr>
<th>Question</th>
<th>Task</th>
<th>Group</th>
<th>Positive</th>
<th>Neutral</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usefulness</td>
<td>A</td>
<td>II</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>I</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td>7</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

their solution choice with a rationale based on performance. As Figure 5.1 shows, neither the task description nor any of the solutions, refers to any performance issues. However, one of the design points (Figure 5.4) that Group II was presented discusses how using setEncoding() can have a significant performance impact. As the only participant who referred to performance had access to design points, it is possible that access to design points may have helped the participant consider a trade-off that they may have not considered otherwise. For Task B, as we noted earlier, all participants in Group I, who had access to design points, focused exclusively on a rationale of maintainability and compatibility. A possibly reason may be found in the design points: many of the design points associated with Task B, such as the one shown in Figure 5.5, were more focused on maintainability as the pull requests they originated from were centered around these topics. It is therefore possible that participants were influenced by the design points and focused their reasoning around this rationale.

As part of the second task performed, we also asked participants:

Did you find the information from these design points to be valuable for the given task? How?

The results of this question are presented in Table 5.5 with the categories determined as described earlier in this chapter. The seven of fifteen (46.7%) participants who answered positively commented that they found the information presented in design points useful to understand the problem and to weigh the different solutions: participant II-2 said that “It makes it easier to decide what solution might be the most suitable for the team/company. and participant II-6 that “I found the information useful on a higher level to give me some
Table 5.6: Representation of how participants sorted the design points they saw. The darker the colour, the more participants sorted that design point in that position, from least useful on the left to most useful on the right.

context for the bug and how fixing it might affect different users/systems. It helped me weigh the different solutions and their trade-offs.”. Other participants (five of fifteen or 30%) did not find the the design points to be useful. In some cases, participants complained that many of the design points did not seem relevant to the task: participant II-4 mentioned that “A bunch of the design points listed don’t feel like they have anything to do with the task at all.”; in others cases, that the design points had not been impactful on their decision, such in the case of II-8: “Not really, I would have chosen a different solution anyway (default encoding from locale that is overridable at each call site) based on prior experience”.

To explore whether participants found the same design points as potentially useful, we
asked participants to order the design points from most useful to least useful. Table 5.6 presents the results. The rows in each table represent the design points for each task. The columns represent the ordering from least useful (leftmost square) to most useful (rightmost square). The intensity of the colour represents how many participants put that specific design points in that position.

For Task A, we can recognize three groups of design points: all participants in this group ranked DP#1, DP#2 and DP#3 as the three most useful design points amongst the ones presented. Similarly, they all marked DP#7 and DP#8 as the two least useful design points. The remaining three DPs were in the middle, with DP#6 tending slightly toward the less useful side.

There is less consistency in viewpoints by participants for Task B. Nonetheless, we can recognize a similar pattern: DP#1 and DP#2 were marked by a large majority among the three most useful design points, while DP#3 and DP#7 where identified largely as the two least useful design points.

Our data shows some evidence that design points may impact the alternatives newcomers consider and that there is some consistency in newcomers identifying some design points as useful or not.

5.3 Threats to Validity

As we did not require the knowledge of a specific programming language or software when recruiting participants, it is possible that the participants did not know much or knew too much about the *Node.js* project and this may have impacted their ability to understand the assigned tasks and proposed solutions. We made this choice intentionally as we wanted to study the behaviour of newcomers to a project. To mitigate the risk of a participant not knowing anything about the project, we included a small description about *Node.js* at the beginning of the questionnaire. Participants were also required to have at least two years of programming experience, further mitigating the risk of not understanding the question posed.
Even thought we did not require prior knowledge of the *Node.js* project, it is possible that some of the participants may have already known the project. We made the decision to not enquire about this kind of knowledge as it may have caused potential participants to not sign up over concerns of not knowing the system. At the same time, we argue that prior knowledge does not directly disqualifies someone from being a newcomer, as they may have used the system before but never contributed to it.

To reduce the learning bias that participants would develop after seeing the first task, we randomized the order in which the tasks were presented to the participants. We made use of a randomizer module within *Qualtrics*, the platform on which the questionnaire was hosted, to assign participants to different groups. As the randomizer has to assign every participant to a group, the split can be uneven due to, for example, a participant opening the questionnaire on a device without actually filling it. As result of hits, 6 participants were assigned to group A (40%) and 9 to group B (60%). We are not able to definitely assess the impact of this threat on the results.

As the design points were exclusively from the *Node.js* project, and the tasks created were based on pull requests and issues from the same project, it is possible that the results may not be directly generalizable to any project. While the tasks are based on *Node.js* and briefly refer to it, they are not tied to the project and *Node.js* could be replaced by other projects without affecting the tasks or solution.

A final threat is that the two tasks had been created starting from existing pull request discussions and design points. This decision was made to create more realistic tasks, but the design points associated to each task may have been written after the discussion on which the task was built, providing information to the participant that they would not have had in a real scenario.

## 5.4 Summary

We considered whether newcomers to an open-source project could understand and make use of design point information when presented with a task and potential solutions for the task. We chose to focus on newcomers as approaches that help newcomers make good
contributions faster can benefit an open-source project. The user study we performed involved fifteen newcomers choosing potential solutions for two development tasks. Though we did not find evidence that design points changed the behaviour of newcomers, we did find some evidence that newcomers can interpret information in presented design points consistently with one another. We also found some evidence that design points could help a newcomer consider different design alternatives when making a design decision.

Our data provides weak evidence that design points might be useful to developers new to a project. Our user study presented automatically discovered design points in a minimalist way. There are many opportunities to investigate other analysis on design points prior to presentation and many other presentation forms. We discuss these possibilities for future work in the next chapter.
Chapter 6

Identifying Useful Design Points

When presented with design points in the context of a task, developers found some design points to be useful and some not to be useful (Chapter 5). To further aid developers, it would be beneficial to automatically classify design points according to whether they are useful. One means of identifying useful design points may be to consider the meaning of the design points. Semantic frames are a general linguistic approach that may help provide an automated way to consider the meaning of design points [14, 42]. We begin this chapter with a general overview of semantic frames (Section 6.1). We then describe an investigation into the applicability of semantic frames to software engineering text (Section 6.2) and introduce a tool, SEFrame, that tailors semantic frames to software engineering text (Section 6.3). We then report on an exploration into applying SEFrame to design points found useful in the study reported on in Chapter 5 (Section 6.5). The development of SEFrame and its evaluation were published in [86].

6.1 Semantic Frames

When a reader extracts information from text they do so based on the roles that words take in a sentence with respect to events of interest [14, 42]. Frame semantics [42] theory provides a lens to explain such a process, where a ‘frame’ is the key unit that assists under-
we could use a leaky bucket algorithm to limit the band-width.

The leaky bucket algorithm fails in limiting the band-width.

These two sentences are lexically similar, making it difficult to use lexicon analysis techniques to categorize them. Di Sorbo’s work categorized both sentences as ‘solution proposal’, even though a more accurate label for the second one would have been ‘problem discovery’.

If we apply frame semantics to these sentences using SEMAFOR, the differences become apparent. Figure 6.1 presents the results of a frame analysis for the sentences. The frames of each sentence (highlighted in grey) represent a triggering event and the frame elements.
(fe) (in red) are arguments needed to understand the event. The enclosing square brackets mark all lexical units, or words, associated with either a frame or a frame element.

In the first sentence, the Using frame captures that an instrument, the leaky bucket algorithm, is manipulated to achieve a purpose, namely to limit band-width. In contrast, in the second sentence, the Success or Failure frame identifies the entity, or agent, that fails to achieve the goal of limiting the band-width.

To the best of our knowledge, there have been only a few uses of frame semantics in software engineering research [6, 61, 67, 85]. These approaches have largely focused on text associated with software requirements, leaving open the question of applicability of the approach to text in documents supporting program comprehension activities. One paper [85] applies semantic frames to such documents, but with the goal of finding patterns in text rather than considering the applicability of the technique to the text. It is thus an open question whether semantic frames can help identify the meaning of software engineering text associated with helping developers build and comprehend programs, such as text in API documents and bug reports.

To determine whether frame semantics are applicable to a broad range of software artifacts used for program comprehension, we introduced SEFrame to tailor semantic frame parsing for program comprehension uses, and considered the following three questions:

1. Do generic frame parsing approaches apply to text used in software engineering artifacts?
2. Does SEFrame produce correct frames?
3. Is SEFrame robust?

Data referenced in this chapter is available at this link\(^1\).

\(^1\)https://figshare.com/s/323a80bd801d45ed7b89

80
6.2 Do Generic Frame Parsing Approaches Apply to Software Engineering Text?

Ideally, frames identified by semantic frame parsers would find appropriate meaning in text related to software engineering artifacts ‘out-of-the-box’ without change. As already seen in the investigation by Alhosan et al. on a small number of software requirement documents [4], this assumption does not hold.

To explore how many frames might need alteration for software engineering text, we ask: “do generic frame parsing approaches apply to text that appears in software engineering artifacts?” Specifically, we investigate the frames that result when a standard parser, SE-MAFOR, is applied to a range of artifacts associated with software engineering activities. We are particularly interested in how the presence of jargon or text specific to software such as method signatures, stack traces, or command-line arguments, might affect the meaning of the frames identified.

To answer this question, my collaborator and I performed an open coding of frames parsed from three different datasets drawn from corpora published in previous software engineering studies. Our open coding analysis comprises the inspection of the 50 most frequently occurring distinct frames that appear in 1,866 sentences.

6.2.1 Method

Datasets For this investigation, we chose artifact types that relate to program comprehension tasks [74, 110]. These artifact types have been the target of many researchers given the prominent role they play in the activities of software developers:

- developers seek Q&A websites, such as Stack Overflow, to analyze and adapt solutions available online for their current development tasks [20, 146, 148, 157, 163]; and

- bug reports serve as a central place for several program comprehension activities, such as documentation, coding, or testing [10, 80, 114].
To minimize any bias in our selection of artifacts, we chose to rely on existing datasets published at major software engineering conferences. For this investigation, we use the first three datasets listed in Table 6.1.

### 6.2.2 Sampling

As the selected datasets comprise more than 53,800 sentences, and given that sentences can contain more than one frame, manually inspecting the frames in each sentence is infeasible. We thus sample a statistically significant number of sentence-frame pairs for inspection.

