Otto von Guericke University of Magdeburg

School of Computer Science

Master's Thesis

Effect of annotation-based variability on program comprehension

Author:

Maria Kanyshkova

April 14, 2019

Advisors:

Prof. Gunter Saake

Dipl.-Inf. Wolfram Fenske

Department of Databases and Software Engineering

Kanyshkova, Maria:
Effect of annotation-based variability on program comprehension
Master's Thesis, Otto von Guericke University of Magdeburg, 2019.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Since the 20th century we have seen an increase in individualised goods across all industries. Whether fast food, computers or car manufacturing, customers enjoy the possibility to customise products by adding, removing and substituting parts. This has been made possible with the introduction of the concept of a product line, which is defined as "a set of products in a product portfolio of a manufacturer that share substantial similarities and that are, ideally, created from a set of reusable parts" [SABK13]. By using product lines the so called mass customization becomes possible, where manufacturers produce different standardised and reusable parts that are then chosen by the customer to create their own variation of the product.

In software development mass customisation is implemented through software product lines. Just like their counterparts from other industries, software product lines are constructed from reusable parts which can be combined to make a tailored product. These reusable parts are called core assets and are used to implement features. A feature is used to communicate differences between products and contains a certain functionality within the software product line. This allows to streamline development for domains where the resulting products have similar but not identical requirements, e.g. different operating systems [SABK13].

There are different approaches for implementing software product lines. One of those approaches is annotation-based variability (also known as preprocessor-based variability). One possible implementation is through `#ifdef` directives or similar constructs. Developers use such directives to annotate code fragments inside a body of code, making feature code that is surrounded by annotations. During the compilation, feature code is removed from the code according to the configuration file, to generate a software variant [KATS12].

A big problem of software product lines is their increased complexity compared to software systems without variability. As code is split between features across several classes or files, it can be hard to keep track which parts of the code will be in a specific

variant of the software. In annotation-based approaches the code of a feature can be scattered across the software system or tangled with the base code or with the code of other features, making program comprehension complex [LAL+10]. Annotation-based variability in form of preprocessor annotation has an especially bad reputation but is widely used in practice [MKR+15]. Such code is hard to read and maintain [MKR+15][MBW16].

Goal of this Thesis

When implementing variability, the code may end up containing variability-aware code smells. Code smells are design flaws in the code that usually indicate the need of a restructuring of the code, i.e. a refactoring. There are specific variability-aware code smells, defined through metrics [FSS17], in software-product lines but so far it is unclear whether they have a negative impact on program comprehension [FS15]. There are many studies [LWE11, MKR+15, LKA11] that investigate the impact of certain annotation-based variability mechanisms (e.g. the effect of undisciplined annotations [LKA11, SLSA13]) and found that through use of annotations the source code may become obfuscated, therefore hindering program comprehension and introducing subtle errors [FSS17]. Since the variability-aware code smells are usually defined through several such aspects, it is not possible to transfer the findings of these studies to the code smells. In this thesis we want to find out how annotation-based variability affects program comprehension of experienced developers. Additionally we hope to clarify how reliable code metrics are in regards of predicting code comprehension of experienced developers.

To test whether developers understanding of code with varying amounts of variability-aware code smells differ, we conduct an online survey with code examples of smelly variable code. The code smells of the examples vary from smelly to very smelly. The online survey is presented in two variants, where we alternate between the original code fragments (as taken from the open source projects Vim and Emacs) and refactored code fragments, where the annotation amount and code smells have been reduced without affecting functionality. The survey asks questions about code understanding which we will use to establish whether the quality and quantity of annotations have an impact on program comprehension. The goal is to establish how well metrics can predict code understanding of real developers. In a nutshell, our hypothesis is that code that reaches higher smelliness levels (e.g. through the use of more nesting, more `#ifdef` statements, etc.) is harder to understand.

The results of this thesis could help further understanding on the effect annotation-based variability has on program comprehension. Moreover, the results of this survey could improve automatic detection of problematic annotation use in variable software.

Structure of the Thesis

The thesis is structured in the following way. Chapter two gives an overview over the relevant background on variable software, variability-aware code smells, program

comprehension, conducting online surveys and linear regression. Chapter three explains the details of the survey design as well as the chosen variability-aware metrics. In chapter four we present the results of our survey and discuss them. Chapter six contains a summary of the thesis. In chapter seven we will give an overview on related work. In chapter eight we summarise the thesis and give a perspective over possible future research.

2. Background

This chapter introduces the background of the themes this thesis touches upon. This includes an introduction of variable software, variability-aware code smells and their metrics, as well as conducting online surveys with program comprehension in mind.

2.1 Variable Software

Variable software is the answer to the rising desire for individualised goods and cheap, easy to maintain software systems [SABK13]. Also known as software product lines, variable software uses a set of features that can be combined to implement new software variants. Features are characteristics of the software system, in software product lines they are used to communicate commonalities and differences of product variants. Moreover features are used to guide the structure, reuse and variation between the phases of the software life cycle.

Variability can be expressed in terms of common and optional features, which is referred to as feature modelling [SABK13]. A feature model specifies how many valid products a software product line can produce. More specifically, since some features might not be compatible due to conflicting implementations and some features may require the presence of other features to build on them, feature models describes such relationships between features and defines which feature configurations are valid.

In practice, there are two main approaches to implement variable software: annotation-based and composition-based variability [SABK13]. Those approaches influence the structure of the source code and the way products are generated.

Composition-based approaches implement features as units (e.g. files, classes, etc.) [SABK13]. An example would be a framework with plug-ins, where each plug-in is a feature. Products can be generated by including or excluding different plug-ins. This approach has the disadvantage that the complexity of the mapping between features can become high and that the variability's granularity is coarse.

Annotation-based approaches have a single code base where an annotation marks the feature a particular code fragment belongs to [SABK13]. When a product variant is derived, all code from deselected features is excluded from compilation. Only not annotated code and selected feature code remains in the product code. Because of the ease of use and because many program environments already provide the means to implement them, annotation-based approaches are widely used in practice. The disadvantages of this approach are the increase of complexity and a lack of modularity. Often the annotation-based approach is implemented using the C preprocessor. The next section explains how CPP-based variability is implemented.

2.1.1 CPP-Based Variability

In this thesis we use C functions which implement annotation-based variability through the C preprocessor. The C preprocessor (CPP) is used to transform the source code for compilation [KR06]. The transformation is done using directives and macros. Macros are shortcuts of definitions that will be expanded when compiled. Directives control the inclusion of definitions from other files and define macros. Directives like `#if` and `#ifdef` also allow conditional compilation of the code. When using conditional compilation, the compiler will include or exclude certain parts of the source code based on the given conditions.

Conditional compilation can be used to implement variability [KR06, LAL+10]. Features are then implemented by guarding the variable code with `#ifdef` and controlled through feature expressions. The name of the macro in the feature expression is referred to as a feature constant.

CPP-based variability has implications for the structure of the source code [LWE11, DSRS03]. The directives used to implement variability are intrusive and break the flow of the code. Next, the feature expressions may use several feature constant which leads to a higher difficulty when tracing the inclusion condition of a certain part of code. Finally, tangling and scattering are likely to be introduced since all feature code is located in a shared code base. Therefore CPP variability can have a negative effect on understanding and changing annotated code.

2.2 Variability-Aware Code Smells

Code smells are bad practises in code that can come from design flaws or from code decay [Fow00]. They lead to a harder understanding and therefore maintaining of the code. Code smells can be resolved by refactoring the code through certain techniques.

Software product lines, as every long-living software system, is prone to code decay. Code smells are affected by the variability, altering them and making them harder to find [FS15, FSMS15]. Variable software creates new opportunities for introducing design flaws into the code. The existing code smells do not capture the nature of code decay introduced by variable software systems. Therefore new variability-aware code

smells were derived from already existing. Here these variability-aware code smells are presented.

Inter-feature Code Clones are a variable-aware code smell that expands the Duplicated Code code smell [FS15]. Similar to the original code smell, code is copied with no or little alteration either within a feature or across several features. This can lead to inconsistent changes as development progresses and increases complexity of the code.

The Annotation Bundle smell is derived from the Long Method smell and describes a method with many variable parts [FS15]. Therefore the code is heavily annotated (e.g. with CPP directives), possibly with nested annotations. While the compiled function may end up being short, the variable source code is difficult to understand for a certain configuration as the annotations obscure the view of the core functionality. Maintenance becomes harder as special care is needed to alter the code and because locating the origin of a bug is more difficult without knowing the exact configuration. In this thesis the Annotation Bundle code smell is affecting our chosen examples.

The Long Refinement Chain is derived from the Long Method smell as well and is the composition-based counterpart of the previously described smell [FS15]. It abuses feature refinements through excessive use. Code is then harder to understand and bug finding is obstructed because the affect of a single feature is unclear.

The code smell Latently Unused Parameter derives from both Long Parameter and Speculative Generality smells [FS15]. The original smells describe methods with many parameters and unused functionality respectively. The variability-aware variant arises when optional parameters (i.e. such parameters that are only needed for certain features) are present. It is problematic because the developers assumes a declared parameter to have some effect on the outcome of the code. Whenever this isn't the case, confusion may arise and the code becomes harder to understand.

2.2.1 Variability-Aware Metrics

It is possible to detect variability-aware code smells using metrics [FSMS15]. A software metric is a collective term used to describe measurements performed on a source code [FN99] in order to provide information on a software system. The variability-aware metrics are briefly described in this section because we investigate whether they properly reflect human program comprehension and subjective perception of code quality.

In this thesis we use several previously established metrics in order to compute informations about our examples [FS15, FSMS15]. The lines of code metric LOC counts the source code lines of the code fragment, ignoring blank lines and comments. For better control of LOC changes between different examples, we do not use the LOC metric, instead we introduce the log2loc metric which tracks changes in LOC amounts through the binary logarithm. The added complexity through the annotations is reflected in the LOAC, NONEST, NONEG, NOFL, and NOFC metrics. The LOAC metric also ignores blank and commented lines and counts lines inside annotations. Lines found in annotations are only counted once. We do not track this metric on its own, rather we

compute the ratio of annotated to not annotated code and use this through the metric loacratio. The NONEST metric reflects the accumulated nesting depth of annotation in the code fragment. The first `#ifdef` in the code has a nesting depth of zero. A second `#ifdef` within the first would have a nesting depth of one and so on. Nesting depths are accumulated, two feature locations with a nesting depth of one each would have a NDacc value of 2. NONEG is the number of negation in the `#ifdef` directives. It includes constructs like `#ifndef`, `#if !defined`. Since `#else` branches can be expressed as `#if X #endif #if !X #endif`, they also increase NONEG by one. NOFL is the number of feature locations, i.e. blocks annotated through a preprocessor directive like `#ifdef`. A complex expression like `#ifdef A && B` is counted once. An `#ifdef` with an `#else` is counted as two feature locations. NOFC is the number of unique feature constants in the code fragment. Feature constants occurring in multiple feature locations are counted only once.

The metric ABSmell describes how badly a function suffers from several individual smells, namely the LocationSmell, the ConstantsSmell and the NestingSmell. The individual code smells can be differently weighted, in order to prioritise the effect a certain smell has. The LocationSmell includes the ratio of annotated code to all code, multiplied with the number of feature locations NOFL. This is based on the assumptions that a bigger amount of LOAC in a smaller amount of LOC and several small feature locations are more problematic. The ConstantsSmell consists of the ratio between NOFC and NOFL in order to capture the average complexity of feature expressions within the code fragment. NestingSmell is the ratio between NONEST and NOFL and accounts for nesting [FS15].

2.3 Program Comprehension

The term program comprehension refers to the activity of understanding how a software system or part of it works. Program comprehension is a task often undertaken before changing code, as developers have to explore relevant parts of the code in order to perform the change. The amount of time spent on obtaining and sharing knowledge about a software system can be as high as 50 percent of a developers work time [MTRK14].

There are several categories of program comprehension models: top-down, bottom-up and integrated models [FSF11]. Top-down models explain how programmers come to a general hypothesis of the purpose of a program and the following stepwise refinement as source code is evaluated. In bottom-up models, the developer starts with code fragments and proceeds to group them into semantic groups to get a better understanding of the functionality and finally the general purpose of the program. The models can be also be combined into the integrated model.

However, since program comprehension is an internal process, we can't observe it directly [FSF11]. Therefore we need to use other indicators to measure it, for example metrics, self assessment, tasks and think-aloud protocols. The metrics used in this thesis are explained in Section 2.2.1. Metrics alone shouldn't be used for program comprehension assessments because they lack human interaction. Self estimation refers to

the subjects subjective rating on how well they understood the code. The problem with this indicator is that it can be easily biased. Subjects can also be asked to perform tasks on the code, e.g. finding a bug. To perform the task successfully, the subjects have to understand the source code therefore perform a comprehension process. Finally in think-aloud protocols, subjects audibly comment their work process on a code fragment, making the process observable.

We decided to use metrics to establish how well the subjects are expected to perform with our examples. Additionally we let the subjects solve tasks that are later evaluated regarding correctness and completeness. The subjects are also asked to self-estimate how well they comprehend the code and how easy maintaining this code would be.

2.4 Designing and Conducting Surveys

2.4.1 Survey Errors

A good survey design is vital for comprehensive and accurate results. When conducting a survey, it is important to consider several points of failure and navigate around them. There are four main survey errors that can lead to a point of failure.

The coverage error is present when the population in question isn't adequately covered. This happens if there are differences between the surveyed and non-surveyed people. These differences would lead to a result that wouldn't fully represent the population.

The sampling error happens when the surveyed sample isn't large enough to ensure random drawings. How large the group of participants has to be can be calculated based on the population size, the confidence level and the margin of error. The population size includes all people the survey aims to represent. The confidence level is the probability that the used sample accurately reflects the attitudes of the population. The margin of error is the percentage range that indicates possible deviations of the population's responses from the sample's responses.

The non-response error is present when people who respond to the survey are different from those who do not respond. The more people respond to the survey, the smaller the non-response error. Accessing and submitting the questionnaire must be made easy for the participants.

The measurement error occurs if the provided answers are inaccurate, e.g. because of the question design. In order to avoid this error, the survey must be designed in a way to encourage honest answers and give options to provide feedback [Dil11].

2.4.2 Benefits and Costs of a Survey

Survey response as social exchange is meant to increase the likelihood of response to the survey by increasing the benefits and decreasing the costs of participation. The benefits of participation are what the participant expects to gain from filling in the survey. They can be financial incentives, social validation through the circle of other

participants, support of group values or personal gratitude from the surveyor. The cost of participation is what the subject loses through participating in the survey, e.g. time.

There are certain ways of increasing the benefits of participation in a survey. One of them is providing information on the survey, especially why it's being conducted and how the person receiving the information can help. Asking for help further encourages responses. This is especially important in the initial e-mail and its subject matter as this will be the first thing the recipients are going to see of the survey. A positive regard shown by personally addressing the subjects is more motivating than a generic e-mail. Likewise, thanking the participant for the consideration of participating in the survey is important.

A more interesting questionnaire will have higher response rates. This includes both the covered topic and the visual representation of the questionnaire. The layout and design should be engaging and the question order should be thoughtfully designed. The questions should also be easy to understand and answer. Materialistic rewards can be used as a motivator, although different people will perceive different rewards as appropriate for their time. A non-materialistic reward could be the support of group values of a certain surveyed group or social validation of the participant within that group.

The perceived costs of participating can be decreased. Participating in the survey should be easy and convenient, e.g. through a personal e-mail with a direct link to the survey. The questionnaire should also be short and easy to complete and ask for little personal information about the participant. The language used throughout the questionnaire and further communication shouldn't subordinate or disrespect the participant. Finally, it can help to refer to previously taken tasks a person took, that are linked to the survey [Dil11].

2.4.3 Designing with Program Comprehension in Mind

There are several potential problems when designing and conducting a survey with program comprehension in mind [DPSK07]. For instance, the key variables can simply include the accuracy of the answers and the completion time or make use of newer technology and track eye movement or other body reactions. A big issue is the choice of appropriate subjects. Students, while convenient, often lack the experience needed to answer the questions. Therefore they are often not adequate representation choices for the population tested. Professionals of the tested field would be optimal but this raises the question how to engage them, since their time and motivation to participate in surveys might be limited.

When testing several approaches, another issue is the identification of benchmarks to compare the collected data [DPSK07]. Standardisation of both the design format and the methodology used to conduct and analyse the survey is important as well.

2.5 Regression Models

Regression is used to predict values from data items or to study trends in the data. We use regression to study trends in the answers our subjects provided in the survey. In its simplest form, we have a set of pairs (xi, yi) in a dataset and we want to establish the dependence between y and x, so that we can predict values of y for new values of x. We refer to xi as explanatory variables and yi as dependent variable [Joh11].

2.5.1 Logistic Regression and Negative Binomial Regression

Linear regression is a very important method of data analysis [OO18]. It serves the purpose of determining model parameters, model fitting, assessing the importance of influencing factors and prediction.

When using linear regression, we assume that the dependent variable y is obtained by evaluating a linear function of the explanatory variables and adding a normally distributed random variable [Joh11]. The random variable has zero mean, because we can't predict its value. The factor ß is a vector of weights, which we are to estimate. ß tells us how much influence a change in the independent variable x has on the dependent variable y.

Geometrically [OO18] the pairs of data form a scatter plot in the plane. During a regression, we draw a line through the scatter plot, so that this line has the best fit. The line is also referred to as the regression line.

Logistic regression is another regression model, where y can take the values of 0 or 1 [OO18]. This means logistic regression can be used to perform regression over a binary dependent variable [Cox58]. The observations on the dependent variable would take one of two possible forms, "success" or "failure", or put differently "true" or "false".

