Growth and Distribution of Vocabulary in Open Source Software

Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, Australia

Submitted for the degree of Bachelor of Science
(Professional Software Development) (Honours)

Allan Jones

2010

Abstract

TODO: Find a way to disambiguate here between natural language and programming language Software developers are afforded great flexibility in the terms of the language they can use to write source code, bound only by the syntactic rules which are governed by the platform they are developing for. This flexibility allows developers to expressive about what the software should do and the way in which it has been constructed. Typically, the language used is not arbitrarily selected, but rather, is a vocabulary of terms that maps to the developers understanding of the software system, which is generally shared by multiple parties.

As a software system is evolved over time, undergoing addition of new functionality and maintenance efforts, the vocabulary will also undergo change, being extended and modified to support these outcomes of evolution. This change threatens the understanding of the software held by developers, driving the need for control over the preservation of the vocabulary.

TODO: Probably need a bit more here to communicate what is lacking in the current landscape, emphasising the need for research of this nature

In this thesis, we address the problem of identifying when, in the evolution of a software system, efforts towards maintaining vocabulary are required and to which areas of the software they are best directed. We conduct a study of over 40 open source Java software systems, investigating how and when source code vocabularies are formed in the process of software evolution, the nature of their growth and change and how the terms in the vocabulary are distributed throughout the codebase. Finally, we determine the applicability of the laws of software evolution in describing the evolution of the source code vocabulary.

Dedications

Acknowledgements

Allan Jones, 2010

Declaration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Notes: Might be able to cut out the first sentence and make the second sentence the leading sentence

computer is boring...better word?

'Write code'/'writing code' – repeated phrase!

Throw some citations in here

Software systems are large, complex entities assembled by developers using source code — the fundamental building block for contemporary software — which communicates to the computer how an application should behave. Collectively, the source code that makes up even a small software system represents a substantial corpus of text, which is, in most cases, almost entirely written by humans. While there is an obligation on the developers part to ensure that the source code is written in a manner in which it can be executable by the system upon which it is designed for, developers have the liberty of using expressive language in writing their code.

Utilising this freedom, developers are able to write code that is rich in semantic information regarding not only the purpose of the application and the domain in which it operates, but also the manner in which it has been designed and implemented. Such information is crucial in facilitating the development of mental models, which allow an adequate and consistent comprehension of the software for the developers involved in its construction.

As software evolves to include new functionality for those who intend to use it, there will inherently be changes within the codebase to support this evolution, bringing about new concepts that require representation within the software and an associated vocabulary used to describe these concepts. This in turn requires the developers to adjust their mental models of the software, which can often be an expensive and timely process.

To preserve the important elements of the mental models formed of software systems and ease the adaptation to include new information, changes in vocabulary can be captured within representative vocabulary that is external to the source code or other such information. While this approach can be useful, it is only effective if it is consistently maintained and up to date with what is present within the source code.

Need to clean up this paragraph a bit

With this in mind, the question naturally arises: *How are vocabularies established and what effect does the process of software evolution have upon them?*

In this thesis we investigate the ways in which programmers use language to form a vocabulary of terms within source code (**re: distribution of terms**), how vocabularies grow to support growth in the software itself **re: growth in number of unique terms** and the changes in how vocabulary is applied as the software evolves **re: rich tokens**. Additionally, we investigate the laws of software evolution in the context of source code vocabulary determine their applicability in describing evolution of the source code vocabulary.

1.1 Research Goals

This study aims to provide an understanding of the nature of vocabulary within the source code of software systems and the impact that the process of software evolution has upon source code vocabulary.

Not happy with this paragraph – needs a stronger/clearer argument

The study is driven by the desire to understand how vocabularies are built and distributed within software systems, how these properties change and in what stages of the software's evolution such changes occur. This knowledge can aid developers and managers in capturing and maintaining documentation for a vocabulary representative of that which is used within the source code by outlining terms which may be fundamental to the system, and by highlighting substantial changes in the vocabulary, giving a clearer indication of when the vocabulary should be documented and what requires documentation. This understanding can be also be applied in the comparison of different types of software systems, to assess the quality and complexity of the vocabulary that has been established and the viability of using the software system.