We follow the procedures described by Bacchelli and colleagues [12][13], statistically sam-
Sampling sentences from each dataset with a 90% confidence level and a 5% error level with respect to the original number of sentences of that dataset. As Table 6.1 shows, sample sizes range from 552 to 653 sentences from each source. We then use SEMAFOR [35] to semantically parse each sentence and obtain its set of frames, i.e., when SEMAFOR is applied to a sentence, it may identify multiple frames for the sentence. We chose to use SEMAFOR because previous studies in the field have used the tool and adopting the same parser facilitates comparison [61, 85].

After applying SEMAFOR to the dataset, we identify 444 distinct frames. Figure 6.2 plots the distribution of all sentences per frame. The distribution follows Zipf’s law with certain frames appearing in a high number of sentences and a long tail where some frames are specific to a few sentences. The most occurring frames appear in 30% (topmost frame) to 10% (top 10th frame) of the sampled sentences. In comparison, the 100th frame appears in no more than 25 sentences, which accounts for 1% of the sentences sampled. We are most interested in those frames that occur frequently and so, we produce the final set of sentence-frame pairs for inspection by sampling sentences for each of the top 50 frames. That is, given all sentences associated to a frame, select a statistically significant number of sentences for inspection.

Our sampling procedures resulted in a set of 1,866 sentence-frame pairs for inspection that comprise all occurrences of the top 50 distinct occurring frames.
6.2.3 Annotation Process

Similar to Alhoshan and colleagues [5], we use an annotation approach which consists of inspecting the frame, its definition according to the FrameNet database\(^2\), and its frame elements and associated lexical units. For each frame, annotators indicated if the frame was correct and, if not, suggested either a replacement from FrameNet or an entirely new frame.

As a first step to get familiarized with the definition of each frame encountered, the annotators read a total of 180 pairs of sentences with their associated parsed frame. Then, the two annotators proceeded iteratively. In the first iteration, 300 sentence-frame pairs were individually annotated by the two annotators. At the end of this iteration, the two annotators discussed potential strategies for the resolution of disagreed frames, resulting in an agreed output schema for the final iteration: a frame was annotated as valid, invalid, or requiring modification (with suggestions for a replacement). At this point, the annotators individually annotated the final set of 1,866 sentence-frames. We did not exclude the previous 300 pairs from this set, as returning to them ensures consistency in how conflicts were resolved.

6.2.4 Results

Table 6.2 summarizes results according to annotation outcomes: valid, invalid, or modify. To assist discussion, we refer to the examples from Table 6.3 by their unique identifiers (UID).

We found that the annotators agreed that SEMAFOR’s parsing of the semantic frames was valid for 35 (64%) of the frames inspected. An annotation of “valid” means that the meaning of frames is similar in both every-day English text and software engineering text. In valid frames, SEMAFOR is also able to correctly identify the predicates, namely the frame elements (in red) and lexical units (enclosing brackets) of software engineering specific text. For example, it successfully identifies ‘turning off XBitHack’ as the *cause* for the observed behaviour in UID\(_{436}\) (Table 6.3). For the *Using* frame (UID\(_{122}\)), it also

\(^2\)https://framenet.icsi.berkeley.edu/fndrupal/frameIndex
Table 6.2: Summary of frames and annotation outcome with percentage of sentences containing the frames in the dataset

<table>
<thead>
<tr>
<th>35 frames (64%)</th>
<th>Valid</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intentionally act</td>
<td>Quantity</td>
<td>Using</td>
<td></td>
</tr>
<tr>
<td>Temporal collocation</td>
<td>Being obligated</td>
<td>Capability</td>
<td></td>
</tr>
<tr>
<td>Cardinal numbers</td>
<td>Causation</td>
<td>Likelihood</td>
<td></td>
</tr>
<tr>
<td>Locative relation</td>
<td>Increment</td>
<td>Relational quantity</td>
<td></td>
</tr>
<tr>
<td>Point of dispute</td>
<td>Existence</td>
<td>Intentionally create</td>
<td></td>
</tr>
<tr>
<td>Attempt</td>
<td>Instance</td>
<td>Frequency</td>
<td></td>
</tr>
<tr>
<td>Desiring</td>
<td>Predicament</td>
<td>Desirability</td>
<td></td>
</tr>
<tr>
<td>Required event</td>
<td>Event</td>
<td>Identicality</td>
<td></td>
</tr>
<tr>
<td>Age</td>
<td>Gizmo</td>
<td>Similarity</td>
<td></td>
</tr>
<tr>
<td>Time vector</td>
<td>Relative time</td>
<td>Awareness</td>
<td></td>
</tr>
<tr>
<td>Measure duration</td>
<td>Scrutiny</td>
<td>Sole Instance</td>
<td></td>
</tr>
<tr>
<td>Possession</td>
<td>Grasp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5 frames (16%)</th>
<th>Modified to Execution</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arriving</td>
<td>Means</td>
<td>Aggregate</td>
</tr>
<tr>
<td>Request</td>
<td>Leadership</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10 frames (20%)</th>
<th>Invalid</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Type</td>
<td>Placing</td>
</tr>
<tr>
<td>Being named</td>
<td>Purpose</td>
<td>Roadways</td>
</tr>
<tr>
<td>Contingency</td>
<td>Connectors</td>
<td>Text</td>
</tr>
<tr>
<td>Usefulness</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

identifies the ‘JNI function’ as the instrument needed for achieving the purpose of creating a certain class.

It is worth noting that even if a frame is considered valid overall, some instances of where that frame is parsed may be invalid. UID<sub>120</sub> in Table 6.3 is an example of an invalid instance for the Using frame, where ‘application’ does not evoke usage. Occurrences of invalid instances represent a small fraction of the data annotated frames and we attribute
Table 6.3: Excerpt of results from annotation procedures

<table>
<thead>
<tr>
<th>UID</th>
<th>Frame</th>
<th>Sentence</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>436</td>
<td>Causation</td>
<td>[Turning off XBitHack in my config] \text{fe:Cause} \text{made} \text{Causation} [this behavior go away] \text{fe:Effect}</td>
<td>Valid</td>
</tr>
<tr>
<td>122</td>
<td>Using</td>
<td>I’m trying to \text{use} \text{Using} \text{a JNI function} \text{fe:Instrument} [to create a Java class] \text{fe:Purpose} and set some properties of that class using the DeviceIdjava constructor method</td>
<td>Valid</td>
</tr>
<tr>
<td>1181</td>
<td>Leadership</td>
<td>Does anyone want to \text{run} \text{Leadership} [a benchmark?] \text{fe:Governed}</td>
<td>Modify</td>
</tr>
<tr>
<td>611</td>
<td>Arriving</td>
<td>I can’t even run this simple tensorflow script, as its result, I \text{get} \text{Arriving} [ImportError: No module named tensorflowpythonclient] \text{fe:Goal}</td>
<td>Modify</td>
</tr>
<tr>
<td>120</td>
<td>Using</td>
<td>Its a desktop standalone Java [application] \text{Using}</td>
<td>Invalid</td>
</tr>
<tr>
<td>1176</td>
<td>Roadways</td>
<td>The \text{command line} \text{Roadways} is what almost every other application will use to build your JAR file</td>
<td>Invalid</td>
</tr>
<tr>
<td>1493</td>
<td>Connectors</td>
<td>This means that at least the string-to-int mapping will stay consistent, even if \text{strings} \text{Connectors} are passing out of memory</td>
<td>Invalid</td>
</tr>
<tr>
<td>263</td>
<td>Being obligated</td>
<td>\text{I have} \text{Being obligated} [two classes in a parent-child relationship]</td>
<td>Disagreement</td>
</tr>
<tr>
<td>692</td>
<td>Type</td>
<td>\text{MQ} \text{fe:Category} [version] \text{Type} 71 on the server</td>
<td>Disagreement</td>
</tr>
</tbody>
</table>
invalid instances to misclassification errors inherent to SEMAFOR\(^3\).

A total of 5 (16%) frames required modification. Annotators observed that the meaning of verbs such as ‘get’ or ‘run’ diverge from FrameNet’s meaning as identified by SEMAFOR. UID\(_{1181}\) illustrates a frame that requires modification. The verb ‘run’ evokes the Leadership frame which is defined as control by a leader over a particular entity or group. However, the sentence related to this frame in Table 6.3 conveys executing an evaluation benchmark. Both annotators suggested renaming the frame to Execution so that it reflects a meaning appropriate to the software engineering domain. The same logic applies to the Arriving frame in UID\(_{611}\).

The 10 frames that were annotated as invalid account for 20% of the inspected data. Annotators found that the presence of source-code related nouns is a common element to the invalid frames. Roadways and Connectors are examples of invalid frames. The sentences for these frames, as shown in Table 6.3, indicate that the nouns that evoke these frames are code-related, such as ‘command line’ and ‘string’, and the frames parsed by SEMAFOR are not suitable for such cases.

Disagreements occurred in scenarios where one annotator suggested removal of a frame while the other said otherwise.

As an example, the annotators disagreed on the resolution for the Being obligated frame (Table 6.3). One annotator indicated that Possession would be a more appropriate frame, while the other indicated that given that only a few sentences with the frame were invalid, the situation is similar to a misclassification error inherent to using the SEMAFOR tool. For other cases, such as in Type, annotators observed that a single lexical unit was not enough to justify a new frame and thus, annotators agreed that the frame was invalid.

Out of the top 50 occurring frames detailed in Table 6.2, 35 (64%) of them apply to text from software engineering without modifications, 10 (20%) are invalid and 16% require modification\(^4\).

\(^3\)SEMAFOR has an average precision of 75.54% for the FrameNet lexicon
\(^4\)Detailed results are available in our replication package [https://figshare.com/s/323a80bd801d45ed7b89](https://figshare.com/s/323a80bd801d45ed7b89)
6.3  **SEFrame**: Tailoring FrameNet for Software Engineering

Our initial investigation found that modifications are needed to SEMAFOR, and some of the frames defined in FrameNet, to support appropriate parsing for software engineering text. We describe **SEFrame**, a tool we have built that tailors SEMAFOR for software design and related software engineering text.

We started the design of **SEFrame** by gathering all frames that required modification or that are invalid and their respective lexical units. Then, we sought to identify patterns in the lexical units from these frames that would produce a simple set of heuristics to be used by **SEFrame** to refine the frames extracted by SEMAFOR.

The first pattern observed by the annotators is that if the lexical unit of a frame is a verb with a particular meaning in the software engineering domain (e.g., run, call, etc), the frame requires modification. A second pattern arises from frames evoked by source-code related nouns, which are deemed as invalid. Table 6.3 provides examples for such patterns where the verb ‘get’ in *UID*611 and the word ‘string’ in *UID*1493 are not correct for the Arriving and the Connectors frames, respectively.