The negative binomial regression provides a mean to model over-dispersed count data [ZKJ08]. The influence of the independent variable x on the dependent y is assumed to be a negative binomial distribution.

To interpret the strength of correlation between the dependent variable y and a significant independent variable x, we make use of the odds ratio [Szu10]. The odds ratio says that two events are independent, if and only if the odds ratio between x and y is exactly one. An odds ratio lower than one means that an increase in x decreases y. An odds ratio higher than one means that an increase in x increases y as well. The increase can be established by subtracting one from the odds ratio. When multiplied with 100, we obtain the percentage y increases or decreases when x increases by one.

2.6 Summary

In this section we explained what variable software is and described the two most used approaches to implement it: annotation-based and composition-based. In this thesis we focus on the annotation-based approach, implemented with the C preprocessor

CPP, that is described in Section 2.1.1. We presented the variability-aware code smell Annotation Bundle, and briefly described its non-variable counterparts. We discussed metrics used to detect variability-aware code smells that we use.

Program comprehension was explained. The three models of program comprehension (top-down, bottom-up and integrated) were presented. We pointed out that since program comprehension is an internal process, measuring it isn't straightforward. We presented four possible approaches to indirectly measure software comprehension through the use of self-assessments, tasks, think-aloud protocols and metrics. We used metrics, tasks and self-assessments to design our survey.

In Section 2.4 we discussed how to design and conduct an online survey. The four common design errors (coverage, sampling, non-response and measurement) were presented. We explained the benefits and costs for the participant of a survey and how the benefits can be increased, while the costs decrease. At last, we discussed the difficulties of designing surveys with program comprehension in mind.

In the final section we explained how regression models can be used to interpret data. Regression helps finding trends in the data and make predictions based on these trends. In this thesis we use logistic regression for boolean values and linear regression for everything else, thus these two regression models are explained.

3. Methodology

In this chapter we discuss how we selected the code examples and questions for our survey. We also explain how we selected our subjects. Finally we present our research questions and detail how the survey questions help answering them.

3.1 Survey in a Nutshell

This section gives an overview of the important aspects of our survey. It serves as a quick overview of the survey design.

Objective: We wanted to compare whether code smell metrics adequately reflect the objective and subjective program comprehension of experienced developers. To test this, we developed a survey and distributed it to open source developers.

Subjects: We asked experienced open source developers from several projects from Github to lend us their expertise. The project selection has been made through consultation of the "The Love/Hate Relationship with the C Preprocessor: An Interview Study" paper [MKR+15] as well as identifying popular open source projects in C on Github.

Source code: We selected heavily annotated functions that have been identified in previous work [FSMS15]. We selected at least one code fragment of each level of smelliness, to establish a relationship between the smelly code metrics and program comprehension. We selected a total of five examples and refactored four of them in order to reduce the amount of annotations. The refactorings had better metrics and could therefore be used to compare finer grained differences between the functions.

Tasks: The subjects were divided into two groups. Each group has been given a variant of a questionnaire with different original and refactored code fragments. One code fragment has been left untouched in both versions as a comparison baseline between the two groups. For each function, the questionnaire contained questions that tested actual and subjective program comprehension.

3.2 Goal of the Survey

Both program comprehension with real developers and metrics to measure expected program comprehension within smelly variable code has been done. However, confirming whether the metrics properly reflect both the actual and the perceived program comprehension of the developers has yet to be done. The goal of this thesis is to close this gap and establish a relationship between the metrics and the experience of real people. In order to achieve this goal, we defined the following research questions:

RQ1: Do different amounts of preprocessor use affect developer effectiveness during program comprehension tasks?

RQ2: Do metrics of preprocessor use reflect subjective assessments of code quality?

RQ3: Which reasons do developers mention for poor program comprehension?

To test RQ1 we chose code examples with varying amounts of preprocessor use (see Section 3.3) and resulting variable smell metrics. The subjects were given comprehension tasks to establish whether they understood the example code. Afterwards we performed regressions (see Section 2.5) to link the correct answers from these questions to our metrics. The goal of RQ1 is to establish whether varying amounts of preprocessor use affect the correctness of solving the comprehension tasks.

For RQ2 the developers were asked to give their subjective evaluation of the code regarding different aspects like understandability and maintainability. To measure their evaluation we used self-assessment questions, where the subjects evaluated whether the preprocessor use in the function is adequate and how easy it is to understand, maintain, expand the code, as well as detect bugs in it. The goal of this research question was to establish whether the subjective perception of problems in the code matches with the estimate we gained from our metrics.

In addition to aforementioned quantitative questions, the questionnaire contained a qualitative part in which subjects could detail why they felt the preprocessor use is not valid for the function. To answer RQ3 we grouped these answers into categories and assigned a mood to each answer through open coding. Later we evaluated how often a specific category would be named and whether the mood of the answer could be linked to the correctness and the subjective assessment from RQ1 and RQ2. The goal of RQ3 was to establish further reasons for poor program comprehension, that were important for our subjects when working on the examples.

3.3 Code Examples and Questions

The survey had a total of ten pages, including an introduction and a closing page, as well as two pages where short context for the code examples was given. One page contained questions regarding the social background of the subject, such as his age, country and gender. Additionally we asked how many years of programming experience in general and specifically in C the subject has. Moreover, the subject was asked to rate their

Table 3.1: Metrics of used examples

Example	1	2		3		4		5	
Name	vim18	vim15		vim13		emacs12		emacs11	
Refactored	0	0	1	0	1	0	1	0	1
ABSmell	3.02	7.2	4	11.78	10.72	5.86	2.33	6.12	3.88
LOC	21	31	30	84	82	25	26	35	39
LOAC	16	26	12	69	67	21	8	26	19
loacratio	0.76	0.84	0.4	0.82	0.82	0.84	0.31	0.74	0.49
NOFL	2	6	4	12	11	4	2	6	4
NOFC	2	6	5	4	4	4	1	4	3
NONEST	1	6	2	9	7	2	0	4	0
NONEG	0	1	2	0	0	2	1	2	1

general and C programming experiences on a Likert scale ranging from beginner to expert.

There were a total of five code examples from two text editors. Three of the examples were from the vim project and two from the emacs project. We chose the initial code fragments from the SCAM dataset [1] of smelly code according to a set of requirements. The SCAM dataset is a collection of functions and their variability-aware metrics from different C projects. The metrics for the functions were computed through the tool SKUNK [FSMS15]. The metrics are shown in Table 3.1. Our requirements for the selected functions were small size and different values of code smells. Short functions were needed to ensure the example can be viewed at once on a regular desktop monitor. Our threshold was 40 LOC, although we used one especially smelly example with 80 LOC. Next, we tried to include examples for every level of smelliness (see Section 2.2). We selected the code examples vim18, vim15, vim13, emacs12 and emacs11 with the smelliness levels of R-1, R0, R1, R-1 and R0 respectively. In theory examples of same smelliness should effect similar program comprehension.

We designed two versions of the questionnaire. In each there were three original code fragments and two that we refactored by reducing the amount of annotations and restructuring them. With the refactorings we aimed to improve readability and, thus, comprehension of the presented code. The refactorings had better smell metrics than the originals (see Table 3.1). As such we expected the subjects to perform better in these examples than the subjects who were presented the original code. The complete order of both original and refactored examples from least smelly to most smelly is the following: emacs12 refactored, vim18, emacs11 refactored, vim15 refactored, emacs12 original, emacs11 original, vim15 original, vim13 refactored and vim13 original.

Listing 3.1 and Listing 3.2 show the original and refactored code of example vim15. The original example uses undisciplined and complex annotations. This was improved

[1] https://www.isf.cs.tu-bs.de/cms/team/schulze/material/scam2015skunk/

in the refactored example. The code functionality and not annotated code were largely left untouched, though code replication was not avoidable in some refatorings.

Listing 3.1: Vim15 original

```
    char_u  *
fix_fname(fname)
    char_u  *fname;
{
#ifdef UNIX
    return FullName_save(fname, TRUE);
#else
    if (!vim_isAbsName(fname)
        || strstr((char *)fname, "..") != NULL
        || strstr((char *)fname, "//") != NULL
# ifdef BACKSLASH_IN_FILENAME
        || strstr((char *)fname, "\\\\") != NULL
# endif
# if defined(MSWIN) || defined(DJGPP)
        || vim_strchr(fname, '~') != NULL
# endif
        )
    return FullName_save(fname, FALSE);

    fname = vim_strsave(fname);

# ifdef USE_FNAME_CASE
#  ifdef USE_LONG_FNAME
    if (USE_LONG_FNAME)
#  endif
    {
    if (fname != NULL)
        fname_case(fname, 0);
    }
# endif

    return fname;
#endif
}
```

We aimed to keep the number of LOC small in order for the examples to fit on one screen without scrolling. This is how we ended with four original and three refactored examples with 21-39 LOC. However, to include one example with a high smelliness rank we had to resort to a code fragment of over 80 LOC. In our examples varying amounts of code were annotated. In vim18 76% of all code was annotated. In vim15 83% were annotated in the original version and 40% in the refactored. For vim13 both the original and refactored examples had 82% of annotated code. Emacs12 had 84% of annotated

Listing 3.2: Vim15 refactored

```
#ifdef UNIX

        char_u  *
fix_fname(fname)
        char_u  *fname;
{
        return FullName_save(fname, TRUE);
}

#else /* !UNIX */

        char_u  *
fix_fname(fname)
        char_u  *fname;
{
        int is_rel_name = !vim_isAbsName(fname)
                        || strstr((char *)fname, "..") != NULL
                        || strstr((char *)fname, "//") != NULL;
# ifdef BACKSLASH_IN_FILENAME
        is_rel_name = is_rel_name || strstr((char *)fname, "\\\\")
           != NULL;
# endif
# if defined(MSWIN) || defined(DJGPP)
        is_rel_name = is_rel_name || vim_strchr(fname, '~') != NULL
           ;
# endif

        if (is_rel_name)
        return FullName_save(fname, FALSE);

        fname = vim_strsave(fname);

# ifdef USE_FNAME_CASE
#  if !defined(USE_LONG_FNAME) || USE_LONG_FNAME
        if (fname != NULL)
        fname_case(fname, 0);
#  endif
# endif

    return fname;
}
#endif
```

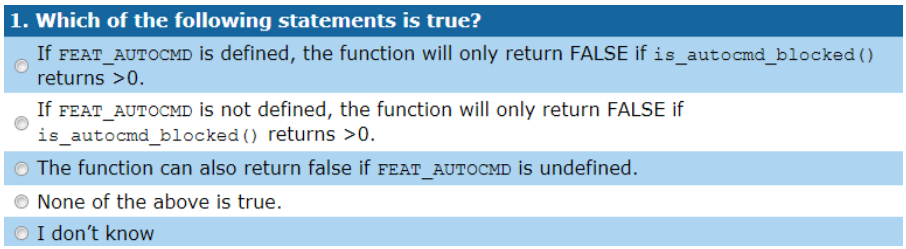

Figure 3.1: The first question of example vim18

code in the original and 31% in the refactored version. The emacs11 versions had 74% of annotated code in the original and 48% in the refactored example.

To have a comparison between the two questionnaires, we left the original code of vim18 in both versions as baseline. By comparing the results of the questions for vim18, we wanted to see whether there are considerable differences between the demographics and skill levels of the two questionnaire versions.

The other four code fragments were presented in either the original or the refactored version. Specifically, questionnaire 1 had the original code for vim15 and emacs12 and the refactored code for vim13 and emacs11. The second questionnaire had the original code for vim13 and emacs11 and the refactored code for vim15 and emacs12. The examples were given in the same order.

We aimed to provide both quantitative and qualitative questions to ensure that the subjects provide measurable expertise on the one hand but also are able to provide their own thoughts on the other. Additionally we asked for feedback at the end of the survey.

Each example had six questions. The first Figure 3.1 presented several statements of which one could be true. The subject was asked to select the true statement.

The second question Figure 3.2 asked the subject to give a valid feature combination that would lead for a certain line in the code to be executed. The subject had to understand the use of annotations in order to correctly answer the first two questions.

The third question asked the subject which lines of code they found hardest to comprehend. As can be seen in Figure 3.3, the subjects could type in single line numbers

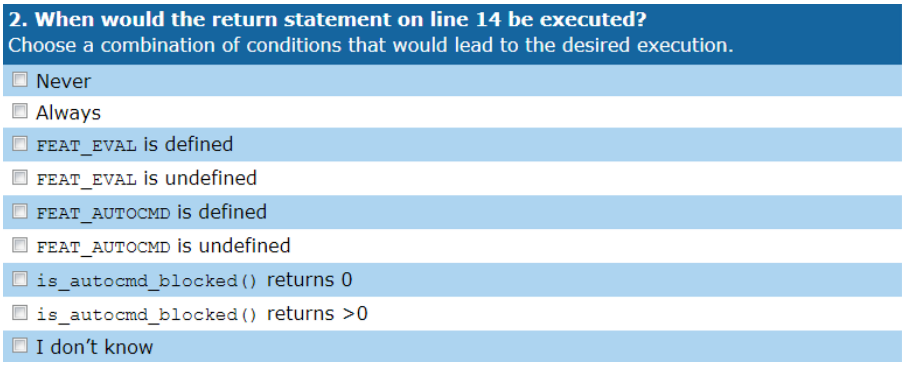

Figure 3.2: The second question of example vim18

or line ranges, separated by commas. The results of this question were used to identify the least understood lines in the code fragments.

The fourth Figure 3.4 and fifth Figure 3.5 questions belong together and asked the subject to rate the appropriateness of the preprocessor annotation and, in case they were found to be inappropriate, how the participant would attempt to fix them.

Finally, the sixth question Figure 3.6 contained four Likert scales that could be used to rank the example with regard to the difficulty of understanding, maintaining, expanding and finding bugs in the code. By asking the self-assessment questions (questions three, four, five and six), we wanted to see the opinion of the subjects regarding annotation use, general code quality and ease of working with the code. Additionally, we wanted to compare the comprehension questions with the self-assessment questions, in order to see whether there is a relation between solving tasks correctly and opinion of the code.

3.4 Subject Selection

In Section 2.4 we established several types of survey errors: coverage, sampling, nonresponse and measurement. In this section, we explain how we tried to minimize these errors in our survey.

To counteract the coverage error, we first tried to establish the amount of C-developers around the world and identify a way to contact them. While the popularity of C is high (e.g. [Spe], [La]), there are no straightforward ways to tell how many C-developers there are. The first idea was to retrieve this information from popular C/C++ internet portals. However, the amount of users between several versioning and forum websites highly

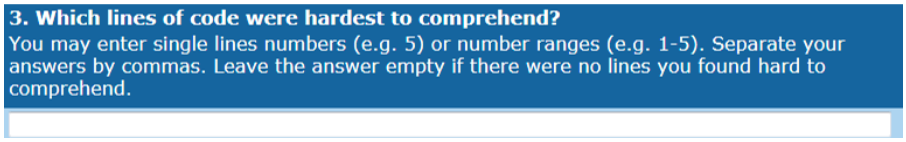

Figure 3.3: The third question of example vim18

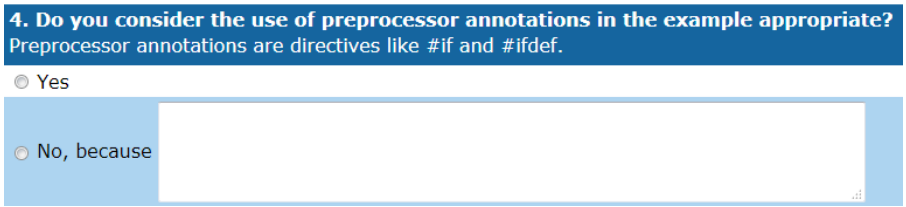

Figure 3.4: The fourth question of example vim18

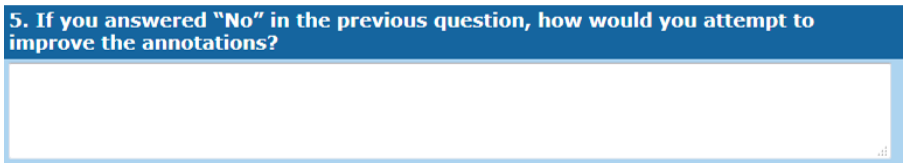

Figure 3.5: The fifth question of example vim18

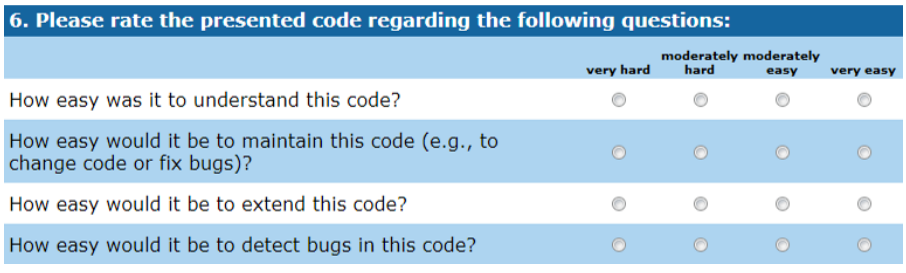

Figure 3.6: The sixth question of example vim18

fluctuates. Additionally those numbers ignore the distinction between experienced and inexperienced developers as well as people who are only interested in the community but do not themselves develop in C. So far the most convincing estimate of the number of C developers in the world has been 1.9 million [Kaz]. We decided to rely on this estimate because it provides a good population size. With a 95% confidence level and a 5% margin of error the sample size would have to be 385. We decided that this is a good population size, because the estimate is high. While keeping the confidence level at 95%, this estimate already provides the highest sample size (e.g. even with a population size of 19 million the sample size would still stay the same).