The goal of this research is to provide a descriptive model of how vocabularies are established and then changed throughout the lifetime of a software system, with the aim of identifying patterns of source code vocabulary evolution that will allow a general understanding of the nature of the evolution, as opposed to investigating activities of software evolution that are the cause of changes to the vocabulary.

1.2 Research Approach

TODO: Rewrite/complete writing this section once we can be a little bit more precise about the approach

What is the approach to addressing the problem? - Stats for the dataset we used (x classes, open source systems) - Data extraction (Mutations)

We study open source Java software systems to due their non-restrictive licensing and available, in fact we use X number for this study.

We extract metric information from the software systems JARs and prepare a representation of the history for the system.

Then, we extract identifiers from the class files and extract individual words from the identifiers to build a representative vocabulary present within the system.

TODO: Bit more detail about what we are trying to find using the measures we used

We then utilise the vocabularies we have extracted for each of the versions and examine certain aspects. We use frequency distribution histograms and the Gini coefficient to measure the distribution of terms within the vocabulary and how this changes over time.

1.3 Main Research Outcomes

In this thesis, we address the problem of identifying how vocabularies are established in software systems, the ways in which they grow to support new terms and the changes in how the set of terms within the vocabulary is applied within the source code.

This thesis makes the following contributions to the body of knowledge of source code vocabulary evolution:

Firstly, we investigated how vocabularies are formed within source code, relating to the collection of terms within the vocabulary and the frequency at which terms are applied across the system. We found that the manner in which term usage is distributed throughout the source code is similar to that of the distribution of words within natural-language corpora, repeating the findings of [37].

Secondly, we investigated the nature of growth exhibited by source code vocabularies. We found that in all but a few **(quantify)** cases, the growth rate of the vocabularies is clearly sub-linear, indicating that the vocabularies will only grow to a certain extent, at which point further growth is only very minor. Through further analysis of the growth, we determine that a large number of new terms that attribute to the growth of the vocabulary are not heavily used within the source code, and thus, should not constitute a significant part in the mental model of developers.

Thirdly, we investigated how the usage of terms within the source code changes as the software is subject to evolution, finding that developers have a preference for frequent re-use of a relatively small number of terms across different versions of a software system, but that some terms are utilised only a small number of times within parts of the codebase and then never again.

Finally, we assessed the applicability of the laws of software evolution proposed by Lehman [25, 27] in describing the evolution of source code vocabularies. Through our assessment, we support for the First law *Continuing Change*, fourth law *Conservation of Organisational Stability*, fifth law *Conservation of Familiarity*, and the sixth law *Continuing Growth*. Despite this, our analysis was unable to provide evidence to show support for the other laws.

1.4 Thesis Organisation

Background provides an overview of research in the area of natural language vocabulary, software mental models and source code vocabulary and motivates this study.

Methodology outlines our approach to the selection of data for our study, the corpus that has been selected and the means of extracting vocabularies from the code for our analyses.

Findings highlights our findings in terms of the establishment and evolution of vocabularies within software systems.

Summary concludes the thesis and present opportunities for future work. In this chapter, we summarise our contributions resultant from the work within the thesis.

Chapter 2

Background

Chapter Introduction

2.1 Vocabulary and Linguistics

TODO: Get a tighter definition of vocabulary

Tidy up flow of this paragraph...just a brain dump for now

Need to be more effective selling the importance of vocabularies here

Will of course need citations of work studying how vocabulary is formed, the importance of vocabulary, how it changes, etc.

Vocabulary describes the collection of words that one has learnt and is able to recall.

We build vocabularies in order to more adequately and consistently describe things we encounter.

Broadening vocabularies can allow us to be more expressive and precise with our communication.

But terms can be synonymous or polysemic depending upon context, meaning we have to choose our words carefully.

Humans don't quite have the capacity to expand their vocabularies to cover a language-worth of terms, so our brain will prioritise it's capacity to remember terms that are of most important and most frequent use to us.