The new proposed frame—Execution—addresses verb related modifications. **SEFrame** makes verbs such as get, return, call, request, run and process evoke the new frame instead. Interestingly, this change applies to all the frames that require modification in Table 6.2 since all of their verbs evoke executing system calls or running some procedure.

To address the second identified pattern, **SEFrame** removes any frames that are marked as invalid in Table 6.2. Annotators identified that invalid frames are often related to code-related nouns what could lead to the creation of a reference list for ignored lexical units. Creating such a list poses the question of where code elements come from and whether the list is complete. Therefore, **SEFrame** opts to simply discard the frame rather than referring to which specific nouns and code-related terms would trigger removal.

Intuitively, **SEFrame** can be seen as a Decorator [46] that extends the functionalities of the SEMAFOR tool. Its current version is written in Python and all its dependencies are
6.3.1 Does SEFrame Produce Correct Frames?

For SEFrame to be usable by software engineering researchers, it must be seen as parsing frames that are meaningful—correct—by more than those who defined SEFrame (i.e., the two annotators). To explore if the results produced by SEFrame are meaningful, we asked 10 evaluators, distinct from the annotators in the initial investigation (Section 6.2), to assess the correctness of semantic frames parsed from sentences selected for all datasets shown in Table 6.1.

Method We applied SEFrame to the datasets listed in Table 6.1. To provide a more generalizable result, we extended the first three datasets used to create SEFrame with a fourth dataset, APIPatterns, which has text extracted from Java documentation (Table 6.1). We include a new dataset as a first step towards investigating if and how frames apply to other types of sentences and also because API documentation is a resource that developers commonly refer to as part of a software development task [81, 119]. For this study, we considered the 36 frames that are included in SEFrame.

The 10 evaluators were all currently graduate students in an English-language based Computer Science program and most of them had at least some previous industry experienced, averaging around 1.5 years. All evaluators were remunerated for their time spent on this work.

For each frame, we sampled 10 sentences that contained the specific frame, for a total of 360 sentences. These sentences were divided in 10 batches, each one containing one sentence for each frame. We then assigned three evaluators to each batch. We assigned evaluators with an approach based on similar studies [153]: any group of three evaluators is evaluating at most one batch together and any group of two evaluators is evaluating at most two batches together.

Each sentence and the associated frame was presented to the evaluator through an online tool, an example of which can be seen in Figure 6.3.
Figure 6.3: Example of a sentence and the associated frame in our online tool. The first line, next to the green dot, contains the sentence being evaluated. The next line, next to the light blue dot, contains the frame proposed by SEFrame: from left to right, the name of the frame, the word that was used to identify the frame, and the definition of the frame; right below, the checkmark the evaluator could use to mark the frame as correct. The last line, next to the orange dot, contains the original frame proposed by SEMAFOR, in a similar fashion as the light blue line.

For each sentence, the evaluator was asked to indicate whether the associated frame correctly represented the intention behind the words used in the sentence or not. Whenever the evaluator indicated a frame to not be correct, a follow-up question displayed the original frame parsed by SEMAFOR for that sentence and asked whether the frame was more suitable. The evaluators had access to a common chatroom where some clarificatory questions were exchanged before the start of the study. This common chatroom meant every evaluator had access to the same information.

Results Table 6.4 presents a summary of the results with the percentage of frames that were evaluated as correct. We define a frame to be correct if at least 2 of the 3 evaluators agreed that it represents the meaning of its associated sentence. Overall, 73% of the 360 sentences were identified to be useful. The frame Execution, that we introduced in SEFrame, was identified as correct in 8 of the 10 sentences it appeared in. In the two sentences where it was considered incorrect, evaluators agreed that the original SEMAFOR frame was not a better candidate. We consider this result evidence that SEFrame is able to
Table 6.4: Results of correctness study

<table>
<thead>
<tr>
<th>Frame</th>
<th>Ratio</th>
<th>Frame</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicament</td>
<td>100%</td>
<td>Being obligated</td>
<td>70%</td>
</tr>
<tr>
<td>Required event</td>
<td>100%</td>
<td>Ordinal numbers</td>
<td>70%</td>
</tr>
<tr>
<td>Attempt</td>
<td>100%</td>
<td>Temporal collocation</td>
<td>70%</td>
</tr>
<tr>
<td>Identicality</td>
<td>90%</td>
<td>Measure duration</td>
<td>70%</td>
</tr>
<tr>
<td>Awareness</td>
<td>90%</td>
<td>Age</td>
<td>70%</td>
</tr>
<tr>
<td>Aggregate</td>
<td>90%</td>
<td>Quantity</td>
<td>70%</td>
</tr>
<tr>
<td>Likelihood</td>
<td>90%</td>
<td>Sole instance</td>
<td>70%</td>
</tr>
<tr>
<td>Existence</td>
<td>90%</td>
<td>Point of Dispute</td>
<td>70%</td>
</tr>
<tr>
<td>Desiring</td>
<td>90%</td>
<td>Relative time</td>
<td>60%</td>
</tr>
<tr>
<td>Instance</td>
<td>80%</td>
<td>Similarity</td>
<td>60%</td>
</tr>
<tr>
<td>Scrutiny</td>
<td>80%</td>
<td>Frequency</td>
<td>60%</td>
</tr>
<tr>
<td>Using</td>
<td>80%</td>
<td>Relational quantity</td>
<td>60%</td>
</tr>
<tr>
<td>Intentionally create</td>
<td>80%</td>
<td>Giving</td>
<td>50%</td>
</tr>
<tr>
<td>Capability</td>
<td>80%</td>
<td>Possession</td>
<td>50%</td>
</tr>
<tr>
<td>Grasp</td>
<td>80%</td>
<td>Time vector</td>
<td>50%</td>
</tr>
<tr>
<td>Inclusion</td>
<td>80%</td>
<td>Causation</td>
<td>50%</td>
</tr>
<tr>
<td>Desirability</td>
<td>80%</td>
<td>Cardinal numbers</td>
<td>50%</td>
</tr>
<tr>
<td>Execution</td>
<td>80%</td>
<td>Locative relation</td>
<td>30%</td>
</tr>
</tbody>
</table>

detect correct frames.

6.3.2 Is SEFrame Robust?

Having shown in the previous section that SEFrame provides correct results in the view of evaluators independent from the creators of the tool, we turn to the question of whether SEFrame can produce correct results over a broader range of kinds of artifacts. To explore if SEFrame generalizes across artifacts, we extend our dataset to cover an additional five types of software artifacts—pull requests, vulnerability reports, a broader set of questions and answers, mailing lists and app reviews—and evaluate the correctness of frames parsed by individuals unknown to the SEFrame creators.
Table 6.5: Results of robustness study

<table>
<thead>
<tr>
<th>Frame</th>
<th>Correct</th>
<th>Incorrect</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Likelihood</td>
<td>20</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Required Event</td>
<td>18</td>
<td>2</td>
<td>90%</td>
</tr>
<tr>
<td>Reasoning</td>
<td>17</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>Existence</td>
<td>17</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>Intentionally Act</td>
<td>17</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>Relative Time</td>
<td>17</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>Time Vector</td>
<td>17</td>
<td>3</td>
<td>85%</td>
</tr>
<tr>
<td>Events</td>
<td>16</td>
<td>4</td>
<td>80%</td>
</tr>
<tr>
<td>Sole Instance</td>
<td>16</td>
<td>4</td>
<td>80%</td>
</tr>
<tr>
<td>Capability</td>
<td>16</td>
<td>4</td>
<td>80%</td>
</tr>
<tr>
<td>Quantity</td>
<td>15</td>
<td>5</td>
<td>75%</td>
</tr>
<tr>
<td>Using</td>
<td>15</td>
<td>5</td>
<td>75%</td>
</tr>
<tr>
<td>Execution</td>
<td>15</td>
<td>5</td>
<td>75%</td>
</tr>
<tr>
<td>Inclusion</td>
<td>13</td>
<td>7</td>
<td>65%</td>
</tr>
<tr>
<td>Similarity</td>
<td>13</td>
<td>7</td>
<td>65%</td>
</tr>
<tr>
<td>Increment</td>
<td>12</td>
<td>8</td>
<td>60%</td>
</tr>
<tr>
<td>Aggregate</td>
<td>12</td>
<td>8</td>
<td>60%</td>
</tr>
<tr>
<td>Causation</td>
<td>11</td>
<td>9</td>
<td>55%</td>
</tr>
<tr>
<td>Temporal Collocation</td>
<td>10</td>
<td>10</td>
<td>50%</td>
</tr>
<tr>
<td>Sufficiency</td>
<td>9</td>
<td>11</td>
<td>45%</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>296</td>
<td>104</td>
<td>74%</td>
</tr>
</tbody>
</table>

**Method**  We extend the dataset with sentences gathered from five different data sources, which are associated with software design and programming:

1. **Pull requests** from the top 5 most starred projects on Github. From each project, we selected the top 10 most commented pull requests, and randomly sampled 50 comments for analysis. We filtered comments with less than 50 characters to avoid common comments found in pull requests, e.g., “Approved” or “Looks good to me”.

2. **Security threats** related to vulnerability management data that are described
using the Security Content Automation Protocol (SCAP) [154]. For this dataset, we randomly select the description field of threat reports from 2019.

3. A new set of questions from Stack Overflow due to the fact that the AnswerBot dataset used in our first study (Section 6.3.1) concerns only certain topics on the Java programming language [158]. For this new set of questions, we selected the 10 most commented questions from each one of the 5 programming languages mostly discussed on the platform.

4. Developers’ Mailing lists as represented by randomly selecting 6 archive threads from established projects of the Apache Foundation. From each thread, we parsed individual emails while ignoring any text in quote blocks such that we ignore duplicated text.

5. App reviews from the 10 most popular apps from the Google Play Store. From each app, we selected the top 50 reviews and randomly sampled 25 reviews for analysis.

Table 6.6 details the sources used. Overall, our selection criteria led to a total of 2530 sentences containing 302 distinct frames. We chose to focus on the top 20 most occurring SEFrame frames as, similar to our initial investigation in Section 6.2, these frames appear in 33% of the sentences in this new dataset. We therefore selected the top 20 frames and randomly sampled one sentence for each.