To contact developers we selected popular open-source C-projects with a big following. Most of the projects were taken from the study of Liebig et al. [LAL+10], the rest has been selected from the trending C projects on Github in October 2018. The projects were analysed to obtain the author names and e-mail addresses from the commits. The tool we used to perform this analysis was only able to access public e-mail addresses (i.e. not those that are hidden behind the Github e-mail anonymiser). As we conduct a scientific study with an open access publication, we believe our process to conform to both the German law regarding unsolicited e-mails [dej] as well as Githubs terms and conditions regarding e-mail address scrapping.

We selected a number of different projects to reduce the coverage error, as well as the sampling error. As such, we expanded our search criteria to include every developer in the last 10 years. We e-mailed 7791 developers. With a total of 544 finished and 1117 total questionnaires, the response rate was 14%.

To ensure a decent number of developers participates in our survey, we made it easily accessible. The online survey was hosted on the survey site of the University of Magdeburg (befragungen.ovgu.de). The site was chosen because it provided all necessary features to design and distribute the survey. The survey was accessible through a personal link distributed by e-mail in the time between the 9th December and the 31st January.

It was important to us to provide anonymity for our subjects, because we wanted them to be assured that their data will not be used outside of this survey. The hosting site provided several ways to ensure the anonymity of the subjects. First of all, the serial numbers provided in the e-mails did not match those that would later be in the finished dataset. As such, we could not track the names of the subjects by comparing the serial numbers. We did not save any browser data or IP and did not use cookies to save the progress of the surveys on the machines of our subjects. Finally, although we asked whether we could contact the subject again for a follow-up interview and the results of the thesis, the e-mails provided to us in this question form were stored separately to the datasets.

3.5 Summary

In this chapter the goals of the thesis were defined in the form of research questions. The research questions ask whether the difference in preprocessor uses affect a) objective

correctness while performing program comprehension tasks and b) subjective assessments of the difficulty of solving the tasks. Furthermore the third research question asks which reasons are given for the perceived difficulty.

In the next section we explained which questions we used in the survey to measure program comprehension. The first and second questions were meant to measure program comprehension through solving tasks on the examples. The fourth and sixth questions measured subjective opinions on code quality and ease of understanding and maintaining the code. These opinions were needed to investigate whether the variability-aware metrics can be used to predict user opinions on the code. The third and fifth questions, as well as the qualitative part of question four were used to find reasons for poor program comprehension. The reasons were extracted through open coding.

Next we presented the variability-aware metrics for our examples. The examples originated from two open-source text editors written in C: vim and emacs. We selected five examples with varying levels of annotations. We refactored four of the examples by simplifying the annotations in the code, in order to investigate whether simpler annotations affect the correctness of task solving and assessments of code quality. Variability-aware metrics were automatically computed for all nine tasks.

In Section 3.4 we explained how we chose our target group. We went from an estimate of 1.9 million C-developers in the world and established that 385 participants were needed to represent this group. Since our initial method didn't yield enough participants, we opted to contact as many developers as we could in the hope to get enough willing subjects.

4. Survey Results

In this chapter we present the results of the survey. First we evaluate the subject demographics. Then we compare how the baseline example vim18 was solved in the two groups of subjects. Afterwards we answer the research questions by evaluating the survey answers and performing regressions on the data.

4.1 Subject Group Comparison

In this section we compare our two groups of subjects. The first group answered questionnaire 1 and the second group questionnaire 2. First we evaluate the self-provided informations of our subjects. Then we compare the results of solving the baseline example vim18 in the two groups.

4.1.1 Subject Demographics

The survey was completed 544 times, of which 273 were questionnaire 1 and 271 were questionnaire 2. In this section a comparison of the subjects between the two questionnaire variants is made.

Figure 4.1 shows the age distribution of the subjects in form of a boxplot. For questionnaire 1 most of the subjects are aged between 32 and 42 years. For questionnaire 2 most of the subjects are aged between 27 and 42 years. The median for both groups lies at 37 and the mean at 36. There is more variety in the ages of the subjects of questionnaire 1 compared to questionnaire 2. The combined boxplot can be seen in Figure 4.2. The majority of the subjects lies between ages 27 and 42, with a median of 37 and a mean of 36.

In Figure 4.3 we see the gender distribution of the participants. A total of 502 are male, 253 in questionnaire 1 and 249 in questionnaire 2. Three participants of questionnaire 1 and eight of questionnaire 2 are female, making a total of eleven female developers.

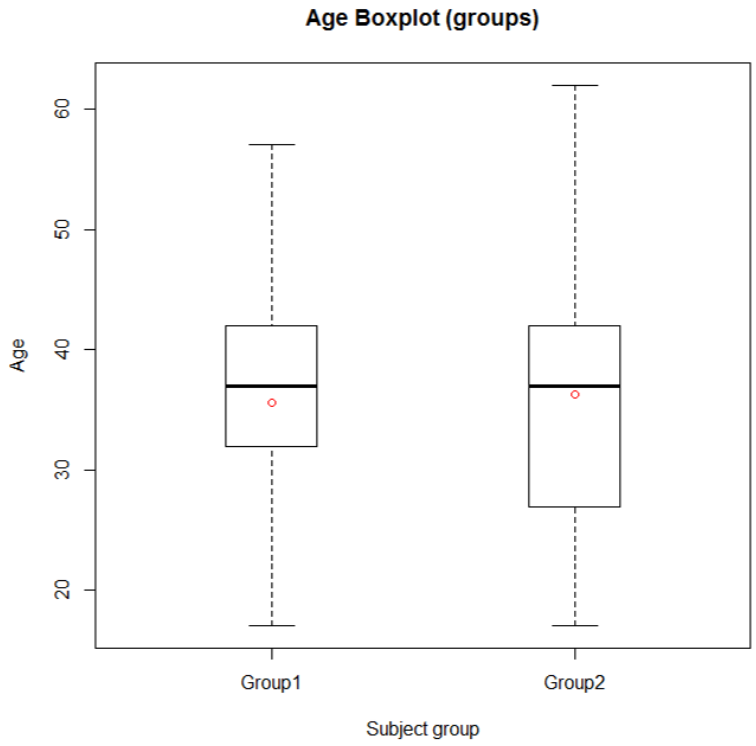

Figure 4.1: Boxplot of the age first (left) and second (right) subject groups

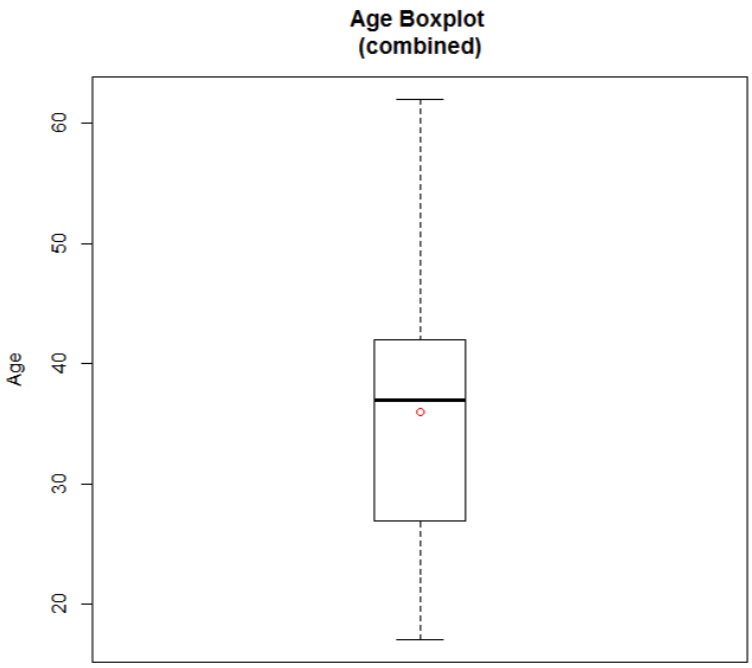

Figure 4.2: Boxplot of the age of both subject groups combined

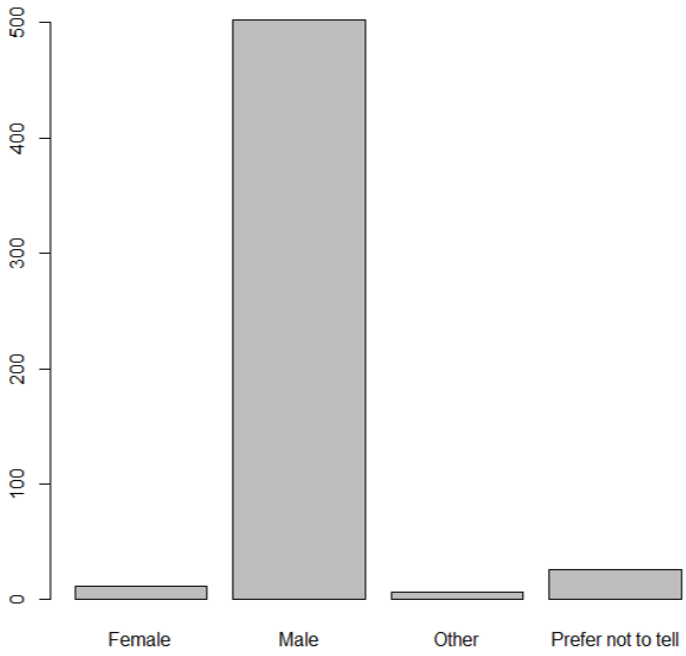

Figure 4.3: Gender of both subject groups combined

Table 4.1: Subjects' roles in projects

Roles in projects	Developer	Team Manager	Project Manager	QA
Questionnaire 1	263	77	63	43
Questionnaire 2	258	72	61	41

One participant of questionnaire 1 and five in questionnaire 2 selected "Other". A total of 25 people preferred not to tell us their gender, 16 in questionnaire 1 and nine in questionnaire 2.

The roles our subjects assumed in projects are presented in Table 4.1. Most of our subjects had the role of a developer in their projects. Additionally, 28% of subjects in questionnaire 1 and 26% in questionnaire 2 were team managers and 23% in both questionnaires were project managers. In the first questionnaire, 16% and in the second 15% of the subjects worked in quality assurance. The two groups of subjects had very similar roles distribution.

Figure 4.4 shows the general programming experience and the C programming experience of both groups of subjects. Most of the subjects who answered questionnaire 1 had between eleven and 25 years of general programming experience. The subjects of questionnaire 2 had mostly between 12 and 25 years of general programming experience. Both the mean and the median of both groups lie at 20 years of general programming

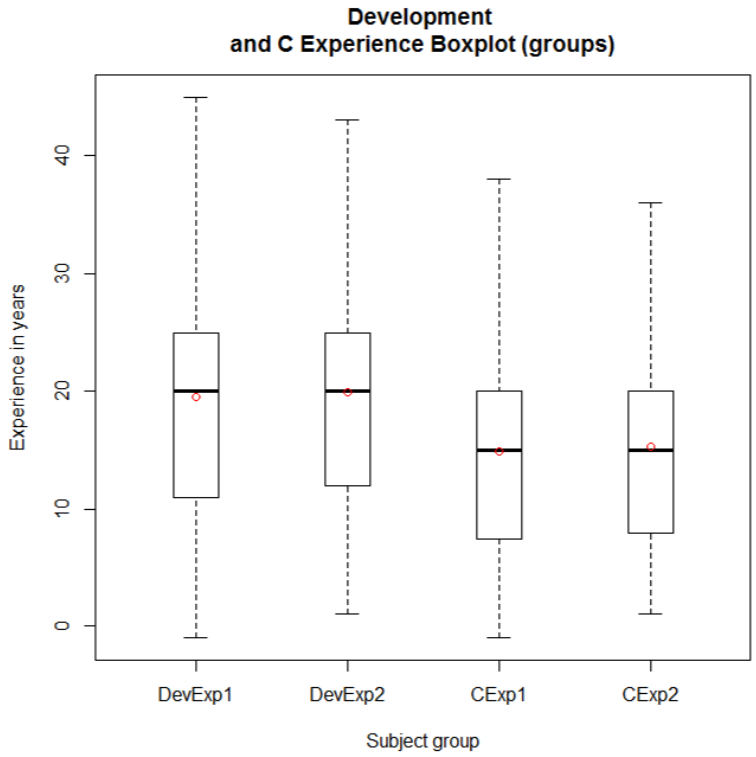

Figure 4.4: Boxplot of the general and C programming experience

experience. The first group had slightly more variety than the second. Most subjects of both groups had between eight and 20 years of C programming experience. The median and the mean of both groups are 15. The comparison of years of programming experience shows almost no differences between the two groups.

In Table 4.2 we see two logistic regression models with the dependent variables correctness in the first and second comprehension tasks. The regression models fail to establish any correlation between the subject demographics and the correctness of solved tasks.

In Table 4.3 we see the logistic regression of the first subjective assessment task. In this task, subjects rated the annotation use in the example as either appropriate or not. The independent variables used in the regression are age, gender, role in project (developer, team manager, project manager, quality assurance), experience in general programming (i.e. regardless of programming language) and C programming and personal rank of general and C programming (ranging from beginner to expert). The significant ($p < 0.05$) correlations are the role of a team manager, the experience in general programming as well as C programming. They all have a negative effect on whether a subject would rate the annotations as appropriate. Being a team manager has a negative effect of 25% on rating the annotations as appropriate. A higher experience in general programming has a negative effect of 19% and in C a negative effect of 18%. In other words, team managers and more experienced developers generally

Table 4.2: Regression of the comprehension tasks with the independent variables age, gender, roles in projects, general and C programming experience and subjective assessment of general and C programming skills

	Correctness Task 1			Correctness Task 2		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
Age	0	1	0.85	0	1	0.91
Gender	0.11	1.11	0.43	0.07	1.07	0.52
Developer	0.11	1.11	0.70	0.06	1.06	0.80
Team Mgr.	0.01	1.01	0.96	0.01	1.01	0.93
Prj. Mgr.	-0.09	0.92	0.54	-0.05	0.95	0.68
QA	0.11	1.12	0.47	0.14	1.15	0.28
Dev. Exp.	-0.01	0.99	0.57	0	1	0.92
C Exp.	0.01	1.01	0.33	0	1	0.96
Dev. Rank	0.04	1.04	0.69	0.09	1.09	0.30
C Rank	0.10	1.11	0.25	0.12	1.13	0.11
(Intercept)	0.47	1.60	0.38	-0.60	0.55	0.19

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

rated the use of annotations as less appropriate than subjects working in other roles and developers with less experience.

The negative binomial regression of the second subjective assessment task is split into two tables Table 4.4 and Table 4.5, with the criteria understanding and maintaining in the first table and extending and finding bugs in the second. The only significant correlation that could be established is that being a team manager influences your opinion on the ease of finding bugs in the example code in a negative way. With $p < 0.05$ the negative influence was 8%.

4.1.2 Baseline Comparison

One of the examples was present in both of the questionnaires in order to compare the skill set of both subject groups between each other. In this section we want to compare the demographics of our subjects by evaluating their performance while solving the baseline example vim18. This way we ensure that the results of both questionnaires are comparable with each other and that the subjects are similarly skilled.

The first group (subjects who worked on questionnaire 1) solved the first task 239 times correctly, 31 times incorrectly and chose the "I don't know" answer three times. The second group solved it 238 times correct and 27 times incorrect, with six people choosing the "I don't know" answer. The second task was solved correctly by the first group 172 times, with further 91 subjects solving it partially correct and ten subjects solving it wrong. The second group solved the second task correctly 163 times, 98 more subjects solved it partially right and ten wrong.

Table 4.3: Regression of the first subjective assessment task with the independent variables age, gender, roles in projects, general and C programming experience and subjective assessment of general and C programming skills

	Appropriateness		
	Coefficient	OR	Pr (>|z|)
Age	0.02	1.02	0.07 .
Gender	-0.14	0.87	0.17
Developer	0.30	1.35	0.19
Team Mgr.	-0.29	0.75	0.01 **
Prj. Mgr.	0.08	1.08	0.5
QA	0.03	1.03	0.82
Dev. Exp.	-0.002	1	0.86
C Exp.	-0.005	1	0.69
Dev. Rank	-0.21	0.81	0.02 *
C Rank	-0.19	0.82	0.01 *
(Intercept)	1.17	3.21	0.01 **

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

Table 4.4: Regression of the second subjective assessment task (understanding and maintaining) with the independent variables age, gender, roles in projects, general and C programming experience and subjective assessment of general and C programming skills

	Understand			Maintain		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
Age	-0.004	1.00	0.1	-0.003	1	0.23
Gender	0.04	1.04	0.19	0.02	1.03	0.44
Developer	0.06	1.06	0.39	0.03	1.03	0.67
Team Mgr.	-0.05	0.95	0.16	-0.06	0.94	0.08 .
Prj. Mgr.	0.01	1.01	0.76	-0.001	1	0.97
QA	0.01	1.01	0.72	0.002	1	0.95
Dev. Exp.	0.002	1	0.84	0.001	1	0.72
C Exp.	0.003	1	0.50	0.004	1	0.23
Dev. Rank	0.02	1.03	0.32	-0.00006	1	1.00
C Rank	-0.01	0.99	0.79	0.02	1.02	0.46
(Intercept)	0.74	2.09	1.41e-11 ***	0.83	2.29	1.87e-09 ***

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

Table 4.5: Regression of the second subjective assessment task (extending and detecting bugs) with the independent variables age, gender, roles in projects, general and C programming experience and subjective assessment of general and C programming skills

	Extend			Detect Bugs		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
Age	-0.001	1	0.63	-0.005	1	0.11
Gender	0.01	1.01	0.71	0.03	1.03	0.28
Developer	0.02	1.02	0.78	0.03	1.03	0.63
Team Mgr.	-0.05	0.95	0.18	-0.08	0.92	0.0197 *
Prj. Mgr.	-0.004	1	0.91	0.01	1.01	0.71
QA	0.01	1.01	0.72	-0.01	0.99	0.77
Dev. Exp.	0.00001	1	1.00	0.002	1	0.66
C Exp.	0.003	1	0.40	0.003	1	0.35
Dev. Rank	0.01	1.01	0.67	0.01	1.01	0.60
C Rank	0.01	1.01	0.83	0.02	1.02	0.37
(Intercept)	0.79	2.19	2.69e-08 ***	0.82	2.26	3.41e-09 ***

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

Table 4.6: Subjective question 2 answers of group 1 and 2 for vim18

	Group 1				Group 2			
	very hard	hard	easy	very easy	very hard	hard	easy	very easy
Understand	10	96	134	33	10	96	131	34
Maintain	32	123	105	13	33	115	97	26
Extend	29	129	97	18	25	128	91	27
Detect Bugs	48	149	64	12	38	137	76	20

When answering the subjective assessments, the first group found the annotation use in vim18 appropriate 126 times and inappropriate 147, while the second group found them appropriate 146 times and inappropriate 125 times.