This allows to be more effective with our vocabularies

While vocabularies are flexible in order to suit our needs...their development comes over time and it is not practical to uproot it all at once

What are the key components to building an effective natural language vocabulary?...how can these principles be applied to building other types of vocabularies?

Linguistics play a role in our comprehension of something (how we relate to it/understand it with the building blocks we know)

2.2 Mental Models

Mental models describe our cognitive process in regard to comprehension of things we are exposed to

We build mental models as they allow us an understanding of something that is more akin to each individuals way of thinking

Team mental models are important, so that team members comprehensions of something are not wildly different (though not all the same – different perspectives can be a good thing...) – need to have a consistent representation of a problem if they are to work towards to a well-formed solution.

Languages play a significant role in building mental models, as we as humans associate our understanding with something using the language we have acquired that can be used to describe it.

2.3 Source Code Vocabulary

When it comes to source code vocabulary, there are very few boundaries regarding vocabulary

We have conventions and idioms that are popular adhered to, but not enforced ... (getters, setters, general principles for naming things...)

Problems addressed by software are so varied, that vocabularies will be different across different domains

There is more to represent in source code vocabulary than just the terminology relating to the domain, etc, which begs the question? What exactly constitutes vocabulary in source code?

Delorey *et al.* proposed a few definitions of vocabulary ... eventually settling on (one) because (the reason they chose)

Delorey noted that we lack an adequate definition of vocabulary within the context of source code – proposed levels at which we could define vocabulary...their study included just about everything as part of the vocabulary – even operators

Abebe et al. split the vocabulary across multiple levels of abstraction (class name, attribute name, method name, comment, etc.)

Citable: Lexicon Bad Smells ... [2] Analyzing the Evolution of the Source Code Vocabulary [1] Mining Programming Language Vocabularies From Source Code [12] The Programmer's Lexicon [20]

2.4 Software Mental Models

Software systems are complex entities, rich with conceptual information relating to the domain in which it operates and the problems it attempts to solve [7]. Users of software face the challenges of establishing adequate comprehension of what a piece of software is designed to do and how it aligns with what they want it to do. Accordingly, users must develop a mental model of the software in order for it to be useful for their purpose.

Developers have the responsibly of not only understanding how the software works, but also to communicate to a computer how it should behave and to other developers how it is constructed. Fortunately, developers are able to rely heavily on real-world objects and metaphors in communicating to other developers how a software system has been built. The use of programming paradigms such as object-oriented programming and design patterns allow a cognitive map for developers to apply traditional thinking within a software context. However, this freedom of expression is a double-edged sword, as developers are by no means obliged to be communicative when writing code.

Concepts, domain terms

Numerous studies have investigated the quality of identifiers within source code [2,8,9,11,22–24,41]. In [11], Deißenböck and Pizka define a formal model for well-formed identifiers, with rules for *concise* and *consistent* naming. Lawrie *et al.* perform an empirical investigation of 78 systems based upon this model to determine whether programmers conform to these rules in their code, revealing that developers frequently violated these rules **TODO: How frequently? Need some numbers here**. Lawrie *et al.* examine the impact of identifier completeness upon the comprehension of developers [24]. In their study, they assess the ability of over 100 developers to comprehend function identifiers at three different levels of completeness — *single letters*, *abbreviations*, and *full words* — based on well-known algorithms and production code samples. Their study showed that full word identifiers lead to the highest level of comprehension, although in a lot of cases the difference between abbreviation and full word identifiers was negligible.

2.5 Natural Language vs. Programmer Language

Citable: Corpus Linguistics ... [6] Corpus Linguistics ... [33] Selective Studies (Zipf) [44]

Studies of natural language/corpus linguistics have shown us that there are tendencies for humans to apply language in a particular way within natural language documents.

These studies have allowed us to develop ways in which we can assist in the production of text corpora (how?)

Lots of studies have been undertaken in the field of natural language. These studies have been effective in providing an understanding of how humans learn, process and use language.

Programming Languages, Natural Languages and Mathematics ... [34]

While programs are written using terms that are close to that of natural language, there is little knowledge as to whether the two are exactly the same.