We then recruited 20 evaluators unknown to us through Amazon Mechanical Turk. These evaluators had to have obtained the Master Qualification after repeated participation in the platform. With this approach, we did not have direct control over the selection of evaluators. We introduced two additional sentences in those seen by an evaluator to validate the evaluator’s responses and to avoid exploitation of the platform with the goal of remuneration. The two validation sentences were hand-picked. The first one had an obviously correct frame associated with it (Figure 6.4a), while the other had an obviously wrong frame (Figure 6.4b). The correct one was placed as the first sentence in the study, while the incorrect one was placed at the end. Neither of these frames were considered in the
Table 6.6: Datasets for the robustness study

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>Size</th>
<th>Sentences</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull Requests</td>
<td>Pull requests from some of the most starred projects available on Github, i.e., Twitter’s Bootstrap, Google’s Flutter, Facebook’s React, FreeCodeCamp, and OhMyZsh</td>
<td>25 pull requests</td>
<td>648</td>
<td>328</td>
</tr>
<tr>
<td>Security Vulnerability</td>
<td>Security vulnerabilities available on the U.S. National Vulnerability Database</td>
<td>1866 reports</td>
<td>4553</td>
<td>579</td>
</tr>
<tr>
<td>Stack Overflow</td>
<td>Stack Overflow most upvoted questions for 5 of the most popular programming languages available, i.e., javascript, java, python, c#, and php.</td>
<td>50 SO questions</td>
<td>14196</td>
<td>633</td>
</tr>
<tr>
<td>Mailing lists</td>
<td>Mail archives from popular projects hosted by the Apache foundation, namely, Apache Commons, Couch DB, HTTPD, Maven, Lucene, and Spark.</td>
<td>6 mail archives</td>
<td>3650</td>
<td>561</td>
</tr>
<tr>
<td>Android App Reviews</td>
<td>User reviews for the ten most popular Android applications available on Google play store.</td>
<td>250 user reviews</td>
<td>1212</td>
<td>429</td>
</tr>
</tbody>
</table>

results. This validation process led to two evaluators being replaced, as they had indicated every frame, including the validation ones, as correct. With this validation step, we contend that the evaluators in this study are a reasonable representation of novice individuals doing program comprehension; novices may be more likely to benefit from tools built using technology like SEFrame.

Before signing up for the study, evaluators could see how much they would be remunerated and a short description of the study. Once they had signed up, they received a link to our study and instructions on how to obtain a unique code to verify they had completed the study.

When starting the study, evaluators were provided a more detailed description of the task, including instruction on how to complete the study. They were then presented a list of sentences with the associated frame, a definition of the frame and any frame elements the frame might have had. An example of this can be seen in Figure 6.4. Participants were
Figure 6.4: Validation frames. The frame in Figure 6.4a is a correct representation of the sentence, while the frame in Figure 6.4b is clearly wrong.

asked to indicate for each sentence if the frame correctly represents the sentence.

6.3.3 Results

Table 6.5 summarizes the results of our study. Overall, in 296 cases (74%) the participants indicated that the frame was correct for the sentence. In the majority of cases, 13 out of 20 frames, the majority of participants (at least 3/4 of them) indicated that the frame extracted was correct. In only one case, Sufficiency, the majority of participants disagreed with the proposed frame, but 9 out of 20 still marked it as correct. This frame was not in the list we evaluated in Section 6.2 to create SEFrame and we further discuss this threat in Section 6.4.

Consistent results from a different set of evaluators on different datasets increase our confidence that our approach generalizes to a broad range of text that appears in software engineering artifacts.

6.4 Threats

We performed three investigations as part of exploring the applicability of semantic frames to software engineering text. Although the details of each investigation vary, there are similar threats to validity across the studies.

Internal and Construct Validity

We developed SEFrame, which tailors semantic frames for software engineering based on data sampled from three datasets described in Table 6.1. As this dataset is not rep-
resentative of all kinds of text appearing in software engineering artifacts, it is possible that *SEFrame* may not adequately parse text from artifacts not considered by our investigation. To address this threat, we included an additional dataset when we investigated the validity of *SEFrame* (Section 6.3.1) and then used a broader set of artifact types and new data—Table 6.6—in our investigation of the generalizability of *SEFrame* (Section 6.3.2).

Another threat lies in the sentences sampled from the datasets, which may not be representative. To strike a balance between enough examples drawn from a diverse number of frames, we considered the distribution of frames over sampled sentences such that the focus was on the frames most likely to appear if *SEFrame* was applied. This led to the selection of the top 50 (Section 6.2) and top 20 (Section 6.3.2) most occurring frames in the datasets. To ensure we were not biased in the artifacts from which we selected sentences, we employed a sampling approach where we computed the frames for each sentence in every dataset, and randomly sampled out of these, regardless of which artifact they belong. This methodology allowed us to focus on the topmost occurring frames regardless of how often they appear in the data.

A key aim in our investigations is to determine the validity of frames parsed for a sentence. We relied on a variety of evaluators for the determination of validity. Across the investigation of validity and robustness, 30 individuals, none of whom are authors of this paper, assessed frames and sentences. While 1/3 of these evaluators were known to the authors (Section 6.3.2), 2/3 were not. The use of a broad range of individuals at arms-length from the authors decreases potential bias and increases our confidence in the results. A threat does arise from our choice of not requiring our evaluators recruited through the MTurk platform to have some certification of English fluency. Recent investigations in the demographic of the MTurk population has shown that the large majority of MTurkers are from the USA (75%) [38]. Additionally, our description of the study was provided in English, and we argue that it is unlikely that someone could have understood the task and passed the validation questions without a good understanding of the language.

There are also threats related to the design of *SEFrame* itself. In Section 6.3, a common pattern that causes a frame to be incorrect for software engineering text is when source-
code related nouns in a sentence evoke a frame. Instead of creating a list with source-code related terms that SEFrame could ignore, we opt to discard the frame entirely. This design decision errs on the side of caution because creating and ensuring that a list of source-code related terms is complete and always up-to-date is challenging. However, this means losing frames that are potentially correct in certain cases.

**External Validity**

In Section 6.3.2, we investigate the generalizability of our study [130]. To achieve this goal, we selected 5 new sources of text to use in place of the datasets used in the previous study. Based on our methodology and the diverse set of data we used, we believe our results are generalizable to different kinds of text in the domain.

### 6.5 Applicability to Design Points

As part of the study conducted to investigate the usefulness of design points in the context of tasks (Chapter 5), we asked participants to rank the design points that they had been shown by their usefulness for the task (Section 5.2.3). As reported in Chapter 5, we found five design points had been identified by most participants as the most useful (DP#1, DP#2, and DP#3 from Task A, and DP#1 and DP#2 from task B), and four had been identified as the least useful (DP#7 and DP#8 from Task A, and DP#3 and DP#7 from Task B). Table 6.7 shows these design points.

To investigate if semantic frames might help distinguish between these two groups of design points, we applied SEFrame to each design point. To give you an example, consider Figure 6.5, that shows a Causation frame appearing in DP#1 from Task A.

For each of the frames found in the nine design points analyzed, Table 6.8 shows how many times a frame was found. Overall, SEFrame found nine frames in the five design points that had been identified as useful; two of these frames appeared three times in these design points: the Awareness frame and the Using frame each appeared twice in DP#1 from Task A and once in DP#2 from Task A. Overall, SEFrame found seven frames in the four design points that had been identified as not useful, with the Capability frame
**Table 6.7:** The two groups of **useful** and **not useful** design points based on the results from the study presented in Chapter 5

<table>
<thead>
<tr>
<th>Useful Design Points</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task A</strong></td>
</tr>
<tr>
<td><strong>DP#1</strong></td>
</tr>
<tr>
<td>@seishun</td>
</tr>
<tr>
<td><strong>Task A</strong></td>
</tr>
<tr>
<td><strong>DP#2</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task A</strong></td>
</tr>
<tr>
<td><strong>DP#3</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task B</strong></td>
</tr>
<tr>
<td><strong>DP#1</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task B</strong></td>
</tr>
<tr>
<td><strong>DP#2</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design Points Not Useful</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task A</strong></td>
</tr>
<tr>
<td><strong>DP#7</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task A</strong></td>
</tr>
<tr>
<td><strong>DP#8</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task B</strong></td>
</tr>
<tr>
<td><strong>DP#3</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Task B</strong></td>
</tr>
<tr>
<td><strong>DP#7</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
This presumption that Linux uses utf8 just isn’t correct. Most applications default to interpret the byte sequence as utf8, but [Linux] \textit{fe:Cause} \textit{makes} \textit{Causation} [no presumption about how to interpret the input. The application that does that] \textit{fe:Effect}.

\textbf{Figure 6.5:} The \textit{Causation} frame identified in the design point #1 from Task A

Isn’t one of the biggest gripes about getting rid of existsSync() that it gives them a simple Boolean response? Throwing is also a breaking change on a heavily used API. You could argue that it’s a bug fix, but [I] \textit{fe:Cognizer} \textit{think} \textit{Awareness} [people will still yell loudly] \textit{fe:Content}.

\textbf{Figure 6.6:} The \textit{Awareness} frame identified in the design point #1 from Task B

appearing once in DP#8 from Task A and twice in DP#7 from Task B.

It is possible that the frames appearing only in the useful design points could be used to differentiate these from less useful design points. For example, consider the design point shown in Figure 6.5. The \textit{Causation} frame identified by \textit{SEFrame} appeared only in this design point that was identified as being useful. Its presence might be an indicator of usefulness.

On the other hand, consider the design point shown in Figure 6.6, that shows the presence of the \textit{Awareness} frame in another design point identified as useful. As Table 6.8 indicates, this frame appeared in three of the five useful design points, and in one of the 4 design points not useful. Even though the frame appears in both groups, its frequent presence in useful design points could be another indicator usable to distinguish them from less useful design points.

As \textit{SEFrame} was developed on a subset of frames, determining how it could used to differentiate design points will likely require additional experimentation to identify frames...
Table 6.8: *SEFrame* frames found in some of the design points (DP) used in the study presented in Chapter 5. The useful design points consists of DP#1, DP#2, and DP#3 from Task A, and DP#1 and DP#2 from task B; the design not useful consists of DP#7 and DP#8 from Task A, and DP#3 and DP#7 from Task B. The value displayed is in how many times the frames appeared in these design points.