In Table 4.6 we see the answers both groups provided for the second subjective task. The ratings of understand, maintain and detect bugs are in the same order in both groups. The highest rated categories in extend were hard and easy for both groups, the third highest for the first group was very hard and for the second group very easy. The differences in the ratings are small.

In summary, it can be said that while there are slight differences between the subject groups of the two questionnaires, they are small enough to be able to compare between them. Both groups have similar subject counts. The first task was solved correctly by 88% of each group. The second task was solved correctly by 63% of the first group and 60% of the second group. Furthermore, the subjective rating of the baseline example according to the four criteria were mostly in the same order between the groups, with the only difference being in the third and fourth category of expanding. The biggest difference is found in the question whether the annotation use is appropriate, as the

Table 4.7: Correctness task 1

	right	wrong	Don‘t know
vim18	477	58	9
vim15	267	5	1
vim15·r	252	14	5
vim13	82	142	47
vim13·r	84	162	27
emacs12	233	47	3
emacs12·r	218	47	6
emacs11	223	25	23
emacs11·r	227	31	15

first group mostly answered no and the second yes. Still the difference between the answers were only 21 votes (8% of the group size).

4.2 Answering the Research Questions

4.2.1 RQ1: Do Different Amounts of Preprocessor Use Affect Developer Effectiveness During Program Comprehension Tasks?

The first research question asked whether the different amounts of preprocessor directives affect developer effectiveness while solving program comprehension tasks. In order to answer this question, we designed two tasks that our subjects answered for five different code examples.

The first task gave a number of statements and the subjects had to choose the correct one. Only one answer per question was possible. The results are presented in Figure 4.5 and the raw numbers in Section 4.2.1.

Vim18, the baseline example, had double the amount of participants of the other examples, resulting in a higher amount of answers. Vim15 and vim15 refactored were the only original example where the subjects answered over 90% correctly. The most problematic original example was vim13 with 142 wrong and 47“I don’t know” answers. Vim13 refactored performed similarly poor as its original, with 162 wrong and 27 escape answers. Both emacs examples had a good solved rate. The refactored emacs examples also performed similarly well to their originals.

The second task asked the subject to provide a feature configuration that will trigger the execution of a specific line of code. The answers were of the form ”Feature X is defined” and ”Feature X is undefined”, as well as the possibility to select that there’s no possible configuration or that all configurations lead to the execution. Additionally an escape answer ”I don’t know” was given. The answers are presented in Figure 4.6, their numbers in Section 4.2.1. Since the escape answer wasn’t unique, i.e. it was possible

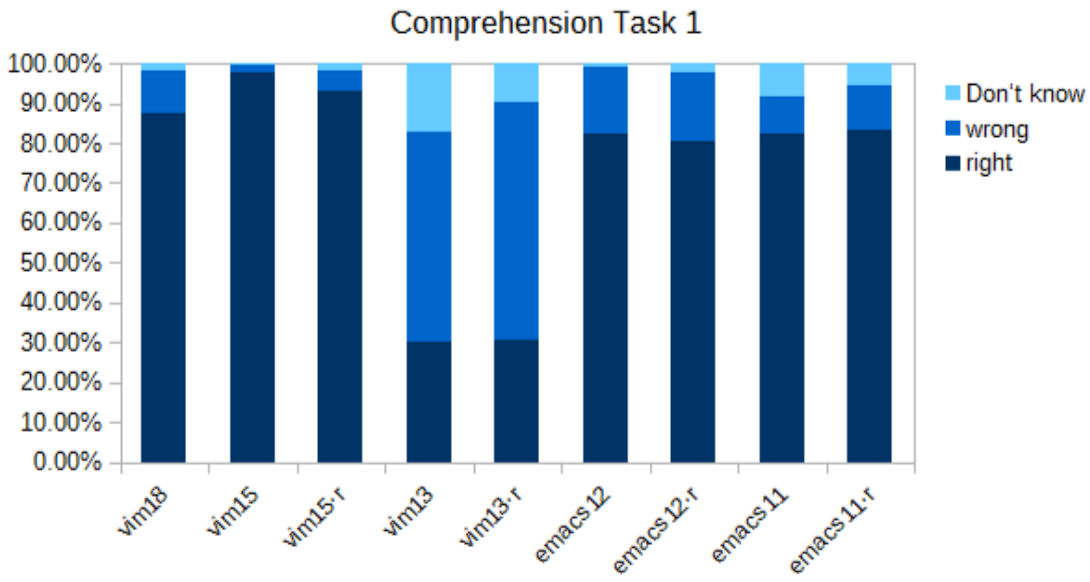

Figure 4.5: Correctness of the first comprehension task

Table 4.8: Correctness task 2

	right	partially right	wrong
vim18	300	189	20
vim15	137	127	9
vim15·r	117	143	11
vim13	49	176	46
vim13·r	53	191	29
emacs12	236	31	6
emacs12·r	210	47	14
emacs11	203	49	19
emacs11·r	180	64	29

to select it alongside with other answers, it is not shown. We differentiate between full correctness, partial correctness and wrong answers. If for example the correct answer for an example would have been features `X && Y`, then only selecting those two features (and nothing else) would have given full correctness, while selecting either one of these (regardless of other selections) would give partial correctness.

With the exception of vim13, the original examples all had at least 50% full and over 80% partial correctness. Notable are the two emacs examples with 236 (86%) right answers for emacs12 and 203 (75%) for emacs11. Vim13 had 18% right and 18% wrong answers (the rest being partially correct). The emacs examples had the highest rate of right answers. The refactored examples of vim15, emacs12 and emacs11 performed slightly worse than their original counterparts. Vim13 refactored performed slightly better than vim13. Emacs12 had the largest amount of right answers amongst the refactored examples with 210 (77%) and vim13 the smallest with 53 (18%).

In order to see which metrics influence the correctness of the first task, we performed a logistic regression analysis in R. The results are depicted in Table 4.9. The significant independent variables with a p-value of <0.1% are NOFL, NONEST, NONEG and log2loc. With a p-value of <1% the independent variable is the ratio between annotated and not annotated code, and with a p-value of <5% the NOFC metric. NOFL has a negative influence on the correctness. If NOFL increases by one, the correctness is expected to decrease by 99%. The other significant variables have a positive effect on correctness. The largest impact comes from log2log with 40280183% improvement of the correctness if log2loc increases by one, and from loacratio, where an increase by one would mean a 157777% increase in correctness. The increase in NONEST leads to an increase by 681%, NONEG to an increase of 357% and NOFC to an increase of 39%.

The logistic regression of correctness of the second task is depicted in Table 4.9 as well. Significant independent variables with p <0.1% are NOFC, NONEG and loacratio. With p <1% NOFL and with p <5% NONEST and log2loc. NOFL and NOFC have a negative effect on correctness of the second task with a decrease of 77% in correctness when NOFL increases by one and a decrease of 54% when NOFC increases by one. From the other significant variables, loacratio has the highest positive impact, with a

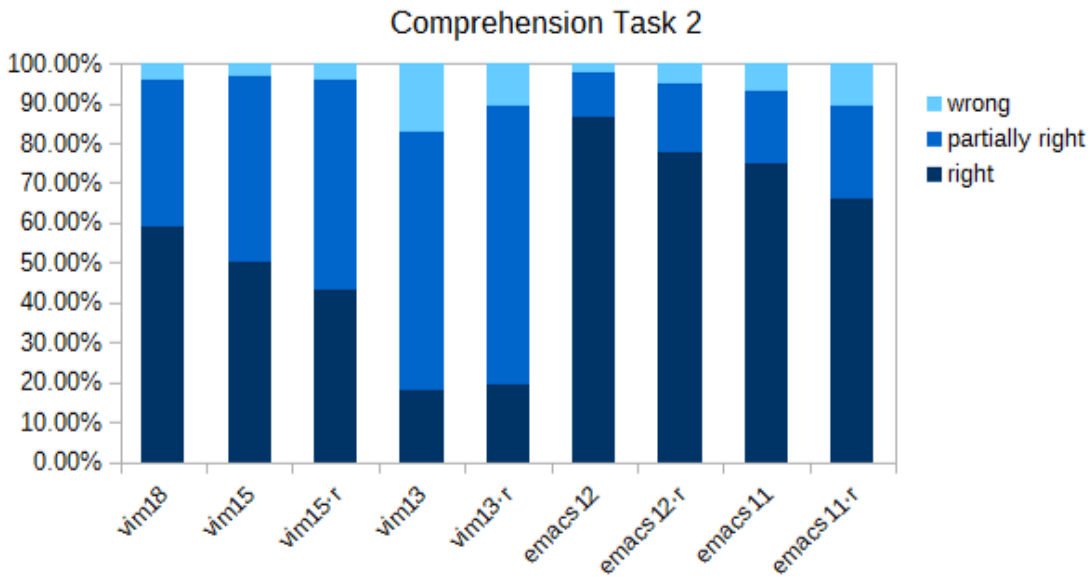

Figure 4.6: Correctness of the second comprehension task

Table 4.9: Regression of the dependent variables correctness of the comprehension task, showing coefficient, odds ratio, p-values and coded significance

	Correctness Task 1			Correctness Task 2		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
NOFL	-4.49	0.01	3.11E-10 ***	-1.46	0.23	0.00 **
NOFC	0.33	1.39	4.87E-02 *	-0.77	0.46	3.59E-07 ***
NONEST	2.05	7.81	2.44E-09 ***	0.69	1.99	0.01 *
NONEG	1.52	4.57	6.75E-05 ***	1.39	4.02	2.55E-05 ***
loacratio	7.36	1578.77	7.62E-03 **	9.69	16140.71	0.0002 ***
log2loc	12.91	402801.84	1.31E-08 ***	3.73	41.51	0.02 *
functions	0.82	2.27	0.44 **	1.84	6.28	0.10
(Intercept)	-55.9	0	8.86E-08 ***	-20.54	0	0.01 *

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

Table 4.10: Subjective assessment question 1

increase of correctness by 16114000%. Log2loc effects an increase by 450%, NONEG by 302% and NONEST by 99%.

We conclude that, while the regression analysis of the two comprehension tasks indicates that variability-aware metrics affect correctness when solving comprehension tasks, the high numbers of their influence sound implausible. Therefore the data we collected does not provide a good base for establishing correlations.

4.2.2 RQ2: Do Metrics of Preprocessor Use Reflect Subjective Assessments of Code Quality?

The second research question asks whether the metrics of preprocessor use reflect the subjective assessments of code quality and understanding of the subjects. We asked two subjective assessment questions to answer this question.

The first subjective assessment question was whether the subject considers the annotation use in the example appropriate. The results are shown in Figure 4.7 and Section 4.2.2.

For vim18 half the subjects found the annotation to be appropriate, while the other half found them inappropriate. The other two original vim examples had a lower acceptance rate for annotation use. In vim13 only 25% found the annotations appropriate. In the refactored vim examples, the subjects rated the annotations as being more appropriate. In the original emacs examples, the acceptance rate goes up, with emacs12 having 68% subjects in favour of the annotations and in emacs11 70%. The highest rating annotation use received was in the refactored emacs12 example with 82% who found the annotation use appropriate. Emacs11 refactored was interesting insofar that it was the only refactored example that was rated slightly worse than its original.

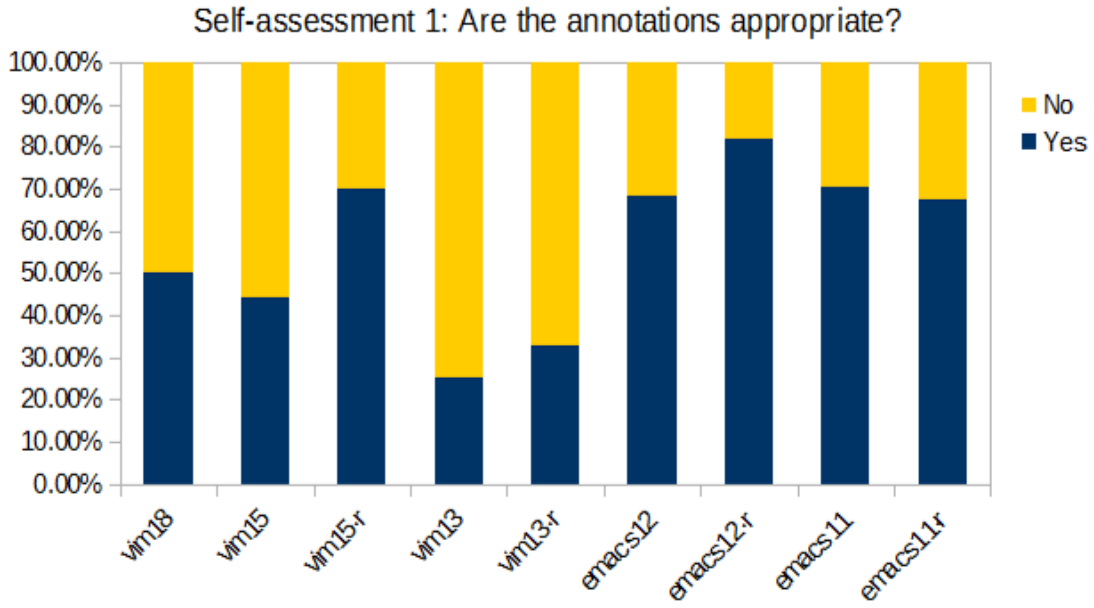

Figure 4.7: Appropriateness of the preprocessor use in percent

Table 4.11: Regression appropriateness of annotation use

	Appropriateness		
	Coefficient	OR	Pr (>|z|)
NOFL	-0.33	0.72	0.43
NOFC	-0.03	0.97	0.80
NONEST	-0.07	0.93	0.75
NONEG	0.36	1.44	0.20
loacratio	-3.21	0.04	0.15
log2loc	1.08	2.94	0.44
functions	-1.66	0.19	0.08 .
(Intercept)	1.06	2.89	0.88

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

To evaluate the significance of variability-aware metrics, we performed a logistic regression on the results of the first subjective assessment. The regression model failed to establish a strong correlation between the metrics and whether the subject would rate the annotations as appropriate or not. A slight correlation, with p <10%, can be seen with the number of functions. While this could mean that participants are 81% less likely to find the use of annotations appropriate when only one function is defined, this correlation is not significant enough to say this with certainty.

The second subjective assessment question was to rate the code fragment regarding four criteria: the ease of understanding, maintaining, extending and finding bugs in the code. The possible answers were *very hard*, *hard*, *easy* and *very easy*. The results of the examples are presented in Figure 4.8, Figure 4.9, Figure 4.10 and Figure 4.11. The numbers are presented in Section 4.2.2.

The perception of how easy understanding the code was (see Figure 4.8), was mixed between *hard* and *easy* for both vim18 and vim15 original examples. For vim18 most answers leaned towards the code being *easy*, and for vim15 *hard*. Vim13 showed a trend to being harder to understand with most subjects rating it *very hard*, and *hard*. Both emacs examples showed a trend to being easier to understand. Emacs12 was rated mostly *easy* and *very easy*. Emacs11 was mixed between *easy*, *hard* (67) and *very easy* (49).

For the ease of code maintenance (see Figure 4.9), vim18's trend shifted towards being hard to understand with 56% rating it either *hard* (44%) or *very hard* (12%). Still, 37% rated vim18 *easy* to understand. Vim13's trends stay the same, although the amount of subjects rating it *very hard* increases by 18% compared to understanding.

Extending the code (see Figure 4.10) of the original vim18 example was described as a mix of *hard* (257) and *easy* (188), followed by *very hard* (54) and *very easy* (45). Vim15 has been rated as *easy* to maintain by 112 subjects, *hard* by 97, *very hard* by 50 and *very easy* by 14. Vim13 was mostly rated *very hard* (138) and *hard* (94), followed by *easy* (31) and *very easy* (eight). For emacs12 the extension ease goes up with 138

Table 4.12: Subjective assessment question 2 (original examples)

		very hard	hard	easy	very easy
Understand	vim18	20	192	265	67
	vim15	19	128	109	17
	vim13	124	109	34	4
	emacs12	2	21	157	93
	emacs11	11	67	144	49
Maintain	vim18	65	238	202	39
	vim15	47	121	95	10
	vim13	149	96	19	7
	emacs12	3	31	150	89
	emacs11	15	69	144	43
Extend	vim18	54	257	188	45
	vim15	50	97	112	14
	vim13	138	94	31	8
	emacs12	6	44	138	85
	emacs11	21	64	139	47
Detect Bugs	vim18	86	286	140	32
	vim15	54	139	73	7
	vim13	152	92	21	6
	emacs12	8	49	141	75
	emacs11	20	89	121	41

subjects having rated it *easy*, 75 *very easy*, 44 *hard* and six *very hard*. Emacs11 was rated *easy* 139 times, *hard* 64 times, *very easy* 47 times and *very hard* 21 times.

The bug detection (see Figure 4.11) in vim18 was rated to be harder than the other criteria, with 68% rating it *hard* (53%) or *very hard* (15%). The amount of subjects rating it *very easy* was half of that of understanding. Vim15 was rated mixed with *hard* being the largest category with 51%, with *easy* following with 27%. Vim13 was highest rated *very hard* and *hard*. Emacs12 was rated comparably to the extending criteria. Emacs11 shows an increase in *hard* answers in favour of *easy* but otherwise keeps its trends.

The refactored vim15 example was perceived mixed, leaning towards *easy*, then *hard*. Like its original, vim13 refactored showed a trend to being hard to understand, although in the refactored version the *hard* rating was higher voted than *very hard*. Emacs12 refactored was rated overall positively with most votes in *very easy* and *easy*. Finally, emacs11 was mixed rated but with a very high (53%) *easy* amount, followed by *hard* (28%).

The ease of performing maintenance and the ease of extending the code were rated slightly worse than understanding. However, the trends from understanding are largely still present. The only bigger difference is that vim13's largest category for maintaining is *very hard* and the second largest *hard*.

Table 4.13: Subjective assessment question 2 (refactored examples)

		very hard	hard	easy	very easy
Understand	vim15·r	20	96	130	25
	vim13·r	103	121	44	5
	emacs12·r	6	16	110	139
	emacs11·r	9	76	146	42
Maintain	vim15·r	36	88	123	24
	vim13·r	132	105	33	3
	emacs12·r	6	23	119	123
	emacs11·r	18	87	131	37
Extend	vim15·r	27	93	124	27
	vim13·r	131	89	48	5
	emacs12·r	8	27	114	122
	emacs11·r	20	87	130	36
Detect Bugs	vim15·r	43	115	92	21
	vim13·r	148	102	18	5
	emacs12·r	9	31	118	113
	emacs11·r	29	96	116	32

Vim15 refactored was mixed between *hard* and *easy* for bug detecting, leaning towards *hard*. The amount of subjects rating vim15 *very hard* doubled for the ease of bug detecting compared to understanding. Vim13 has an even larger trend towards being hard to work with, with only 8% rating it either *easy* or *very easy*. Both emacs examples keep their trends from previously rated criteria.

The negative binomial regression models for understanding and maintaining are shown in Table 4.14. The regression for understanding failed to detect any significant independent variables. The regression for maintaining only shows a significant correlation with a p-value of <0.1 for NOFC. It would mean an increase in maintaining difficulty by 7%, but the significance is not high enough to draw this conclusion.

The negative binomial regression for extending the example code and finding bugs within it are depicted in Table 4.15. Extending the code is significantly ($p < 0.05$) influenced by the NOFC metric. The ease of extending the code is expected to sink by 11% with each increase of NOFC by one. There is a slightly significant ($p < 0.1$) positive correlation with the NONEG metric with an increase of 19%, which we dismiss due to the low p value. The regression analysis for the outcome "ease of finding bugs" failed to find a correlation between the dependent and independent variables.

Through regression analysis we found that metrics largely have no effect on subjective assessments of code quality on our examples.

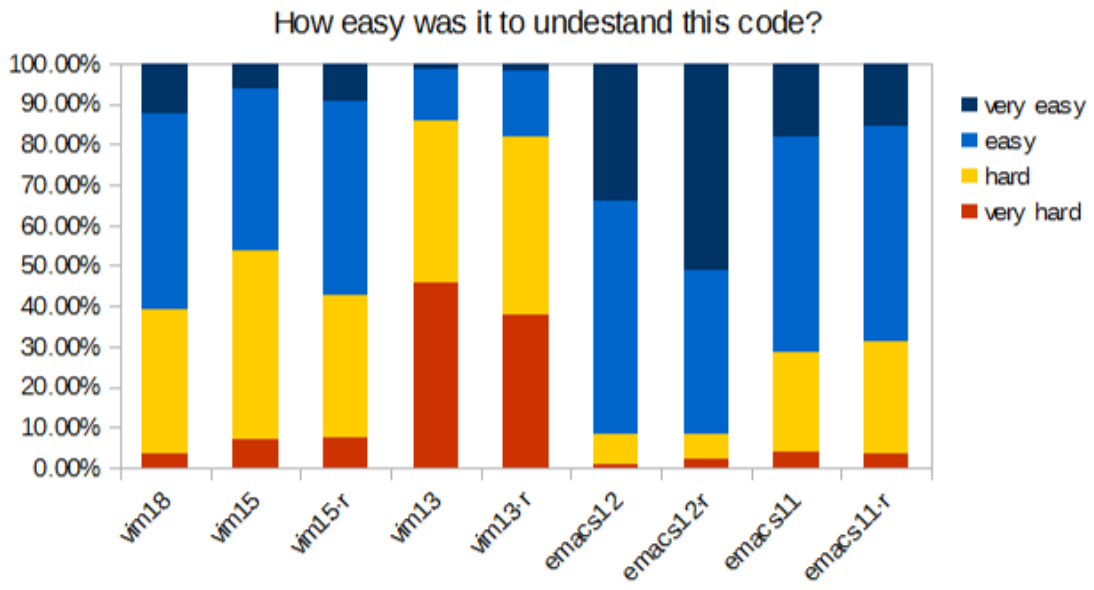

Figure 4.8: Ease of understanding the example code

Table 4.14: Regressions of criteria understanding and maintaining the example code

	Understand			Maintain		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
NOFL	-0.14	0.87	0.24	-0.13	0.88	0.28
NOFC	-0.05	0.95	0.22	-0.07	0.93	0.10 .
NONEST	0.03	1.03	0.69	0.03	1.03	0.66
NONEG	0.08	1.08	0.35	0.13	1.13	0.16
loacratio	0.28	1.32	0.70	0.43	1.54	0.56
log2loc	0.25	1.29	0.51	0.22	1.24	0.58
functions	-0.12	0.89	0.71	-0.03	0.97	0.91
(Intercept)	0.44	1.55	0.83	0.32	1.38	0.88

Signif. codes: ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1

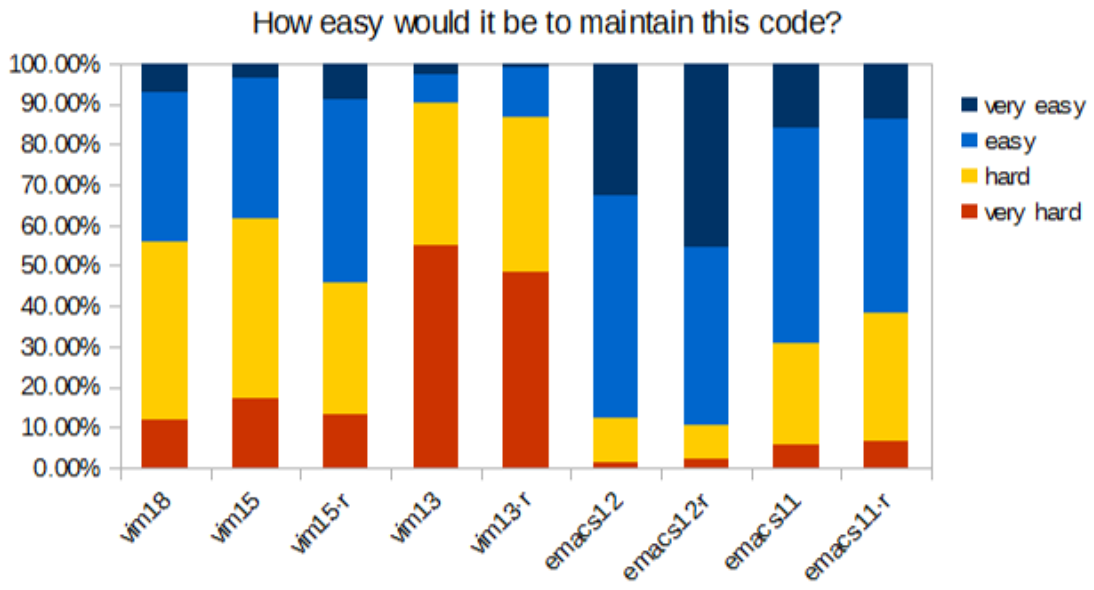

Figure 4.9: Ease of maintaining the example code

Table 4.15: Regressions of criteria extending and finding bugs in the example code

	Extend			Detect Bugs		
	Coefficient	OR	Pr (>|z|)	Coefficient	OR	Pr (>|z|)
NOFL	-0.17	0.85	0.18	-0.12	0.89	0.33
NOFC	-0.12	0.89	0.01 *	-0.07	0.94	0.14
NONEST	0.08	1.08	0.31	0.03	1.03	0.63
NONEG	0.18	1.19	0.05 .	0.11	1.12	0.21
loacratio	0.94	2.55	0.22	0.34	1.40	0.64
log2loc	0.29	1.34	0.47	0.17	1.19	0.66
functions	0.19	1.21	0.57	-0.03	0.97	0.93
(Intercept)	-0.58	0.56	0.78	0.52	1.68	0.80

Signif. codes: '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1

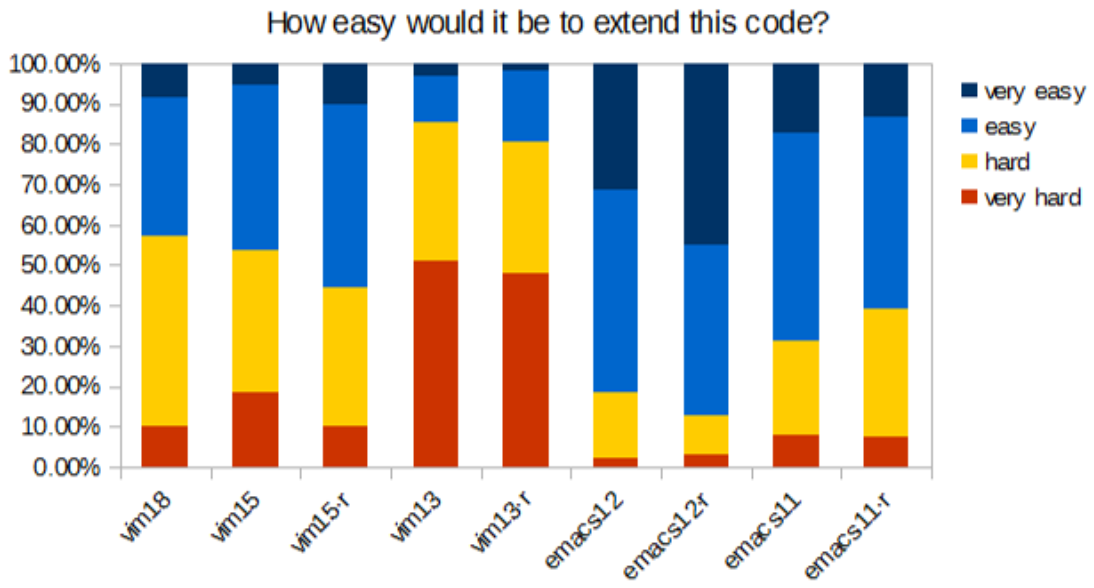

Figure 4.10: Ease of extending the example code

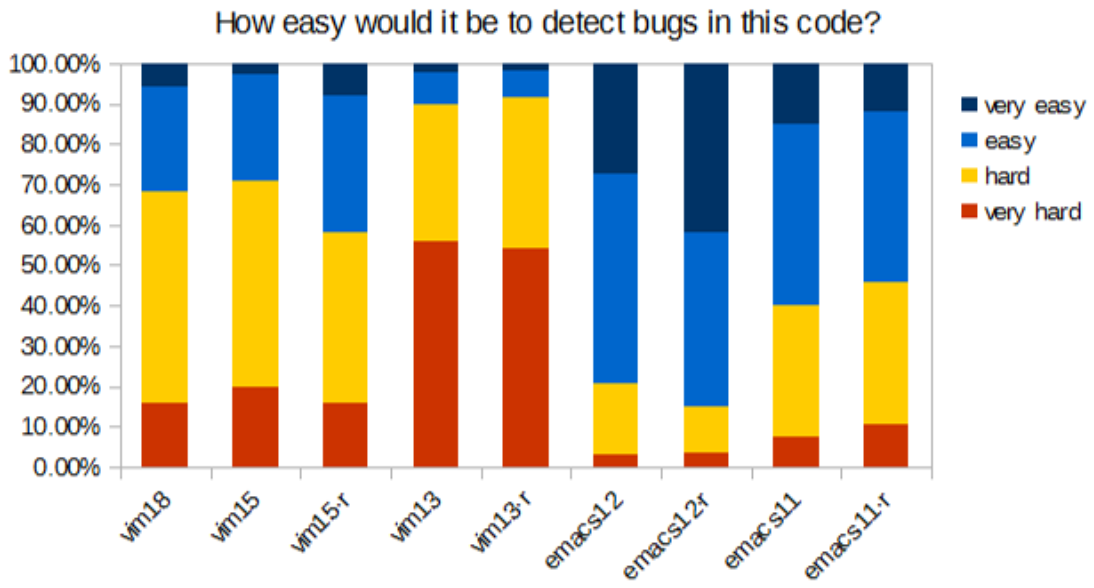

Figure 4.11: Ease of finding bugs in the example code

4.2.3 RQ3: Which Reasons Do Developers Mention for Poor Program Comprehension?

The third research question aimed to provide further insight into why the subjects found the code hard to understand. In order to answer this questions qualitative answers were evaluated, categorised and their mood was assessed.

Figure 4.12 shows the category distribution between the examples. Vim18's largest category was understanding with 121 votes, closely followed by code quality with 110 votes, while complexity received 56. The highest reasons stated for vim15 were complexity (87) and understanding (70), with code quality being a distant third (19). For vim13 understanding was given as the reason 113 times, complexity 87 times and code quality 32 times. Emacs12's highest reason was code quality with 45 votes, followed by complexity with 27 and understanding with 17. Emacs11 was rated complex 57 times, 21 times hard to understand and five times having poor code quality. In the refactored examples, vim15's reasons were mixed, with 36 votes in complexity, 29 in understanding and 16 in code quality. Vim13 had 104 votes for code quality, 66 for understanding and 25 for complexity. Emacs12's highest rated was complexity with 38 votes, followed by code quality (ten) and understanding (seven). Last but not least, the refactored emacs11 example was rated by 59 subjects to have poor code quality, 25 subjects found it hard to understand and four objected its complexity.

Following are representative quotes for the three categories (understanding, complexity and code quality):

> Understanding: *The style choices are bad for readability. Having a preprocessor conditionally compiled statement effectively in the middle of a compound if makes it hard to tell where the if's conditional ends.*

> Complexity: *Nested. Too largely scoped.*

> Code Quality: *The preprocessor annotation ifdef FEAT_AUTOCMD affects a fraction of an expression, which is bad practice.*

In qualitative answers the subjects stated further reasons for poor understanding. Some stated that they found the questions and the provided answers to be ambiguous insofar that they don't reflect the code accurately. Additionally there has been critique about missing context, as the definitions for global variables were not provided and more code would have been helpful for better understanding.

Often the indentations were named as being a reason for bad understanding. However, when stating this as reason, subjects would also separate indentations from the code (i.e."The code isn't hard, the indentation is weird"). It is important to note that the selected software systems, vim and emacs, use different indentation styles that were left

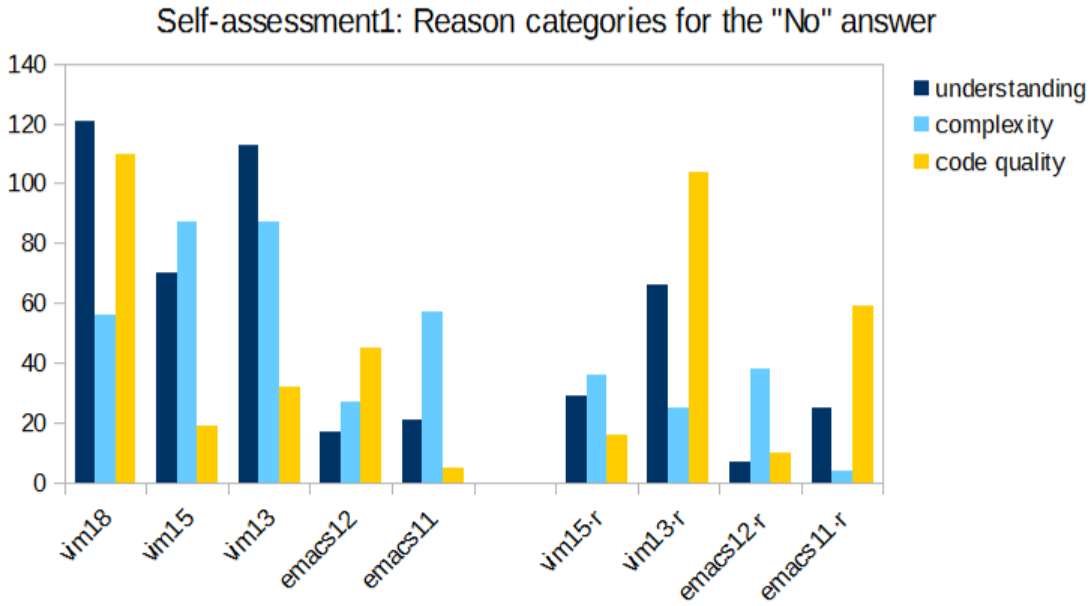

Figure 4.12: Categories of reasons provided for poor understanding of the example code

unaltered for the survey. Therefore the critique regarding indentations did not stem from code modifications from our site.