<table>
<thead>
<tr>
<th>Frame</th>
<th>Useful DP</th>
<th>DP Not Useful</th>
<th>Frame</th>
<th>Useful DP</th>
<th>DP Not Useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>-</td>
<td>-</td>
<td>Intentionally create</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Aggregate</td>
<td>-</td>
<td>-</td>
<td>Likelihood</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Attempt</td>
<td>-</td>
<td>-</td>
<td>Locative relation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Awareness</td>
<td>3</td>
<td>1</td>
<td>Measure duration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Being obligated</td>
<td>-</td>
<td>-</td>
<td>Ordinal numbers</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Capability</td>
<td>1</td>
<td>3</td>
<td>Point of Dispute</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cardinal numbers</td>
<td>-</td>
<td>-</td>
<td>Possession</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Causation</td>
<td>1</td>
<td>-</td>
<td>Predicament</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Desirability</td>
<td>-</td>
<td>-</td>
<td>Quantity</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Desiring</td>
<td>-</td>
<td>-</td>
<td>Relational quantity</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Execution</td>
<td>-</td>
<td>-</td>
<td>Relative time</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Existence</td>
<td>1</td>
<td>1</td>
<td>Required event</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Frequency</td>
<td>-</td>
<td>1</td>
<td>Scrutiny</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Giving</td>
<td>1</td>
<td>1</td>
<td>Similarity</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Grasp</td>
<td>-</td>
<td>1</td>
<td>Sole instance</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Identicality</td>
<td>-</td>
<td>-</td>
<td>Temporal collocation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inclusion</td>
<td>-</td>
<td>-</td>
<td>Time vector</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>1</td>
<td>-</td>
<td>Using</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

most associated with design decisions. We leave this determination to future work.

### 6.6 Summary

In this chapter, we have investigated whether frame semantics, a general linguistic approach aiming to represent the intended meaning of the words in a sentence, might help identify which design point is useful.

We developed a tool, called *SEFrame*, that tailors an every-day English semantic frame parser (SEMAFOR) for software engineering text artifacts. We evaluated *SEFrame* in two
ways: first to evaluate its correctness (Section 6.3.1) and then to evaluate its robustness (Section 6.3.2). In both studies, SEFrame performed well, extracting the correct frame in 73% of the cases in the first study and 74% in the second study. Based on these results, we argue that SEFrame can reliably extract the correct frame for sentences across a broad range of software engineering text appearing in software engineering text artifacts.

We also report on initial results of exploring whether applying SEFrame to design points can be used to identify which of these design points would be useful to developers. We further discuss possible future work in this direction in Chapter 7.
Chapter 7

Discussion and Future Work

We began this dissertation describing our investigation to understand the perspectives software engineering researchers and practitioners have about software design (Chapter 3). We then introduced an automatic approach to locate paragraphs with design information—design points—in developer discussions (Chapter 4) and investigated whether developers could make use of latent design information recovered in the form of design points (Chapter 5). We also considered if semantic frames might hold promise to disambiguate useful from less useful design points (Chapter 6). There are many open questions remaining about how design points might be identified, prioritized and used to aid developers. In this chapter, we consider opportunities for:

- refining the location of design points (Section 7.1);
- extracting better representations of design from design points (Section 7.2);
- other sources for latent design information (Section 7.3); and
- applying more modern natural language processing (Section 7.4).
7.1 Refining the Location of Latent Design

In Chapter 4, we introduced a classifier based on a supervised learning approach to identify latent design points in developer discussions. This classifier identifies design points as paragraphs in developer discussions. An advantage of identifying a paragraph is that a paragraph can provide context and rationale for understanding the nugget of design being considered as part of a decision. Consider the design point in Figure 7.1 from pull request #1077 of the Node.js project. The design nugget in this paragraph is the information about the need to consider trade-offs between performance, a comprehensible implementation and the API used to access the software being built. Information included in the design point that explains this is a ‘leaky abstraction’ helps to frame the precise design considerations: A ‘leaky abstraction’ is vocabulary that comes from other work in design in the community [133].

An alternate approach to identifying latent design could target a more precise definition of design within a paragraph, such as just the sentence or phrases in Figure 7.1 that contain precise design information about a system. We decided early in our investigation to focus on the granularity of a paragraph, as considering each sentence separately could cause fragmentation of an author’s intent. Other researchers have shown that sentences can also be used to mine design information [11, 72] and the exploration of sentences for design points would be an interesting future work direction.

With this dissertation, our focus has been on demonstrating the possibility and value of identifying latent design. Future directions, such as considering finer grained text than paragraphs, may help refine the precision and potential usefulness of latent design infor-
7.2 Extracting Design from Design Points

As presented in this thesis, design points place the onus on the developer to extract specific design information from a design point. Design points may be more useful to developers if additional processing was undertaken to automatically extract and represent specific design information. We consider three opportunities for extracting more specific design information from design points: 1) topic identification among design points (Section 7.2.1), 2) rhetorical structure (Section 7.2.2), and 3) frame semantics (Section 7.2.3). We discuss each in turn.

7.2.1 Topic Identification

As we argue in Chapter 5, design points might have particular value for newcomers to an open source project. A challenge with presenting design points to newcomers is that they likely lack the ability to distinguish and interpret contextual references to projects that might be embedded in design points: for example, multiple participants in the study we ran reported that they couldn’t grasp the context of some of the design points they were shown, even with the additional paragraphs provided.

It may be more helpful to newcomers if, instead of presenting design points as paragraphs from developer discussions, the design topics that should be considered in suggesting changes to the system are extracted from design points and presented to newcomers. For example, instead of showing a newcomer the design point in Figure 7.1, a tool could extract the design topics of performance and APIs from the design point, and show these topics to the newcomer to initiate consideration of these aspects to the newcomer.

To investigate whether topics might be automatically identified from design points, we randomly sampled 50 paragraphs from the 2,378 paragraphs identified as containing a design point in the dataset described in Chapter 5. We then applied open coding [138] to determine the design topics being discussed.
Two annotators coded the paragraphs separately, followed by a meeting where we discussed the differences in our coding. We then randomly sampled 50 additional paragraphs and coded them. We repeated this process three times, for a total of 150 paragraphs. Between each iteration, we discussed the difference in topics, until a common set of topics was determined and no new design topic appeared. Finally we sampled a last set of 75 paragraphs, which they then coded and for which we measured the inter-rater agreement using Cohen’s Kappa Coefficient. The inter-rater agreement was 0.64, a value considered to be “substantial agreement” [43, 69].

Table [7.1] summarizes topics that were considered during the process. The left side of the table (after the double line) list the main topics that we considered over the entire set of 275 paragraphs. The number of times each one of us topic is indicated. For brevity, we do not include topics that appeared less than five times. Although there are similar numbers for some topics, many topics have a fairly large difference. These differences can be explained by the fact that Coder #2 tended to differentiate more between design topics, thus spreading the observations to a larger set of topics. In other words, Coder #2 tended to have more topics that were used only once or twice, and therefore not shown in the Table.

The right side of Table [7.1] shows the final set of design topics found in the dataset, each with a short description of its meaning. Specifically, we list the results of the coding of the last 75 paragraphs, which took place we had finalized the common set of topics with significant inter-rater agreement. We show only the number of occurrences for which we both agreed. Therefore, the total number of occurrences is 53, rather than 75. Given the size of the sample, some of the topics are present only in small numbers, we observe that in the larger set, on the left side, those topics are represented in a more significant number of cases. We thus consider them as relevant. Some of the topics of the left side had been grouped in single categories, as shown in the Table: for example, Robustness also includes Safety.

The determination of design topics presented multiple challenges. First of all, developers rarely explicitly state the topic of a paragraph. Thus, we often had to rely on our own
Table 7.1: Results of the coding process. Left side of the table contains information on all the 275 paragraphs coded, the right side includes only the 75 paragraphs with the finalized codebook

<table>
<thead>
<tr>
<th>Category</th>
<th>Occurrences Coder #1</th>
<th>Occurrences Coder #2</th>
<th>Category</th>
<th>Occurrences Agreed on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>105</td>
<td>105</td>
<td>code</td>
<td>11</td>
<td>Implementation issues</td>
</tr>
<tr>
<td>maintainability</td>
<td>58 (8)</td>
<td>38 (0)</td>
<td>maintainability</td>
<td>14</td>
<td>Future plans, OS support, code standards...</td>
</tr>
<tr>
<td>planning</td>
<td>10 (0)</td>
<td>0 (4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plan</td>
<td>6 (4)</td>
<td>1 (0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dependencies</td>
<td>7 (0)</td>
<td>0 (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compatibility</td>
<td>7 (0)</td>
<td>0 (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>usability</td>
<td>0 (3)</td>
<td>0 (3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testing</td>
<td>7 (3)</td>
<td>3 (1)</td>
<td>testing</td>
<td>1</td>
<td>Tests and testability</td>
</tr>
<tr>
<td>robustness</td>
<td>32 (0)</td>
<td>36 (6)</td>
<td>robustness</td>
<td>13</td>
<td>Robustness, safety, security</td>
</tr>
<tr>
<td>safety</td>
<td>0 (6)</td>
<td>0 (6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>performance</td>
<td>11 (11)</td>
<td>11 (2)</td>
<td>performance</td>
<td>2</td>
<td>Performance, runtime optimization</td>
</tr>
<tr>
<td>configuration</td>
<td>20 (14)</td>
<td>14 (4)</td>
<td>configuration</td>
<td>4</td>
<td>Configuration files, flags and options</td>
</tr>
<tr>
<td>documentation</td>
<td>23 (26)</td>
<td>26 (1)</td>
<td>documentation</td>
<td>1</td>
<td>Documentation in-code and off-code</td>
</tr>
<tr>
<td>clarification</td>
<td>22 (29)</td>
<td>29 (7)</td>
<td>clarification</td>
<td>7</td>
<td>Assorted requests to clarify statements</td>
</tr>
</tbody>
</table>

intuition to understand what topic the paragraph was about. The decision of topic was made harder by the fact that paragraphs may not always make sense without the context of paragraphs around them.

The coding process also required a large amount of time. Our investigation was made possible thanks to the reduced sample size, but the amount of work that would be required to obtain enough data to use with an automated tool would make a manual categorization
To overcome some of these challenges, design-related keywords might be automatically extracted from paragraphs in discussions, based on techniques such as TextRank [91]. Those keywords can become associated with manual annotations about design topics, and a supervised learning approach can then be used to automatically determine the topic of a paragraph with a design point.

If the topic of design could be identified automatically, this information could be used to build tools to help developers. Consider, for instance, the fact that discussions for pull requests can become lengthy. For example, pull request #4765 of the Node.js project has a total of 223 comments. In such discussions, it is not uncommon for there to be many back-and-forth comments between developers about a design topic, such as performance. Resolving the design issue in these cases can require the solicitation of another project member who has more expertise on the topic to weigh in. If the design topic can be determined automatically as suggested above, a recommender tool could be provided to automatically introduce the expert project member. Such a recommender would go beyond existing work that recommends reviewers only at the beginning of a pull request based on the source code modifications [65, 144, 145, 161]. The recommender we propose would go beyond this existing work to include information found in discussions about a code review; information that we believe has been largely untapped.

Our initial investigation on what design topics can be recognized appeared in [152].