We tried to establish a general mood of the given reasons. This was done manually after reading the reasons. Each reason was assigned a positive, neutral or negative mood according to the language used and the reasoning itself. Positive mood would be present e.g. when the subject remarked that in the right context the annotation use could very well be appropriate or that only with annotations the goal of the code is achievable. Neutral mood would state that the code could be written better or that the subject would've implemented it differently. Negative mood often used stronger language and suggested that the code has really bad quality and will inevitable lead to problems. However, when evaluating the categories and moods, it is important to note that only those who rated the annotation use as inappropriate are included and that not everyone who voted "no" gave a reasoning. Following are example quotes for the three moods (positive, neutral, negative).

> Positive: *I like to indent preprocessor annotations if they're nested. I also don't like to separate # and the command by spaces, but that's not really inappropriate.*
>
> Neutral: *Seems convoluted, can be written easier*
>
> Negative: *Using #ifdef inside of functions is bad enough, but using it inside of expressions as is the case here is very bad style.*

Figure 4.13 shows the mood of the "no" answers. Vim18's reasons were negative 144 times, 98 times neutral and 16 times positive. Vim15's were positive seven times, neutral 30 times and negative 112 times. Vim13 was positive five times, 27 times neutral and 168 times negative. Emacs12 was rated positive six times, 21 neutral and 54 negative. Emacs11 was rated positive five times, 33 neutral and 38 negative. For the refactored examples, vim15 had four positive, 29 neutral and 48 negative reasons. Vim13 had three negative, 21 neutral and 25 negative answers. Emacs12 was rated positive two times, neutral 21 and negative 25. Emacs11 was rated positive two times, 20 times neutral and 58 times negative.

We asked the subjects to tell us which lines they found hardest to understand in the examples. The full table is located in **??**. In Listing 4.1, Listing 4.2, Listing 4.3, Listing 4.4, Listing 4.5, Listing 4.6, Listing 4.7, Listing 4.8 and Listing 4.9 we show the extracts from our examples containing the hardest lines only. It can be generally noted that, with the exception of emacs11 original, all extracts contain complex `#ifdefs`. Often subjects found structures (e.g. `ifs`) harder to understand if they were interrupted by `#ifdefs`. Most refactored examples show the same code extracts as their originals. A notable difference is found in vim13 and vim13 refactored, where additionally to the lines that were hard to understand in the original example, a second problematic code segment was added in the refactored version.

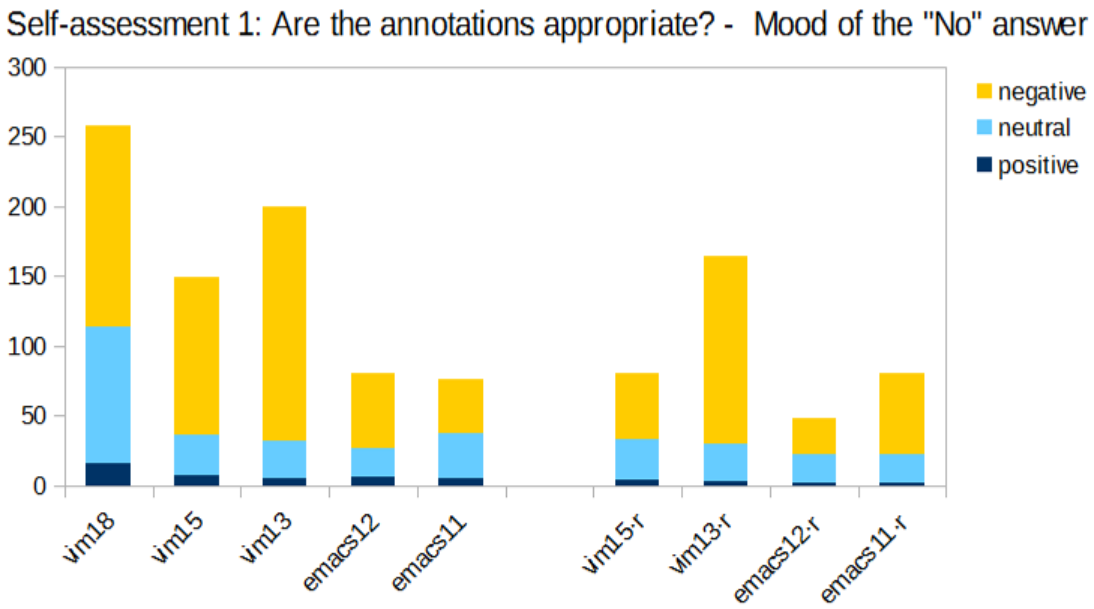

Figure 4.13: Mood of reasons provided for poor understanding of the example code

Listing 4.2: Hardest lines vim15 original

```
8       if (!vim_isAbsName(fname)
9               || strstr((char *)fname, "..") != NULL
10              || strstr((char *)fname, "//") != NULL
11 # ifdef BACKSLASH_IN_FILENAME
12              || strstr((char *)fname, "\\\\") != NULL
13 # endif
14 # if defined(MSWIN) || defined(DJGPP)
15              || vim_strchr(fname, '~') != NULL
16 # endif
17              )
...
23 #  ifdef USE_LONG_FNAME
24      if (USE_LONG_FNAME)
25 #  endif
```

Listing 4.1: Hardest lines vim18

```
5       if (p_imsf[0] != NUL)
...
9           if (exiting
10 #    ifdef FEAT_AUTOCMD
11                  || is_autocmd_blocked()
12 #    endif
13                   )
14              return FALSE;
...
16          is_active = call_func_retnr(p_imsf, 0, NULL, FALSE);
```

RQ3 asked which reasons developers provided for poor program comprehension. The reasons are that the code examples are hard to read due to the use of annotations and have a high complexity due to the nesting of annotations. The subjects considered annotations that break control flow hard to understand. Additionally the code quality was named to be generally bad in the examples, with design flaws and problematic indentations.

4.3 Discussion

4.3.1 Demographics Discussion

The regression failed to show significant correlations between the differences in demographics (age, gender, programming experience, roles in projects, subjective ranking of own programming experience) and the correctness in solving the comprehension tasks. We conclude that the demographics have little to no influence to understanding the code and working on it.

Listing 4.3: Hardest lines vim15 refactored

```
16      int is_rel_name = !vim_isAbsName(fname)
17                              || strstr((char *)fname, "..") != NULL
18                              || strstr((char *)fname, "//") != NULL
    ;
19 # ifdef BACKSLASH_IN_FILENAME
20      is_rel_name = is_rel_name || strstr((char *)fname, "\\\\")
    != NULL;
21 # endif
22 # if defined(MSWIN) || defined(DJGPP)
23      is_rel_name = is_rel_name || vim_strchr(fname, '~') != NULL;
24 # endif
...
32 #  if !defined(USE_LONG_FNAME) || USE_LONG_FNAME
```

Listing 4.4: Hardest lines vim13 original

```
13 #if defined(FEAT_AUTOCMD) || defined(FEAT_CONCEAL)
14          if (ready && (
15 # ifdef FEAT_AUTOCMD
16                      has_cursormovedI()
17 # endif
18 # if defined(FEAT_AUTOCMD) && defined(FEAT_CONCEAL)
19                      ||
20 # endif
21 # ifdef FEAT_CONCEAL
22                      curwin->w_p_cole > 0
23 # endif
24                      )
25              && !equalpos(last_cursormoved, curwin->w_cursor)
26 # ifdef FEAT_INS_EXPAND
27              && !pum_visible()
28 # endif
29             )
```

Listing 4.5: Hardest lines vim13 refactored

```
16              int need_update = 0;
17 # ifdef FEAT_AUTOCMD
18              need_update = need_update || has_cursormovedI();
19 # endif
20 # ifdef FEAT_CONCEAL
21              need_update = need_update || (curwin->w_p_cole > 0);
22 # endif
23              need_update = need_update && !equalpos(
   last_cursormoved, curwin->w_cursor);
24 # ifdef FEAT_INS_EXPAND
25              need_update = need_update && !pum_visible();
26 # endif
...
67          if ((conceal_update_lines
68                  && (conceal_old_cursor_line !=
   conceal_new_cursor_line
69                      || conceal_cursor_line(curwin)))
70                  || need_cursor_line_redraw)
```

Listing 4.6: Hardest lines emacs12 original

```
4 #if C_CTYPE_CONSECUTIVE_DIGITS \
5     && C_CTYPE_CONSECUTIVE_UPPERCASE &&
   C_CTYPE_CONSECUTIVE_LOWERCASE
6 #if C_CTYPE_ASCII
7   return ((c >= '0' && c <= '9')
8           || ((c & ~0x20) >= 'A' && (c & ~0x20) <= 'F'));
```

Listing 4.7: Hardest lines emacs12 refactored

```
1 #if C_CTYPE_CONSECUTIVE_DIGITS \
2     && C_CTYPE_CONSECUTIVE_UPPERCASE &&
   C_CTYPE_CONSECUTIVE_LOWERCASE
...
8   return ((c >= '0' && c <= '9')
9           || ((c & ~0x20) >= 'A' && (c & ~0x20) <= 'F'));
```

Listing 4.8: Hardest lines emacs11 original

```
13        for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
14             got_one >= 0;
15             got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
16          count++;
...
21        for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
22             got_one > 0;
23             got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
24          count++;
25        if (got_one < 0)
26          return -1;
```

Listing 4.9: Hardest lines emacs11 refactored

```
11 # if HAVE_ACL_TYPE_EXTENDED /* Mac OS X */
12        for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
13             got_one >= 0;
14             got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
15          count++;
16 # else /* Linux, FreeBSD */
17        for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
18             got_one > 0;
19             got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
20          count++;
21        if (got_one < 0)
22          return -1;
23 # endif
```

The subjective assessments of use of annotations had correlations with age, being a team manager and both programming and C experience. It is possible that older subjects are more used to the way variability was implemented in the shown examples, as new tools and ways weren't always available to them. Team managers could be more experienced with certain programming standards and more interested in clean code that new team members can understand more easily. The same could be said about programming and C experiences, since developers with more experience are more likely to have seen more projects and code and therefore have a better idea how the functionality of the code could be implemented with a cleaner solution.

The second subjective assessment only had a correlation between maintaining the code and being a team manager. It is possible that team managers have a better overview over the time and ease of maintaining code, since they have a reference on how long it would take the team members to perform a maintenance task.

4.3.2 Notable Examples

When choosing the original examples, we tried to include each of the three smelliness ranks (R-1: minor smell, R0: medium smell, and R1: strong smell) at least once. Vim13 was the example with smelliness rank R1. The refactored example of vim13 still has the second highest ABSmell. As can be seen in the results, our subjects had the highest number of problems with both vim13 and vim13 refactored. With 30% each, the amount of subjects who solved the first task correctly was less than half of that of next lowest solved (80%). In the second task, less than 20% of the subjects answered with full correctness in both original and refactored vim13 examples. Notably the partial correctness is high in both examples (63%-70%). This suggests that while the subjects didn't fully understand the functionality of the code, they were able to correctly identify a part of it. This suggests that working with stronger smelling code requires a higher time investment. When the developer doesn't fully understand the code he is working with, bugs and design flaws can be easier to introduce and harder to fix.

When evaluating the subjective assessments of vim13 and vim13 refactored, one can see that the annotation use in these examples is far less accepted than in the other examples. There is a considerate increase of negative comments. However, the reason categories in the original and refactored vim13 example are different. When in the original example, subjects would state the impact on understanding as the biggest reason and complexity as the second biggest, with code quality being a distant third, in the refactored code quality was named the biggest reason, understanding second and complexity third. It seems that refactoring the example, while not actually having any direct effect on the task solving, changed the perception of the root of the problems.

Vim15 and vim15 refactored had the highest amount of correct solutions for the first task. This is interesting because of two reasons. First, the difference in the metrics of the original and refactored example is fairly high. For example, vim15 refactored is closer to the ABSmell of emacs11 refactored than vim15 original. Second, vim15's metrics are rather similar compared to the other examples. They're usually neither

the highest nor lowest, with the exception of the NOFC metric, where the original is highest and the refactored second highest. A possible explanation could be that because vim15 was the second example in the questionnaire, the performance of the subjects was higher due to a higher motivation and having had trained on vim18.

In the second task, however, both vim15 examples perform worse. It is possible that the question was not formulated clear enough and the subjects had to get used to it. The subjective assessments of vim15 and vim 15 refactored were mixed. This means that the subjects did not consider vim15 particularly easy to understand or work with, despite performing so well in the first task.

All four emacs examples performed very good in both understanding tasks, with emacs12 being slightly better than emacs11. The second task was solved better in the emacs examples than in the vim examples. This is reflected in the subjective assessment questions as well, with a high amount of subjects stating that the annotations in these four examples are appropriate and understanding and maintaining the code would be mostly easy for emacs12 and emacs12 refactored. Emacs11 is mixed between ”easy” and ”hard”, with ”easy” always being the highest rated option. The reason emacs12 is rated better than emacs11 in the rating scales is likely because emacs12 is shorter. The ABSmell metric would suggest that emacs12 refactored performed best, followed by emacs11 refactored, with emacs12 and emacs11 following with a perceivable distance. This is not the case, the original examples performed slightly better than the refactored ones. One possible explanation for this is that we altered some code characteristics beyond the discipline annotations, negatively affecting code comprehension. Especially splitting one heavily annotated function into two less annotated could have had a negative effect, leading to a worse performance of our subjects.

4.3.3 Reasons and Mood

The first category of reasons found for poor annotation use was that they negatively affect understanding. Examples that have been refactored through us were generally rated more positive than the originals. This suggests that the discipline of annotations helps perceiving the code easy to understand.

The complexity of annotations were often linked to nesting of annotations. The examples in which the use of annotations was perceived as most complex were vim15 and vim13, with emacs11 and vim18 being in the second place. When combining this information with the hardest lines of the examples, we can see that the annotations used in vim15 and vim13 were used to combine conditions inside `if`s. Although disciplining the annotations in such `if`s helps, a different design approach could be beneficial for reducing perceived complexity.

Code quality was another reason the subjects rated the annotation as inappropriate. However, the subjects often would separate their concerns about code quality and annotation use and view them as different things. The reason for this could be that faulty indentations, long functions and similar problems in the code are seen as their own

problems, while annotations are seen as tools to reach a certain goal. So while annotations make it easier to introduce quality problems into the code, they aren't seen as being at fault.

Of the subjects who saw the annotation use as not appropriate, most reasons given were formulated negatively, with the lowest being 50% in emacs11 original and highest 84% in vim13 original. It is probable that subjects who chose to critique the annotations inherently chose a negative tone. The neutral reasons are more interesting, as they usually stated the subjects preferences in coding or code design and indicated that while the subject himself wouldn't write the code like this, annotation use isn't seen as critical by him. The least represented category were the positive comments, coming from subjects who acknowledged that while the annotation use isn't ideal, it serves a certain purpose that is likely harder or impossible to reach without them.

The hardest lines of the examples were similar between the original and refactored examples. They often contained complex `#ifdefs` that interrupted the control flow of the code. Although the refactored examples had fewer subjects rating the annotations inappropriate, this suggests that disciplining annotations alone does not suffice to make complex `#ifdefs` inside bodies of code well understandable.

4.3.4 Regression Analysis

The correctness of the two comprehension tasks was dependent on most of our defined metrics. The only metric that had no influence was the number of functions. Surprisingly we found that only NOFL had a negative effect on the correctness of the first comprehension task and only NOFL and NOFC on the correctness of the second comprehension task. The other metrics (NONEST, NONEG, loacratio, log2loc and NOFC in case of the first task) had a positive effect on the correctness.

A higher number of feature locations leads to a complexer code, where the code flow is often interrupted, therefore explaining the worse correctness rate. The NOFC metric positively influencing one comprehension task and negatively the other should be further examined. NONEST having a positive effect on solving the tasks could be due to increased concentration when a higher amount of nestings is present, as code has to be evaluated more carefully, thus more attention has to be paid. The loacratio has a positive correlation with correctness as well. Code with a higher amount of annotated lines is more likely to have big blocks of continuous annotated code, as opposed to single annotated lines throughout the code. Therefore reading these bigger blocks of annotated code is easier, as it doesn't break the code flow as much. Last but not least, the log2loc ratio having a positive effect is surprising, as the example with most lines of code had by far the lowest correctness. This contradiction suggests that the amount of examples is too low to gain significant insights through regression analysis. We conclude that no clear correlation could be established.

It would have been better to choose or construct examples with a gradually increase of certain metrics as opposed to having most examples having small to moderate code

smells and only two with strong code smells. This way we couldn't track well when the correctness starts to suffer.

The first subjective assessment task (the question whether the annotation use is appropriate) had a correlation with the number of functions in the code. This is probably due to the increase of complexity that comes with both the introduction of functions and annotations. When only one is present, it is easier to justify the use. A heavily annotated function in otherwise not annotated code could be easy to excuse. However, several heavily annotated functions in a short distance from each other can be too complex to comfortably be working with.

When performing regressions for the second subjective assessment task (rating the examples regarding understanding, maintaining, extending and finding bugs), the regression reached its iteration limit. This means that no clear correlation could be established for the four criteria. Further interpretation of the regression results have to take this into account.