7.2.2 Discussion Structure

At present, we treat all design points as independent from each other. However, within a pull request, the design points exist as part of a discussion between developers. If we could automatically understand the form and content of the discussion, we believe we could better support a developer in using design point information. The research described below appeared in [151].
For example, consider a developer joining the Rails project who wonders why the MailPreviews component manipulates MIME parts in a particular way. The developer might query system artifacts with this question. A tool that understands written design discussions present design points in the discussion over Rails pull request #14519 and extract for the developer the information that Rails developers have decided that: “it’d probably be clearer (and simpler) to return new MIME parts rather than alter them in-place”.

To explore whether there are future possibilities to exploit the structure of discussions, we report on an in-depth analysis of the structure of discussions from three failed pull request from the Rails open source project. Failed pull requests represent suggested changes to a project that ultimately were not merged into the system. We study these failed pull requests because design-related concerns are some of the most common factors that impact the choice of whether to integrate a pull request [52].

We considered the following two research questions:

1. How does design information occur in written discussions?

2. How are elements of design information related in a written discussion?

We consider these two questions in the context of pull request discussions prior to identifying design points so as to not bias the consideration of discussion structure. We focus on a small number of pull requests from one project to enable a rich understanding of the context of the discussions. The project we focus on is Rails, a framework for building web applications. We select three failed pull requests at random from a set of 23 Rails discussions extracted with GHTorrent [51]. The three pull requests analyzed were: #14519, #11407, and #3871.

We opted for a coding process that builds on earlier work (e.g., [87]) determining and applying labels to sentences, phrases, and implicit suggestions in any comment of a discussion extracted from a pull request. We refer to these labels as “elements”. When coding

---

1]https://github.com/rails/rails

2]Rails #14519 line 245
a discussion, we identified the elements introduced in each comment, and we determined how different elements across the discussion relate to each other.

**GitHub Pull Request Structure**

On GitHub, a pull request discussion consists of multiple time-stamped comments. Each comment is made by an author, belonging to three possible categories: the original poster is the creator of the pull request and author of the first comment in the discussion; we consider authors who are part of the core development as project members; otherwise, we consider participants as other.

Each comment consists of a number of paragraphs, interspersed with occasional code snippets. The discussions also contain occasional events such as code commits, cross-references to other pull requests or issues, links to code reviews, etc. As our focus is on developer discussions, and not on the code being modified, we opted to ignore code snippets in our coding process.

**Labelling**

We read and applied abstract labels to paragraphs of the discussion for #11407. We started with a small initial set of labels, inspired from previous work [87] and independently annotated the discussion. After consolidating the annotations, we expanded the set of labels to capture additional types of “moves”: a move is a part of a comment introducing a new element, a concept inspired from RST and Black et al. [18]. We then used the new label set to annotate the other two pull requests.

Our final set of labels was: Question (Q): the comment author poses a design decision that needs to be addressed; Candidate (C): the author proposes a way to resolve a Q; Support (S)/Reject (R): the author supports/rejects a C; and Justification J and Rationale L: the author justifies the prefixed Support/Reject with a rationale.

Figure 7.2 contains a snippet of an annotated discussion. Labels are placed below the paragraph to which they refer and are prefixed with the string “;##”. The developers are discussing whether inline style elements can be a security threat. In line 8, @Alamoz intro-
Someone will have to decide whether to add a default stylesheet to rails apps. Also, there is discussion to the effect that inline style elements are not a real security threat (see http://lists.w3.org/Archives/Public/public-webappsec/2013Feb/0015.html) and that JQuery relies on inline style elements for things like tooltips. So this warrants further investigation.

;##Q4 Are inline style elements a real security threat?

As for the security issues of inline styles, there are various dangerous things you can do with expressions and moz-binding etc, which is why the CSP disallows them. However there’s no issue with the values we’re currently generating,

;##C1::Q4 No.

‘‘Inline style elements certainly give the attacker leverage. The question isn’t whether it’s a vulnerability but rather how severe a vulnerability.’’

;##C2::D4 Gives an attacker leverage.

@Alamoz you’re missing the point a bit here, there’s no question about whether the code we’re generating is insecure, it’s not. There’s no vulnerability at all ever from setting padding and margin on a div.

;##L12 Padding and margin on a div are not vulnerable

;##R C2::Q4 J: L12

Figure 7.2: Discussion snippet from Rails pull request #11407
If email is of type ‘multipart/relative’, then we render the text/html part contained in it.

;##Q1 Should multipart/relative emails render the text/html contained in them?
;##C1::Q1 > Q2
;##S C1::Q1 J: L1

We’d like to render the image using a URL [...]  
;##Q2 How to render the image?
;##C1::Q2 using a URL

@jeremy Ruby on Rails member added a note May 1, 2014

Seems we’d need to decode the body before munging it.  
;##Q3 Does the body need to be decoded?
;##C1::Q3 Yes.

@tute added a note May 2, 2014

Calling decode rather than raw_source here didn’t work. It didn’t render the images properly, can investigate further if needed.  
;##L6 calling decode doesn’t render the images properly.  
;##R C1::Q3 J: L6

Figure 7.3: Discussion snippet from Rails pull request #14519

induces the discussion by pointing out an ongoing debate whether inline style elements are a threat or not. Hence, the paragraph is annotated with Q4, to indicate that this developer wishes to discuss an open point about the design. We assigned unique numbers to each label to allow cross-referencing. Line 19 has the label “C1 :: Q4”, indicating the first candidate solution C1 to question Q4. We use the symbol “::” to link Cs with their respective Qs. Similarly, line 28 has the label “R C2 :: Q4 J: L12”, indicating the candidate solution C2 for the question Q4. In this move, the candidate “C2 :: Q4” is being rejected as a solution for Q4, as indicated by the prefix R. The suffix “J: L12” provides further details, indicating that this rejection is justified (J) by the rationale L12.

Figure 7.3 contains a more complex example. In this case, the developers are discussing a bug fix to an email rendering subsystem. In line 16, @jeremy speculates whether it would help to decode the message body before processing it, and the speculative question
is annotated with Q3. However, his phrasing implies that he is also proposing a candidate solution to his own question. That is to say, he thinks that they should in fact do the decoding. In this case, line 16 is also labelled “C1 :: Q3” to demonstrate that the same paragraph both poses the question and offers a candidate solution. Later, this candidate is the rejected by @tute at line 25, which is indicated with the label “RC 1 :: Q3 J : L6”. Again, the suffix “J : L6” shows that this rejection is argued for and justified by the rationale L6.

The snippet also shows labels from the expanded set. Line 5 has the annotation Q1, indicating a question. This is followed by line 7, where we have used the label “C1 :: Q1  >  Q2” to indicate that question Q2 only makes sense if Q1 is resolved by selecting the candidate solution C1. Deciding whether it is a good idea to render images as a URL (Q2) is only a meaningful question if the team has already decided that multipart emails should be rendered as text (C1 :: Q1). In other words Q2 elaborates C1 :: Q1. We indicate the elaboration relationship with the symbol “>”.

Results

The coding process provides insight into the two research questions. With respect to our first question (how design information appears in discussions) we gained two major insights. First, multiple design concerns may be brought up in the same comment (e.g., @tute’s comment between lines 1-14 in Figure 7.3). This suggests that the use of comments as has been considered in earlier work (e.g., [66] and [122]) is too coarse, and a finer level, paragraphs for example, may be a more appropriate granularity for localizing design.

Second, contrary to the intuitions found in previous work on modelling design discussions [18, 87], design concerns do not always appear in the form of design questions (Q) that are subsequently answered by respective candidate solutions (C). It is not always straightforward to unambiguously classify an element as either Q or C, because discussion participants may propose ideas that do not answer some previously posed question.

Specifically, we observed two cases: often, someone first proposes a solution, which can then be understood as posing a new question about design (i.e., whether their particular
solution be accepted). For example, in line 16 of Figure 7.3, @ jeremy is asserting that the email bodies should be decoded, a position rejected by @ tute in line 25. The implication is that in line 16, @ jeremy simultaneously poses a Q (albeit in the form of an assertion) and proposes a C to resolve it, which is later rejected by @ tute. In other cases, an element could be a candidate solution to a previous question, while continuing the discussion with a further question at the same time. While our approach would annotate them as both Qs and Cs, these cases seems to indicate that our initial division of elements between Qs and Cs may have been too naive.

With respect to our second question (the relationship between design elements) we observe that developer comments are almost always related to each other, effectively defining a structure for the discussion. These relationships can be logical and rhetorical. On the one hand, the logical structure of a discussion captures the relation between Qs and Cs, as well as the interdependencies between design concerns (either Qs or Cs), such as Q2 being dependent on C1 in line 10 of Figure 7.3. On the other hand, the rhetorical structure of the discussion captures the argumentation that takes place between developers, including support/reject elements and expositions of rationale. Dependencies between rhetorical elements can be complex, such as conditionally supporting a C, or critiquing someone else’s rationale for supporting or rejecting a position.

Applicability to Design Points

We found that some parts of the structure of a discussion are relatively clear. For example, we found general patterns of co-referential questions and candidates. However, there is also a much less ordered side to design discussions. We found moves to support and reject sometimes appear to reference multiple candidates, often lack justification, and can be counter-rejections to another reject.

If we were able to develop a model of this less ordered structure to a design discussion and combine it with labelling of which paragraphs indicate design points, we might be able to:

- determine which design points are relevant from a discussion, such as not retrieving or displaying design points from rejected parts of the discussion; and

113
• present design points within an argument structure that makes their contents easier to consume for the developer.

Further investigation of argument structure in developer discussions and a means to automatically identify the structure could help in the presentation of latent design based on design points to developers.

7.2.3 Frame Semantics

In Chapter 6, we discuss frame semantics, a technique that aims to represent the meaning behind the words in a sentence. This information could be valuable to identify which design points might be useful to a developer working on a task. A similar use of frame semantics has been considered by Marques et al. [85] who analyzed which semantic frame appeared in text highlights that had been deemed important to a task by developers.

As software engineering vocabulary can differ from a more general vocabulary, it is important to verify that semantic frames are applicable to software engineering. To achieve this goal, we have introduced SEFrame, a frame semantic tool tailored for software engineering text. Although we have described a brief exploration of the application of SEFrame to design points, a larger study is needed to understand which semantic frame can be used to better identify relevant design points. Another direction of future study is to investigate whether semantic frames can be used to represent the design point information, rather than using the design point itself.

Finally, a current limitation of SEFrame is that it was developed on a subset of semantic frames. Additional work is necessary to expand this set and investigate the remaining semantic frames that have not been verified in this domain. We leave these open questions to future work.