The correlations that were found by the regression were NOFC for the ease of maintaining and extending the code and NONEG for extending the code. The NOFC could have a negative effect because the developer has to pay more attention on the feature interactions between the different features referenced by the constants. These results seem to be consistent with Melo et al. [MBW16]. It is also probable that when extending code, several variants of the extension have to be written to accommodate all features, making it harder. NONEG having a positive effect could be due to the relative ease of handling `#ifdef...#else...` constructs as opposed to several unconnected `#ifdefs`. Since in a negation, only one feature is either present or not, it can't interact with other features and the developer perceives extending such code as easier.

4.4 Threats to Validity

In this section we discuss threats to validity of our results. Threats to internal validity affect the extent to which a result supports a specific claim. Threats to external validity expresses which results can be generalised.

4.4.1 Internal Validity

The first problem is that, despite our best effort, we could have influenced our subjects during the survey. For example, the question asked may imply that we critique the annotation use in the examples. This could lead to the subject thinking about problems that could arise, negatively influencing his answer.

Some questions were criticised as being too ambiguous, specifically the second comprehension and the first subjective assessment questions. The second comprehension task asked to provide a valid configuration to execute a certain line from the example. It is possible that wording of the question itself, as well as the provided answers, confused our subjects. The first subjective assessment asked whether the annotation use

in the example is appropriate. We worded the question purposely vague, to minimise influencing the subject. However, it is possible that we confused the subjects instead.

The questions of the survey could be too narrow. Most questions only provided quantitative answer possibilities. More qualitative answers would be beneficial to obtain more nuanced developer feedback. Additionally the second subjective assessment question (ranking the criteria understanding, maintaining, extending and detecting bugs) was not based on previously performed tasks but rather the subjects' instinct. However, we are confident that the high amount of programming experience lead to an accurate estimate.

Another problem is that, even though we tried to minimise the completion time of the survey, it still took 30 minutes to complete. This excluded a number of developers who were not willing to invest a high amount of time into completing a survey. A shorter survey with two instead of five examples and more subjects could have provided better answers.

4.4.2 External Validity

The external validity was controlled through the selection of subjects and code examples for the survey. The examples were taken from open source projects. They represent naturally occurring code smells in variable software systems that are still used in practice.

When performing the survey, we obtained an appropriate amount of responses. Since our subjects were developers experienced in C programming, we are confident in their ability to work on C code and assess the quality of previously unacquainted code. The subjects were split into two groups, so we compared their demographics and performance. The differences between the subject groups were negligible, therefore we are confident that the results of both groups are equally valid.

A problem can be that, when working on hard to understand code, developers are likely to use tools in order to improve program comprehension. We did not account for this, though we can not tell how big the influence of this aspect is.

4.5 Summary

In this chapter we presented and discussed the results of our survey. Since we had two groups of subjects, we started by comparing the two groups. The first group worked on questionnaire 1, the second one questionnaire 2

The comparison was made through two means. First we compared the demographics of the two subject groups. We performed logical and negative binomial regressions in order to establish whether age, gender, roles in projects, programming experience and subjective ranking of personal programming skills influence correctness of solving tasks and subjective assessments of annotation quality in the code. The regression analysis showed no correlation between the demographics and solving accuracy. Being a team

manager, as well as rating your development and C skills higher, had a correlation to rating the appropriateness of annotation use. Being a team manager had a negative influence of 25% on finding the annotation use appropriate. Ranking your own skills higher in general programming experience or C programming experience had a negative impact of 19% and 18% respectively. Second, we compared the answers of the baseline example, vim18, between the two groups. The questions were answered similarly by both groups. Therefore we concluded that the differences between the two groups are negligible and their results can be comfortably compared between each other.

In the next section we answered the research questions. The first research question asked whether different amounts of preprocessor annotations affect effectiveness during program comprehension tasks. We found that the correctness of solving the comprehension tasks did vary with different examples. A logistic regression analysis showed that the metric NOFL has a negative correlation with the correctness of the first comprehension task, with a 99% decrease in correctness per new feature location. The metrics NOFC, NONEST, NONEG, loacratio, log2loc had a positive correlation, with an increase of correctness with the increase of the respective metric. However, the correlation numbers were too high and therefore not believable.

The second research question asked whether variability-aware metrics have an effect of subjective assessments of code quality. In order to answer this question we performed a logistic regression on the question of appropriateness of annotation use and a negative binomial regression on the four criteria (understand, maintain, extend, detect bugs) that were subjectively assessed by our subjects. The regression faced similar issues like the two performed in the previous research question. The logistic regression failed to establish a correlation, which suggests that the metrics have no effect on how likely the subjects would rate the annotation use appropriate. The negative binomial regression only established a correlation between the metric NOFC and the ease of extending code. With each new feature constant the ease of extending code would decrease by 11%. This means that the metrics mostly have no effect on subjective perception of code quality.

The third and last research question asked which reasons the subjects would give for poor program comprehension. To answer this question, we performed open coding on the qualitative answers our subjects provided. We assigned categories and a mood to the answers, in order to quantify them. The resulting categories were understanding, complexity and code quality. Understanding most often would remark bad readability. Complexity was a reason when the nestings were criticised. Code quality was for reasons like bad indentations, bad code design and similar. The possible moods were positive, neutral and negative. Only the subjects who perceived the code or annotation use as problematic gave reasons on why the code is hard to understand. Therefore most moods given were negative, although some neutral and even positive moods were present too. Neutral comments would often state that they see a certain problem with the code, without further rating it. Positive comments usually remarked that while the subject has different preferences when writing code, the kind of annotation use presented in the

example had its place. The subjects additionally stated which lines of code they found hardest to comprehend. The hardest lines contained `#ifdefs` that broke code flow.

Threats to validity were discussed in the next section. Internal threats to validity could be ambiguous wording of questions, too narrowly asked questions and too few qualitative answer possibilities. External threats to validity could be the lack of tools to help program comprehension.

5. Related Work

In this chapter we present related work. Since this thesis is based around the themes annotation-aware variability and program comprehension, we discuss work on them.

Annotation-Based Variability

The Love/Hate Relationship with the C Preprocessor: An Interview Study[MKR+15]

In this paper interviews with 40 developers were performed and cross-validated with 202 developers from open source projects and previous studies regarding the perception of the C preprocessor. The results showed that the C preprocessor is widely used in practice to solve portability and variability problems. While the developers are aware of the problems, they see no alternatives to the preprocessor and are wary of new technologies. Preprocessor bugs are seen as easier to introduce, harder to fix and more critical than other bugs. Most developers see a problem with undisciplined annotations regarding code maintainability and comprehension.

While this paper mentions variability implementation through the C preprocessor and its impact on comprehension and maintainability, it didn't link back to variability-aware metrics. A relationship between the metrics and developer perception have not been established. Our work focuses on establishing this relationship between variability-aware metrics and subjective perception of comprehension and maintainability from developers.

Do background colors improve program comprehension in the #ifdef hell?[FKA+13]

The thought that developers don't want to abandon the C preprocessor is continued in this paper. To circumvent some of the problems C preprocessor annotations introduce, it is suggested to use background colours to highlight the directives and their code. The idea is to colour all code guarded by `#ifdefs`. Nested `#ifdefs` get their own colour. The authors performed three controlled experiments with a total of over 70

subjects. The focus of the first and third experiments was on program comprehension and the focus of the second was on how the subjects use the opportunity to switch between background colours and preprocessor directives. The first two experiments were performed on medium-sized software product lines, the third one a large-sized software product line. The experiments showed that not only do background colours help with program comprehension, they also scale from medium-sized to large-sized projects.

The paper worked with both variability and program comprehension but with a focus on improving the work with preprocessor directives. Code smell metrics weren't of importance.

How does the degree of variability affect bug finding?[MBW16]

Melo et al. design and conduct a controlled experiment in order to measure the effect of variability on debugging variable code. The variability was implemented through CPP. Three different degrees of variability were defined for the example systems. Speed and precision of bug finding tasks were measured during the experiment. The speed of finding bugs decreased linearly with each degree of variability, while the effectiveness was comparably independent from the degree of variability. Identifying the faulty configuration proved to be harder for the subjects than finding the bug in the first place.

This paper controlled the variability of annotation-based software product lines in order to research the effect of differing levels of variability on the correctness and speed of solving tasks. We expanded the effect of differing levels of variability through self-assessment questions in order to capture the subjective opinions of our subjects regarding the presented code examples.

Does the discipline of preprocessor annotations matter?: a controlled experiment[SLSA13]

The effect of the discipline of preprocessor annotations on program comprehension was measured in a controlled experiment. Disciplined annotations align with the structure of the source code, while undisciplined do not. The subjects performed tasks on code with either disciplined or undisciplined annotations. Speed and correctness of solving the tasks were measured. The results suggested that the discipline of annotations have no effect on program comprehension.

When refactoring our examples, the discipline of annotations was of great importance for us. The original examples had undisciplined annotations, which we disciplined. Additionally to performing comprehension tasks on the annotated code, like in this paper, we were interested in developers' perception of working with disciplined and undisciplined annotations.

Program Comprehension

An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension[AKGA11]

Abbes et al. performed an empirical study to establish whether certain antipatterns (which we refer to as code smells), namely Blob and Spaghetti, affect program comprehension. They designed three experiments, two for each code smell and one with both code smells at once, where 24 subjects each compare smelly code with non-smelly code. The subjects were asked to perform tasks on the given examples. The performance was measured with the NASA task load index for the effort, the performance time and the percentage of correct answers. The results showed that the presence of one code smell didn't significantly reduce the program comprehension, while the presence of both code smells had a negative impact. Therefore systems with high amounts of code smells are less likely to be understood by developers which could lead to worse maintenance and an increase in system ageing.

This study showed that while moderate amounts of code smells are unproblematic, their increase negatively impacts program comprehension. This study was performed using Java and didn't consider variability.

An empirical investigation of an object-oriented design heuristic for maintainability[DSRS03]

This paper focused on the impact code design has on the maintainability of object-oriented code, as well as establish a connection between the code design and metrics. The performed study shows that design heuristics do affect the performance of subjects. Additionally, an effect on code evolution has been established and a relationship between the used metrics and code design was found.

This study was performed on object-oriented code and it's smells (specifically the god class). It lacks the variability aspect of our work and focuses on a different code smell.

A controlled experiment investigation of an object-oriented design heuristic for maintainability[DSA+04]

The impact of design heuristics on maintainability of object-oriented code was a focus of this paper as well. A controlled experiment was performed with undergraduate students, in order to understand how a specific design heuristic (the 'god class') influences the quality of developed designs. The experiment verified that the design heuristic affects the evolution of design structures. Additionally it affects the way subjects apply the inheritance mechanism.

This work focused on the way object-oriented software systems can be expanded and maintained when the code smell God Class is present. While we asked ourself similar questions, we were more interested to ask the opinion of experienced developers on the ease of expanding and maintaining variable code.

When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells[FSMS15]

In this paper a metric-based method to detect variability-aware code smells has been proposed in order to improve code comprehension and maintenance. In order to improve understanding of code using preprocessor directives, the aggregated metric ABSmell has

been introduced. The tool SKUNK has been implemented to exercise the application of the metrics on variable code. SKUNK is then tested on five highly configurable software systems to detect problematic code.

In this thesis we make use of the findings of this paper as well as the tool implemented for it. The code examples we selected were evaluated by SKUNK to gain insight into their smelliness levels. However, the results of their study haven't be confirmed on human subjects which this thesis aims to change.

6. Conclusion

Software product lines, also known as variable software, allow to implement individualised software with lower cost and time investment. Software product lines consist of features, that define the characteristics of the software system. Software product lines can be implemented either through composition-based or annotation-based approaches. Annotation-based variability is often used in practice.

A widely used implementation of annotation-based variability is the C preprocessor. CPP annotations come in the form of directives and macros, where the code from a feature is surrounded by `#ifdefs` or other conditional compilation directives. An often critiqued problem when using directives like that is the reduced readability of the code, since they break the code flow. Especially when several `#ifdefs` are nested within each other, developers may find it difficult to understand what's going on in the code and later have difficulties maintaining the code.

So far heavily annotated C code has been evaluated by certain variability-aware metrics. Presumably the metrics allow to make predictions on how well a developer can be expected to understand the code. However, since metrics lack subject interaction, we wished to confirm that experienced developers perceive code similarly problematic regarding heavily nested annotations, number of `#ifdefs`, feature constants or negations used in the code, ratio of annotated to not annotated code and other metrics.

In order to test whether variability-aware metrics can indeed be used to predict how well a developer will understand the code, we designed an online survey and distributed it amongst developers from open source projects. We defined our research questions to reflect program comprehension through both tasks and self-estimations. RQ1 asked whether different amounts of preprocessor use would affect the effectiveness during program comprehension tasks. The first research question was measured through two comprehension tasks where the subjects answered questions about the code based on their understanding. RQ2 asked whether our metrics reflect the subjective assessments of code quality. In order to answer this research question the subjects were asked

whether the annotation use in the example was appropriate and how they would rate the ease of understanding, maintaining, extending and finding bugs in the code on a Likert scale. RQ3 asked which reasons were given by the developers regarding poor program comprehension. This was answered through open coding evaluation of qualitative questions asked during the survey.

The online survey had five examples with different levels of smelliness. The other examples were present in two versions: original and refactored. There were two different variants of the questionnaire which alternated between an original and a refactored example. The examples came from real environments, namely the vim project and the emacs project.

The survey was distributed amongst experienced developers. It was completed 544 times, 273 times with the first and 271 with the second questionnaire. We compared the two groups of subjects and found the differences between them negligible.

When answering the research questions we showed the collected data and performed logistic and negative binomial regressions. The regression analysis showed that the results are inconclusive due to the small amount of examples. The results suggested that the NOFL metric has a big negative effect on correctness. The NOFC metric had a positive effect on the first comprehension task and a negative on the second. The other variability-aware metrics had a positive effect ranging from a 39% increase of correctness per increase in the metric to 16114999%, which seems implausible. More testing with more examples should be done in order to establish whether the metrics have a definitive effect on solving program comprehension tasks.

The regression analysis of the answers of the second research question failed to establish a correlation of the metrics and how likely the subjects was to rate the annotation use appropriate. Understanding and maintaining had no significant correlation either. A subject was 11% more likely to find extending the example code harder, when the NOFC metric increased. A correlation between the metrics and the ease of detecting bugs in the code could not be established. Generally speaking, the subjective assessments of code quality did not seem to be affected by the metrics.

Reasons given for poor program comprehension were the use of annotation and general code quality. Code quality included the code design and bad indentations. Annotation were named as a reason for poor code comprehension, because they make the code less readable and more complex. The complexity was present in the form of highly nested annotations. Annotations were also problematic insofar that they broke the code flow.

Future Work

In the end of our survey we asked the subjects to provide their e-mail addresses in case they are interested in a follow-up interview. Since feedback during the duration of the survey showed that the subjects wanted to share more opinions than the provided qualitative questions allowed, conducting these interviews would be a good start to get better insight in these opinions.

Controlled experiments measuring time, correctness and perceived workload could be designed in order to get a better insight into the process of program comprehension while working with annotated code. Think-aloud protocols could complement these experiments.

Another idea would be to replicate the survey with less examples and more questionnaire variants, to cut the completion time for the subjects and making the participation more attractive. Our code examples were all very heavily annotated, code examples where the annotation severity fluctuate more would give more generalised results.

Further work on the variability-aware metrics could be a part of future work as well. Implementing code examples that work well with the introduced new metrics could be fruitful, especially in comparison with decade-old code. Writing new examples from scratch, one could ensure that they are tied to only one metric, e.g. the annotation severity but not the amount of constants etc..

Even though we contacted contributors to C- projects, we received several e-mails stating that the person doesn't work with C any longer due to it being considered harmful. Therefore a replication of the survey in other languages could be called for to get developers working in modern languages on board.

The survey design could be improved to ensure the subjects answer the questions in good faith, e.g. by introducing control questions. This would improve the quality of the answers and improve the significance of the gained insights.

A. Appendix

A.1 Code Examples

A.1.1 Vim18

Listing A.1: Full code of example vim18

```
    int
im_get_status(void)
{
#  ifdef FEAT_EVAL
    if (p_imsf[0] != NUL)
    {
        int is_active;

        if (exiting
#   ifdef FEAT_AUTOCMD
                || is_autocmd_blocked()
#   endif
                )
            return FALSE;
        ++msg_silent;
        is_active = call_func_retnr(p_imsf, 0, NULL, FALSE);
        --msg_silent;
        return (is_active > 0);
    }
#  endif
    return im_is_active;
}
```

A.1.2 Vim15 Original

Listing A.2: Full code of example vim15

```
    char_u  *
fix_fname(fname)
    char_u  *fname;
{
#ifdef UNIX
    return FullName_save(fname, TRUE);
#else
    if (!vim_isAbsName(fname)
        || strstr((char *)fname, "..") != NULL
        || strstr((char *)fname, "//") != NULL
# ifdef BACKSLASH_IN_FILENAME
        || strstr((char *)fname, "\\\\") != NULL
# endif
# if defined(MSWIN) || defined(DJGPP)
        || vim_strchr(fname, '~') != NULL
# endif
        )
    return FullName_save(fname, FALSE);

    fname = vim_strsave(fname);

# ifdef USE_FNAME_CASE
#  ifdef USE_LONG_FNAME
    if (USE_LONG_FNAME)
#  endif
    {
    if (fname != NULL)
        fname_case(fname, 0);
    }
# endif

    return fname;
#endif
}
```

A.1.3 Vim15 Refactored

Listing A.3: Full code of example vim15 refactored

```
#ifdef UNIX

        char_u  *
fix_fname(fname)
        char_u  *fname;
{
        return FullName_save(fname, TRUE);
}

#else /* !UNIX */

        char_u  *
fix_fname(fname)
        char_u  *fname;
{
        int is_rel_name = !vim_isAbsName(fname)
                        || strstr((char *)fname, "..") != NULL
                        || strstr((char *)fname, "//") != NULL;
# ifdef BACKSLASH_IN_FILENAME
        is_rel_name = is_rel_name || strstr((char *)fname, "\\\\")
            != NULL;
# endif
# if defined(MSWIN) || defined(DJGPP)
        is_rel_name = is_rel_name || vim_strchr(fname, '~') != NULL
            ;
# endif

        if (is_rel_name)
        return FullName_save(fname, FALSE);

        fname = vim_strsave(fname);

# ifdef USE_FNAME_CASE
#  if !defined(USE_LONG_FNAME) || USE_LONG_FNAME
        if (fname != NULL)
        fname_case(fname, 0);
#  endif
# endif

    return fname;
}
#endif
```

A.1.4 Vim13 Original

Listing A.4: Full code of example vim13

```
   static void
ins_redraw(ready)
    int          ready UNUSED;
{
#ifdef FEAT_CONCEAL
    linenr_T     conceal_old_cursor_line = 0;
    linenr_T     conceal_new_cursor_line = 0;
    int          conceal_update_lines = FALSE;
#endif

    if (!char_avail())
    {
#if defined(FEAT_AUTOCMD) || defined(FEAT_CONCEAL)
        if (ready && (
# ifdef FEAT_AUTOCMD
                    has_cursormovedI()
# endif
# if defined(FEAT_AUTOCMD) && defined(FEAT_CONCEAL)
                    ||
# endif
# ifdef FEAT_CONCEAL
                    curwin->w_p_cole > 0
# endif
                    )
        && !equalpos(last_cursormoved, curwin->w_cursor)
# ifdef FEAT_INS_EXPAND
        && !pum_visible()
# endif
       )
    {
# ifdef FEAT_SYN_HL
        if (syntax_present(curwin) && must_redraw)
        update_screen(0);
# endif
# ifdef FEAT_AUTOCMD
        if (has_cursormovedI())
        apply_autocmds(EVENT_CURSORMOVEDI, NULL, NULL, FALSE,
            curbuf);
# endif
# ifdef FEAT_CONCEAL
        if (curwin->w_p_cole > 0)
        {
        conceal_old_cursor_line = last_cursormoved.lnum;
        conceal_new_cursor_line = curwin->w_cursor.lnum;
```

```
            conceal_update_lines = TRUE;
            }
# endif
            last_cursormoved = curwin->w_cursor;
        }
#endif
#ifdef FEAT_AUTOCMD
        if (!ready && has_textchangedI()
            && last_changedtick != curbuf->b_changedtick
# ifdef FEAT_INS_EXPAND
            && !pum_visible()
# endif
            )
        {
            if (last_changedtick_buf == curbuf)
            apply_autocmds(EVENT_TEXTCHANGEDI, NULL, NULL, FALSE,
                curbuf);
            last_changedtick_buf = curbuf;
            last_changedtick = curbuf->b_changedtick;
        }
#endif
        if (must_redraw)
            update_screen(0);
        else if (clear_cmdline || redraw_cmdline)
            showmode();
# if defined(FEAT_CONCEAL)
        if ((conceal_update_lines
                && (conceal_old_cursor_line != conceal_new_cursor_line
                    || conceal_cursor_line(curwin)))
                || need_cursor_line_redraw)
        {
            if (conceal_old_cursor_line != conceal_new_cursor_line)
                update_single_line(curwin, conceal_old_cursor_line);
            update_single_line(curwin, conceal_new_cursor_line == 0
                            ? curwin->w_cursor.lnum :
                                conceal_new_cursor_line);
            curwin->w_valid &= ~VALID_CROW;
        }
# endif
            showruler(FALSE);
            setcursor();
            emsg_on_display = FALSE;
        }
}
```

A.1.5 Vim13 Refactored

Listing A.5: Full code of example vim13 refactored

```
    static void
ins_redraw(ready)
int             ready UNUSED;
{
#ifdef FEAT_CONCEAL
    linenr_T    conceal_old_cursor_line = 0;
    linenr_T    conceal_new_cursor_line = 0;
    int         conceal_update_lines = FALSE;
#endif

    if (!char_avail())
    {
#if defined(FEAT_AUTOCMD) || defined(FEAT_CONCEAL)
        if (ready)
        {
            int need_update = 0;
# ifdef FEAT_AUTOCMD
            need_update = need_update || has_cursormovedI();
# endif
# ifdef FEAT_CONCEAL
            need_update = need_update || (curwin->w_p_cole > 0);
# endif
            need_update = need_update && !equalpos(last_cursormoved
                , curwin->w_cursor);
# ifdef FEAT_INS_EXPAND
            need_update = need_update && !pum_visible();
# endif
            if(need_update)
            {
# ifdef FEAT_SYN_HL
                if (syntax_present(curwin) && must_redraw)
                    update_screen(0);
# endif
# ifdef FEAT_AUTOCMD
                if (has_cursormovedI())
                     apply_autocmds(EVENT_CURSORMOVEDI, NULL, NULL,
                         FALSE, curbuf);
# endif
# ifdef FEAT_CONCEAL
                if (curwin->w_p_cole > 0)
                {
                    conceal_old_cursor_line = last_cursormoved.lnum
                        ;
```

```
                    conceal_new_cursor_line = curwin->w_cursor.lnum
                        ;
                    conceal_update_lines = TRUE;
                }
# endif
                last_cursormoved = curwin->w_cursor;
            }
        }
#endif
#ifdef FEAT_AUTOCMD
        int update_last_changed = !ready && has_textchangedI() &&
           last_changedtick != curbuf->b_changedtick;
# ifdef FEAT_INS_EXPAND
        update_last_changed = update_last_changed && !pum_visible()
           ;
# endif
        if (update_last_changed)
        {
            if (last_changedtick_buf == curbuf)
                apply_autocmds(EVENT_TEXTCHANGEDI, NULL, NULL,
                    FALSE, curbuf);
            last_changedtick_buf = curbuf;
            last_changedtick = curbuf->b_changedtick;
        }
#endif
        if (must_redraw)
            update_screen(0);
        else if (clear_cmdline || redraw_cmdline)
            showmode();
# if defined(FEAT_CONCEAL)
        if ((conceal_update_lines
            && (conceal_old_cursor_line != conceal_new_cursor_line
                || conceal_cursor_line(curwin)))
            || need_cursor_line_redraw)
        {
            if (conceal_old_cursor_line != conceal_new_cursor_line)
            update_single_line(curwin, conceal_old_cursor_line);
            update_single_line(curwin, conceal_new_cursor_line == 0
                            ? curwin->w_cursor.lnum :
                                conceal_new_cursor_line);
            curwin->w_valid &= ~VALID_CROW;
        }
# endif
        showruler(FALSE);
        setcursor();
        emsg_on_display = FALSE;
    }
```

```
}
```

A.1.6 Emacs12 Original

Listing A.6: Full code of example emacs12

```
bool
c_isxdigit (int c)
{
#if C_CTYPE_CONSECUTIVE_DIGITS \
    && C_CTYPE_CONSECUTIVE_UPPERCASE &&
       C_CTYPE_CONSECUTIVE_LOWERCASE
#if C_CTYPE_ASCII
  return ((c >= '0' && c <= '9')
          || ((c & ~0x20) >= 'A' && (c & ~0x20) <= 'F'));
#else
  return ((c >= '0' && c <= '9')
          || (c >= 'A' && c <= 'F')
          || (c >= 'a' && c <= 'f'));
#endif
#else
  switch (c)
    {
    case '0': case '1': case '2': case '3': case '4': case '5':
    case '6': case '7': case '8': case '9':
    case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
      return 1;
    default:
      return 0;
    }
#endif
}
```

A.1.7 Emacs12 Refactored

Listing A.7: Full code of example emacs12 refactored

```
#if C_CTYPE_CONSECUTIVE_DIGITS \
    && C_CTYPE_CONSECUTIVE_UPPERCASE &&
       C_CTYPE_CONSECUTIVE_LOWERCASE

bool
c_isxdigit (int c)
{
#if C_CTYPE_ASCII
  return ((c >= '0' && c <= '9')
          || ((c & ~0x20) >= 'A' && (c & ~0x20) <= 'F'));
#else
  return ((c >= '0' && c <= '9')
          || (c >= 'A' && c <= 'F')
          || (c >= 'a' && c <= 'f'));
#endif
}

#else /* !(C_CTYPE_CONSECUTIVE_DIGITS \
           && C_CTYPE_CONSECUTIVE_UPPERCASE &&
              C_CTYPE_CONSECUTIVE_LOWERCASE) */

bool
c_isxdigit (int c)
{
  switch (c)
    {
    case '0': case '1': case '2': case '3': case '4': case '5':
    case '6': case '7': case '8': case '9':
    case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
      return 1;
    default:
      return 0;
    }
}

#endif
```

A.1.8 Emacs11 Original

Listing A.8: Full code of example emacs11

```
int
acl_entries (acl_t acl)
{
  int count = 0;

  if (acl != NULL)
    {
#if HAVE_ACL_FIRST_ENTRY /* Linux, FreeBSD, Mac OS X */
# if HAVE_ACL_TYPE_EXTENDED /* Mac OS X */
      acl_entry_t ace;
      int got_one;

      for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
           got_one >= 0;
           got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
        count++;
# else /* Linux, FreeBSD */
      acl_entry_t ace;
      int got_one;

      for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
           got_one > 0;
           got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
        count++;
      if (got_one < 0)
        return -1;
# endif
#else /* IRIX, Tru64 */
# if HAVE_ACL_TO_SHORT_TEXT /* IRIX */
      count = acl->acl_cnt;
# endif
# if HAVE_ACL_FREE_TEXT /* Tru64 */
      count = acl->acl_num;
# endif
#endif
    }

  return count;
}
```

A.1.9 Emacs11 Refactored

Listing A.9: Full code of example emacs11 refactored

```
#ifdef HAVE_ACL_FIRST_ENTRY /* Linux, FreeBSD, Mac OS X */
int
acl_entries (acl_t acl)
{
  int count = 0;
  acl_entry_t ace;
  int got_one;

  if (acl != NULL)
    {
# if HAVE_ACL_TYPE_EXTENDED /* Mac OS X */
      for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
           got_one >= 0;
           got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
        count++;
# else /* Linux, FreeBSD */
      for (got_one = acl_get_entry (acl, ACL_FIRST_ENTRY, &ace);
           got_one > 0;
           got_one = acl_get_entry (acl, ACL_NEXT_ENTRY, &ace))
        count++;
      if (got_one < 0)
        return -1;
# endif
    }
  return count;
}

#else /* IRIX, Tru64: !HAVE_ACL_FIRST_ENTRY */
int
acl_entries (acl_t acl)
{
  int count = 0;

  if (acl != NULL)
    {
# if HAVE_ACL_TO_SHORT_TEXT /* IRIX */
      count = acl->acl_cnt;
# endif
# if HAVE_ACL_FREE_TEXT /* Tru64 */
      count = acl->acl_num;
# endif
    }

  return count;
```

```
}
#endif
```

A.2 Hardest Lines

Line	vim18	vim15	vim15·r	vim13	vim13·r	emacs12	emacs12·r	emacs11	emacs11·r
1	8	3	3	24	33	2	29	3	10
2	6	2	1	22	33	1	23	2	4
3	5	2	9	25	37	2	8	2	4
4	13	1	10	24	32	50	6	2	3
5	37	7	12	35	41	46	6	2	4
6	12	8	1	35	41	35	6	2	3
7	12	11	2	35	41	17	12	2	3
8	12	66	2	35	41	37	16	9	3
9	98	65	2	35	41	15	49	18	4
10	121	65	2	36	41	11	11	21	4
11	137	76	3	48	46	11	11	21	36
12	120	80	11	47	48	11	11	21	57
13	82	76	12	114	71	18	11	45	61
14	57	76	11	134	71	14	12	50	56
15	22	78	4	133	71	1	8	43	50
16	36	69	61	135	76	2	6	34	47
17	20	50	58	133	79	3	8	27	58
18	10	39	60	135	78	3	6	28	63
19	11	15	48	139	78	3	4	29	59
20	12	17	59	134	78	4	2	29	48
21	28	10	48	131	78	2	2	50	54
22	5	37	49	133	78	1	2	56	49
23		49	53	131	78	1	2	50	35
24		57	46	129	78	1	2	40	7
25		45	9	126	78	1	2	41	7
26		35	13	123	78	1	2	36	7
27		35	10	124	71		2	23	6
28		37	8	123	70		2	17	16
29		35	8	115	72		2	14	7
30		33	6	89	73		2	13	7
31		8	25	83	73		2	13	5
32		7	45	82	72		2	13	5
33		8	21	82	70		2	14	5
34		2	23	81	70		2	13	7
35			20	81	70		5	10	7

Line	vim18	vim15	vim15·r	vim13	vim13·r	emacs12	emacs12·r	emacs11	emacs11·r
36			20	81	70			2	12
37			4	81	70			2	12
38			4	81	69			3	14
39			3	78	68			2	14
40			3	74	68				12
41				73	68				12
42				73	68				5
43				73	68				3
44				73	68				4
45				73	65				4
46				73	63				7
47				72	63				
48				72	64				
49				73	58				
50				66	65				
51				69	58				
52				69	58				
53				67	58				
54				68	56				
55				67	55				
56				65	55				
57				62	55				
58				61	55				
59				61	55				
60				61	55				
61				61	55				
62				61	56				
63				62	56				
64				44	55				
65				44	55				
66				44	60				
67				44	97				
68				47	99				
69				73	98				
70				71	97				
71				71	68				
72				71	65				
73				50	62				
74				49	60				
75				47	58				

Line	vim18	vim15	vim15·r	vim13	vim13·r	emacs12	emacs12·r	emacs11	emacs11·r
76				48	59				
77				47	54				
78				47	53				
79				46	37				
80				45	37				
81				30	36				
82				30	35				
83				30					
84				33					

Bibliography

[AKGA11] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE, 2011. (cited on Page 62)

[Cox58] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958. (cited on Page 11)

[dej] dejure.org. Rechtsprechung - bgh, 20.05.2009 - i zr 218/07. (cited on Page 22)

[Dil11] Don A Dillman. *Mail and Internet surveys: The tailored design method*. John Wiley & Sons, 2011. (cited on Page 9 and 10)

[DPSK07] Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 281–285. IEEE, 2007. (cited on Page 10)

[DSA+04] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004. (cited on Page 63)

[DSRS03] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003. (cited on Page 6 and 63)

[FKA+13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do Background Colors Improve Program Comprehension in the #Ifdef Hell? *Empirical Software Engineering (EMSE)*, 18(4):699–745, August 2013. (cited on Page 61)

[FN99] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999. (cited on Page 7)

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000. (cited on Page 6)

[FS15] Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '15, pages 3:3–3:10, New York, NY, USA, 2015. ACM. (cited on Page 2, 6, 7, and 8)

[FSF11] Janet Feigenspan, Norbert Siegmund, and Jana Fruth. On the role of program comprehension in embedded systems. In *Proc. Workshop Software Reengineering (WSR)*, pages 34–35, 2011. (cited on Page 8)

[FSMS15] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–180. IEEE, 2015. (cited on Page 6, 7, 13, 15, and 63)

[FSS17] Wolfram Fenske, Sandro Schulze, and Gunter Saake. How preprocessor annotations (do not) affect maintainability: A case study on change-proneness. *SIGPLAN Not.*, 52(12):77–90, October 2017. (cited on Page 2)

[Joh11] James L Johnson. *Probability and statistics for computer science*. John Wiley & Sons, 2011. (cited on Page 11)

[KATS12] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):14:1–14:39, July 2012. (cited on Page 1)

[Kaz] Anastasia Kazakova. Infographic: C/c++ facts we learned before going ahead with clion. (cited on Page 22)

[KR06] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006. (cited on Page 6)

[La] Alyson La. Language trends on github. (cited on Page 19)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. IEEE Computer Science, May 2010. (cited on Page 2, 6, and 22)

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202. ACM, 2011. (cited on Page 2)

[LWE11] Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. pages 143–150. IEEE Computer Science, September 2011. (cited on Page 2 and 6)

[MBW16] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. How does the degree of variability affect bug finding? In *Proceedings of the 38th International Conference on Software Engineering*, pages 679–690. ACM, 2016. (cited on Page 2, 57, and 62)

[MKR+15] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. *practice*, 12:40, 2015. (cited on Page 2, 13, and 61)

[MTRK14] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31, 2014. (cited on Page 8)

[OO18] Michael Oberguggenberger and Alexander Ostermann. *Analysis for Computer Scientists*. Springer, 2018. (cited on Page 11)

[SABK13] Gunter Saake Sven Apel, Don Batory, and Christian Kästner. *Feature-Oriented Software Product Lines: Concepts and Implementation*. 2013. (cited on Page 1, 5, and 6)

[SLSA13] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter?: a controlled experiment. In *ACM SIGPLAN Notices*, volume 49, pages 65–74. ACM, 2013. (cited on Page 2 and 62)

[Spe] IEEE Spectrum. Ieee top programming languages: Design, methods, and data sources. (cited on Page 19)

[Szu10] Magdalena Szumilas. Explaining odds ratios. *Journal of the Canadian academy of child and adolescent psychiatry*, 19(3):227, 2010. (cited on Page 11)

[ZKJ08] Achim Zeileis, Christian Kleiber, and Simon Jackman. Regression models for count data in r. *Journal of statistical software*, 27(8):1–25, 2008. (cited on Page 11)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 18. April 2019