7.3 Design Points in Other Discussion Forms

The work presented in this thesis, in particular regarding the location of design points (Chapter 4) and their application (Chapter 5), focused on discussions that result from the
opening of pull requests. The decision to focus on these discussions was made as pull requests have been shown to often contain design information [24] [147].

At the same time, there are many other channels through which developers communicate with each other. Recently Gousios et al. surveyed 645 developers about their preferred channel of communication to propagate changes [53]: the top two responses were Issue Trackers and Pull requests, with Emails claiming a third spot. Less popular responses included IRC, various synchronous channels (such as Skype, Google Hangouts, or other IM platforms), and Twitter. Panichella et al. [102] investigated the commonalities and differences between three communication channels in seven projects: similarly to Gousios et al., they also found that developers have moved from emails to chat and issue trackers. More importantly, they found that for the majority of projects they investigated only two out of three channels were being used regularly, while the third was being used only sporadically. Which channels were being used depended on the project, suggesting that multiple channels of communications need to be investigated if one aims to to obtain a complete picture.

Discussions happening on issue trackers or bug reports are likely to be fairly similar in structure, as both share many similarities in structure to pull requests on platforms such as GitHub. For this reason we expect that our approach could be applied to these sources without needing changes.

Emails and instant messaging (IM) channels provide additional challenges. While language utilized by developers is unlikely to change from the one we have observed, applying our approach to these forms of discussions poses additional challenges. Emails have been shown to not have clearly defined discussion boundaries and to have a smaller presence of core members in the discussions [55].

IM channels, such as IRC, have been compared to the “watercooler” conversations [48]. As such, we can expect even less structure in these forms of discussions, further exacerbating the problem of defining the boundaries of a discussion.

Additionally, both emails and IM channels have been shown to suffer from the multiple
alias problems [129]. As our classifier makes use of information about the author of the message, including it’s role, and of the position of the paragraph in the discussion, additional work is required to understand how our approach can be generalized to these forms of discussions.

### 7.4 Natural Language Processing

Natural language processing is a fast evolving field. This thesis considers how well-established older natural language processing approaches, such as semantic frames, might help in the processing of the text in design points. There are a number of other natural language processing techniques that could be used to extract more refined design information from design points and from latent design in natural language software artifacts. One example could be the use of sentiment analysis [25, 41], a set of techniques that aim at classifying text, voice, or video based on the affective/emotional state. Basic techniques of sentiment analysis aim to differentiate only between positive, negative, or neutral text [92], but more sophisticated approaches have been developed that can differentiate between more complex emotions [137]. Sentiment analysis has been applied to software engineering texts [101], albeit researchers have warned that existing tools, even when re-trained on software engineering texts, often do not perform well [79].

An exploration of sentiment analysis could investigate whether the tone in which a design point is written might indicate whether or not the information in the design point might be useful. As another example, exploring pre-trained models, such as BERT [36], might help do a better job than semantic frames at extracting meaning from design points.

The exploration of other sources of developer discussions and the assessment of applying these modern NLP techniques to locate design points is left to future work.
Chapter 8

Conclusion

Every software project involves design. Although an understanding of design can help ensure changes to a system are made appropriately [104], design information about a project is not always recorded, and when design is recorded, it is typically not kept up to date [96].

In this thesis, we investigate whether latent design information embedded in artifacts associated with the project might be used to surface design information to a developer. The following statement describes the intent of the thesis:

*Design information about a software system embedded in written discussions between developers can be identified automatically and made available to developers in a form that allows them to make better design decisions as the system evolves.*

We began this dissertation by describing an investigation of developers and researchers perspectives on software design. As we found no evidence in the literature of a common definition of software design, we interviewed twelve academics and four practitioners, asking them how the view design (Chapter 3). We found five major recurring themes in their responses. One theme focused on where design might be found in a project, namely when decisions are made.
We focused on decisions as an anchor for identifying and recovering design information from recorded discussions between developers already occurring as part of their normal work process. We coined the novel concept of a design point. We define a design point as a piece of a discussion relating to a decision about a software system’s design that a software development team needs to make. We annotated a large dataset of 10,790 paragraphs extracted from pull requests, identifying 2,378 paragraphs as design point. We then used this dataset to create a classifier that is able to automatically localize design points in pull requests discussions (Chapter 4). This confirmed that design information can be automatically identified within written discussions between developers.

To verify if the localized information can be used to help developers make better design decisions, we conducted a user study simulating the existence of a tool that provides design points to newcomers when they are working on a development task (Chapter 5). The results from this study provided evidence that design points are sometimes both understandable and useful to developers.

The user study also identified that newcomer developers regarded some design points as useful and some as not useful for a task. To consider whether delving into the meaning of a design point might support the differentiation of useful and not useful design points, we investigated the application of semantic frames, an existing natural language processing approach, to design points. We first considered whether semantic frames are applicable to the kind of natural language text appearing in software engineering text, developing a tailored semantic frame tool, SEFrame, for this purpose. We then applied SEFrame to design points from the user study, finding some evidence that semantic frames might be useful in distinguishing useful design points (6).

Finally, we explored (Chapter 7) some remaining open questions in regards to the applicability of design points, focusing in particular on how to improve the localization of design points, how to better represent them, and how generalizable our approach is to other forms of communication.

Design is, and will remain, a core component of software development. In a similar fashion, development will remain a collaborative activity and developers will continue to share
information with one another. With this work we hope to have built the foundations for an approach to recovering latent design that others can continue to generalize and can continue to apply to different kinds of recorded developer discussions in a variety of evolving communication channels.
Bibliography


124


[84] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics, 18(1):50–60, 1947. → page[54]


techniques to extract rationale from existing documentation. In Design Computing

[125] D. Romano and M. Pinzger. Using source code metrics to predict change-prone
java interfaces. In Software Maintenance (ICSM), 2011 27th IEEE International
Conference on, pages 303–312. IEEE, 2011. → page[51]


doi:10.1109/32.799955 → page[16]

[128] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering
Architectural Design Decisions. In 2018 IEEE International Conference on

[129] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the use of developer IRC
meetings in open source projects. In 2009 IEEE International Conference on


[131] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study
page[2]


Appendix A

Pull Requests Annotation Codebook

The following pages contain the codebook that was used in the annotation process described in Chapter 4.
Design Points Protocol and Codebook

Scope (cf. Glossary):
A conversation is within scope if it has more than 20 comments and the pull request is CLOSED. Conversations of wishlist items and other tickets are out of scope.

Overview:
1. Identify the paragraphs of the conversation that contain design points (DPs)
2. Annotate the DPs with respect to the following dimensions:
   - role: who posits the DP
   - invited: whether the poster was invited in the conversation
   - form: how is the DP posited
   - relationship: how does the DP fit in the conversation

Procedure:
Start from the top, localizing and annotating DPs as you are reading the conversation (do not skim ahead in the conversation).

If a paragraph contains a DP, add these annotations directly above it:

;##D<Identifier> <Summary>
;##ROLE <Role>
;##INV  <Invited>
;##FORM <Form>
;##REL  <Relationship>

Always start annotations with the prefix ";##".
Each DP should be given a unique number <Identifier> and a short <Summary>

Each DP is expected to have exactly one tag per dimension in accordance with the Codebook.

If none of the tags fit a particular DP, expand the codebook and rationalize your decision.

If a paragraph contains multiple DPs, append them sequentially so that all DP annotations are directly above the paragraph.

When code is interspersed in the conversation, ignore it.

You might find it useful to keep a separate file open where you copy paste the first line of each annotation (";##D<id> <summary>") for easy access when you are writing ;##REL annotations.
It might also help to keep track of people being invited and when commits started happening.

Codebook:
The following codes apply only to paragraphs that contain at least one DP.
Role:
Use the dimension ;##ROLE to characterize the role of the author of the paragraph containing the DP using one of these tags:
OP -- "original poster":
   the author is the same person who submitted the pull request
PM -- project member:
   the author is a core member of the GitHub project
ETC -- other:
   the author does not fall in any of the categories

Invited:
Use the dimension ;##INV to indicate whether the author of the paragraph containing the DP has been explicitly invited into the conversation by some other participant. Use these boolean tags:
T -- True:
   the author was specifically invited
F -- False:
   the author was not specifically invited

Hint:
It will be useful to keep an eye out for the occurrences of invitation events (e.g. "@user, care to comment?") as you are reading the conversation.

Form:
Use the dimension ;##FORM to characterize how the DP is posited within the discourse of the conversation, using these tags:
OPQ -- open question:
   the DP is posited as an open ended question, without assuming a list of possible solutions
ENU -- enumeration (closed question):
   the DP is posited as a choice between a set of possible solutions
SOL -- solution:
   the DP is posited as an "obvious" choice, i.e., in the form of a statement as opposed to a question

Disambiguation:
Detecting SOL-type DPs does not guarantee the absence of false negatives. The hardest ones to find are statements that are without rationale and/or speculative language that address an entirely new design aspect. In order to separate arbitrary statements from those that are also DPs, follow these guidelines:
- does the statement have speculative language? e.g. "Should probably", "could", "recommend"
- does the statement come with a rationale
- does the statement elaborate/generalize an existing DP
If you identify other ways of separating arbitrary statements from DPs that are SOL-type statements, add to this list and rationalize

Relationship:
Use the dimension ;##REL to characterize how the DP fits in the conversation with respect to other DPs that have already been identified.

Use these tags:

NEW -- new:
the DP is about a previously unexamined aspect of the design discussion
ELAB D_ -- elaborates:
the DP decomposes and adds detail to an already existing DP. Make sure to point to this already existing DP. Example usage:
;##REL ELAB D10
GEN D_ -- generalizes:
the DP abstracts from an already existing DP, i.e., an existing DP can be said to elaborate this new DP. Make sure to point to the existing DP. Example usage:
;##REL GEN D10
REFR D_ -- reframes:
the DP casts an existing DP in a different light, exposing aspects that were previously not considered. Make sure to point to the existing DP. Example usage:
;##REL REFR D10

If none of the tags fit a particular DP, expand the codebook and rationalize your decision.

Glossary:
Design Point:
A decision relevant to design that the development team needs to make. A DP exists independently from whether a decision was indeed made. In other words, any decision taken by the team is a particular way of resolving a DP.

Note:
In earlier iterations, we used to distinguish between “design question” (e.g. “which cafe should we go to?”) and “candidate solutions” (e.g. “Starbucks”). We no longer make this distinction, and instead consider both to be DPs.
Rationale 1: it is hard to distinguish because a “candidate solution” can present as a question (“should we go to Blenz?”) and it can also end up being elaborated to become a question (“the Blenz on campus”).
Rationale 2: ultimately it doesn’t matter: they are all “points” that need to be decided.

Project:
A GitHub project
Pull request:
A patch submitted to a project; it has a conversation, an associated piece of code, and a revision history

Conversation:
A discussion over a pull request for a project

Original Poster (OP):
The person submitting the pull request

Participant:
The author of a comment

Comment:
An individual posting to the conversation

Paragraph:
A comment contains multiple paragraphs, identified because they are separated by two consecutive newline characters

Sentence:
Any piece of text contains multiple sentences, identified as complete sequences of words. Often, they start with a capital letter and finish with a period. But this is not always the case.
Appendix B

Questionnaire on the Usefulness of Design Points

The following pages contain the complete questionnaire that was used in the study described in Chapter 5. Only the version seen by Group I is provided.
Investigating the usefulness of design points in a developer's decision making process

Background Information

With which gender do you most identify

- Woman
- Man
- Transgender woman
- Transgender man
- Non-binary
- Two-spirit
- Not listed: [ ]
- Prefer not to answer

If you are a student, at which year in the course program are you at?

- 1st year
- 2nd year
- 3rd year
- 4th year
- 5th+ year
- Graduate student
- Not a student

How many years have you been programming?


Have you ever worked as a professional programmer/developer? If yes, for how many years?
Study Structure

Thank you for your interest in participating in this study.

As you are likely aware, while working on a software development project, software developers often need to make a decision on which approach to follow to solve a specific task.

In this study, we will ask you to answer a few questions regarding two coding tasks. You will not be required to write any code, but rather reflect on the approach to take to solve the tasks.

Both tasks are based on the Node.js project. Node.js is a back-end JavaScript runtime environment, allowing to run JavaScript code outside of web browsers. While reading these tasks, it is important to remember that Node.js serves two distinct groups of users: developers who make use of it to build their own projects, and final users who use these projects.

For each task, we ask you to imagine that you are a software developer who has just joined a new project and has found a task to work on. You will be presented with a description of the task and three potential solutions that could solve the task.
Investigating the usefulness of design points in a developer’s decision making process

Task One

Task Description

You are a developer who has recently started contributing for the Node.js project. You have found multiple issues opened by third party developers about Node.js throwing errors when attempting to open local files. You started investigating the reports and were able to replicate it correctly: based on your findings, Node.js is unable to correctly open files whose name contains foreign characters. As you know, Node.js uses a low level C library to handle I/O operations, and after a quick check you realized that it supports only UTF-8 as encoding.

Your goal for this task is to find a solution allowing Node.js to correctly handle cases in which files provided by a user are not encoded in UTF-8. Note that, in this context, “user” refers to the third party developer who’s code is providing the files, and not to the final users.

After some time reflecting on the task, you have come up with three possible solutions.

- **Solution 1:** Continue assuming that UTF-8 is the encoding being used and add error handling in the form of providing a message back to the user. This would guarantee that NodeJS continues working as expected, and notifies the user that there’s a problem with their file encoding. The downside is that the user may not know how to handle this problem.

- **Solution 2:** Continue assuming that UTF-8 is the encoding being used, but force convert any filename provided to UTF-8. Node.js will continue operating as normal and will be able to handle files in other encodings too. Even though you can be certain that filenames will be converted to UTF-8 correctly, you are not certain if any other side effect may happen.

- **Solution 3:** Continue assuming that UTF-8 is the encoding being used, but offer an optional parameter allowing the user to specify which encoding the files are. Node.js will continue operating as expected and will be able to handle files in other encodings too. The drawback is that the user may not known which encoding to provide or provide it in the wrong way.

Using your experience and the information provided, which solution do you think would be best suited for the task?

- [ ] Solution 1
- [ ] Solution 2
Why did you choose this solution?
Investigating the usefulness of design points in a developer's decision making process

In the previous question you chose the solution:

and motivated it by saying:

When deciding between different solutions, our choice can be based on tradeoffs that we have identified between the various options.

Did you consider any tradeoff when making your decision? And if so, which ones?

When deciding between different solutions, our choice can be based on tradeoffs that we have identified between the various options.
Investigating the usefulness of design points in a developer's decision making process

Task Two

At this point, we want to introduce you to the concept of a design point. A design point is a piece of a discussion that arose between the developers of the project about a software system's design that a software development team needs to make.

For the next set of questions, imagine that you had access to a tool that retrieves design points relevant to the task at hand. This information has been extracted from discussions recorded as part of normal work in bugs, pull requests developers of the Node.js project. After the description of the task, you will be provided the result that the tool would have returned for this tasks. To help understand the design point, context about the design point from the discussion in which it appears is also provided.

The design points will be presented as paragraph within a gray box. The context will be provided as the previous and next paragraph in the original discussion, presented as the paragraphs before and after the design point box. Below you can find an example of what it will look like.

**Previous Paragraph**

**Design Point**

**Next Paragraph**

Task Description

You are a developer who has recently started contributing for the Node.js project. You have found an issue in which a third party developer is reporting that the existsSync() method is erroneously indicating that a file does not exists. The existsSync() method is a simple synchronous method that returns a boolean True if the provided path (to a directory or to a file) exists, False if it does not. The developer has manually verified that the file exists, and the path to it is correct, so they can't understand the behaviour. After some discussion, you discover that the issue is caused by Node.js not having permissions to read the directory in which the file is located.

Your goal for this task is to find a solution to clarify the behaviour of existsSync() in the particular case in which Node.js does not have permissions to read the directory in which the path is located.

**Design Point #1**

I'm not sure I understand what is the difference with the EPERM error on Windows and EACCESS on UNIX.
Isn't one of the biggest gripes about getting rid of existsSync() that it gives them a simple Boolean response? Throwing is also a breaking change on a heavily used API. You could argue that it's a bug fix but I think people will still yell loudly.

No, it's that the others require excess options and stuff.

**Design Point #2**

Is anyone positive for undeprecating it and leaving its functionality?

Should EACCESS or EPERM just say the file exists? Saying it doesn't actually seems less correct in most cases?

@Fishrock123 see the example here #4077 (comment) There is no data to say it exists or it doesn't.

**Design Point #3**

If exists/existsSync throws early if it can detect one of those exceptional conditions I really don't mind. Also whether it returns a genuine boolean or simply a truthy/falsy value I also really don't mind. fs.accessErrorSync solves a problem I don't have, I, and I think thousands of others, feel that there's absolutely nothing problematic with the definition of whether a file 'exists' or not. I think we are perfectly happy with the fact that the mere existence of a file at a particular point in time provides no guarantees that anything can actually be done with it at a later point in time. There's no need to confuse/couple these two requirements together.

@amb26 repeating myself: we can't give a straight answer to whether a file exists or not if its inside a directory which we cannot enter.

/root/.ssh/id_rsa (your user does not have the rights to access /root, but the file exists)

**Design Point #4**

Let me know about the isDarwin flag.

I think the past few comments in the main comment thread didn't really support the isDarwin flag. I'm not really in favor. I think Apple needs to fix this, not us. (not saying I'm not sensitive to this issue and what you're trying to achieve, but as @saghul pointed out, we'll never realize it when Apple improves performance).

I would like to ask though, has this been benchmarked on Linux and Windows? Is it really OSX only? Or has that been the only OS tested?

**Design Point #5**

@saghul I have not tested it on Windows, but I can do it if you want.

I don't see the need, different Windows versions could yield different results, same goes even for Linux.

I wanted to dig more in what the problem is, and it is definitely an OS X thing - I'd guess they are allocating the memory somewhere (at least it shows the same performance pattern when I do the malloc myself).

**Design Point #6**

I think the past few comments in the main comment thread didn't really support the isDarwin flag. I'm not really in favor. I think Apple needs to fix this, not us. (not saying I'm not sensitive to this issue and what you're trying to achieve, but as @saghul pointed out, we'll never realize it when Apple improves performance).

I would like to ask though has this been benchmarked on Linux and Windows? Is it really OSX only? Or has that been the only OS tested?

@ronkorving the reason why I dig into this rabbit hole was because I could not explain why passing an array was slower on my box (OS X). I have also tested on Linux (virtualized), and passing an array is 20% faster than calling concat(). I have not tested on Windows, should I?
After some time reflecting on the task, you have come up with three possible solutions.

- **Solution 1**: existsSync() should return a boolean: True if the file exists, or False if the file doesn't exist or if a directory on the path can't be accessed. This solution will maintain the current functionality, but it will be impossible to determine if a file does not exist or if it's located in a directory not accessible.

- **Solution 2**: existsSync() should return a boolean: True if the file exists, or False if the file doesn't exist; if a directory on the path can't be accessed, it should throw an error: EACCESS (for Windows) or EPERM (for Unix). This solution will allow to distinguish between the file not existing and the directory not being accessible, but it will break existing code when an error is thrown.

- **Solution 3**: existsSync() should return a boolean: True if the file exists, or False if the file doesn't exist. In the case Node.js is being run on Windows, it should internally make use of fs.accessSync() to handle cases in which the directory is not accessible and avoid throwing an error. This solution would handle every case correctly, but introduces an additional dependency and splits the execution depending on the OS.

Using your experience and the information provided, which solution do you think would be best suited for the task?

- Solution 1
- Solution 2
- Solution 3
Why did choose this solution?

Were you able to understand the information contained in the design points? Please explain.

Did you find the information from these design points to be valuable for the given task? How?

Please order the design points (by drag and drop) in order of value for the given task, with the most valuable design point at the top and the least valuable at the bottom.

Isn't one of the biggest gripes about getting rid of existsSync() that it gives them a simple Boolean response? Throwing is also a breaking change on a heavily used API. You could argue that it's a bug fix but I think people will still yell loudly.

Should EACCESS or EPERM just say the file exists? Saying it doesn't actually seems less correct in most cases?

@amb26 repeating myself: we can't give a straight answer to whether a file exists or not if its inside a directory which we cannot enter.

I think the past few comments in the main comment thread didn't really support the isDarwin flag. I'm not really in favor. I think Apple needs to fix this, not us. (not saying I'm not sensitive to this issue and what you're trying to achieve but as @saghul pointed out, we'll never realize it when Apple improves performance).
I don't see the need, different Windows versions could yield different results, same goes even for Linux.

I would like to ask though has this been benchmarked on Linux and Windows? Is it really OSX only? Or has that been the only OS tested?

`Call()` interacts poorly with domains and other mechanisms that hang off `async_hooks` so it's really only safe for internal JS code. W.r.t. handle leaks `Call` does that as well (the return value.)

It would be better to instead set these variables in the config object passed to `fork()`

Do you have any final comment you want to share?

---

Version 1.3

Ethics ID: H21-03850

05/02/2022

Powered by Qualtrics