Mining Programming Vocabularies ... [12]

An Empirical Exploration of Regularities in Open-Source Software Lexicons ... [37]

However, we suspect that there are some similarities between the two which would allow us to apply what we know of natural language to programmer languages

Investigation of the similarities between the language used in source code and the language used in natural language corpora indicate that there are definite similarities between the two, opening the door for the possibility of the application of natural language analysis techniques to source code.

Outline studies that have investigated the similarities between the two, what success they had and what they claimed it could yield

2.6 Software Evolution

2.7 Source Code Vocabulary Evolution

While there have been numerous studies that have investigated the nature of evolution of source code within software systems, which have given an idea of what to expect through various stages of evolution **(Cite some!)**; what is unclear is how the vocabulary that is captured within the source code evolves, as the system itself evolves.

It would be reasonable to assume that — given the vocabulary of the source code is a core part of the source code itself — it would be subject to the same evolutionary pressures. Antoniol *et al.* compared the evolution of the source vocabulary and that of the program structure [5], with a particular focus on their stability. Their study found that the vocabulary (or *lexicon*, as they referred to it) exhibited a high level of stability — higher than that of the program structure for all versions of the software systems they analysed. While they were able to find a correlation between the two, with major changes in structure yield similarly significant changes in the lexicon, they noted that the underlying distributions between the two were statistically different.

[5] also investigated the frequency of changes to program entities (methods/functions) resultant from identifier refactoring, with the results showing that changes to the lexicon brought about via renaming are very rare, implying developers are hesitant to alter the names of entities within a program once they have been put in place. They speculate that this is due to a reluctance of developers to bring about substantial changes to their mental model of the software.

Abebe *et al.* [1] found that the size of the system and the vocabulary exhibit similarities in their evolution.

Their study established separate vocabularies for the different levels of abstraction within source code, including *class name*, *attribute name*, *function name*, *parameter name* and *comment* vocabularies. Interestingly, they found that of the vocabularies they had established, the comment vocabulary was the largest contributor to evolution of the vocabulary and rich in representation of the entire vocabulary, containing over $\frac{3}{4}$ of the terms found in any of the vocabularies.

Additionally, they investigated the introduction of new identifiers within a release, to determine whether they were bringing about new terms or whether there was a tendency towards re-use of existing terms. Their findings were that new identifiers consisted of only existing terms in 56% percent and 70% of cases for ALICE and WinMerge respectively.

2.8 Summary

Summarise current state of research that is most closely related to this thesis and highlight the gaps

Outline how our research fits in and how it addresses those gaps

Chapter 3

Method

Introduce Method chapter

3.1 Data Collection

3.1.1 Types of Evolution Data

There are broadly 3 categories of data we could conceivably use for software evolution studies:

Cover what type of studies the different types of histories are appropriate

Say that we're using release-history for our study and outline some other studies that have used it to some success...include those directly studying vocabulary/vocabulary evolution

Provide a justification as to why we have chosen release-based history for the purposes of our study – include our rationale (i.e. what it enables us to do/find out)

History	Description
Release History	Source code, binaries, release notes, and release documentation
Revision History	Version control logs, issue/defect records, Modification history of documentation, Wiki logs
Project History	Messages (email, Instant message logs), Project documentation (plans, methodology, process)

Table 3.1: The different types of histories that typically provide input data for studies into software evolution.

3.1.2 Open Source Software (OSS)

Outline that we will be using OSS for our studies

Briefly cover why we've chosen OSS for our studies, as opposed to commercial projects

Highlight the benefits of using OSS

Rationalise that we are not missing anything too vital by not using commercial projects for our study

3.1.3 Open Source Project Repositories

List the places we got our systems from

3.1.4 Selection Criteria

List the selection criteria that we've used and descriptions for each

Rationale of Selection Criteria

TODO: Few sentences for each criterion – can probably cite Raj's work here for bolstering rationale

3.1.5 Selected Systems

Outline the categorisation scheme chosen for our studies

Briefly cover the systems selected and any noteworthy information

Table of systems selected for analysis

3.2 Statistical Measures

3.3 Research Questions

What are the growth patterns exhibited by vocabulary in source code?

How are terms distributed?

Does vocabulary follow laws of software evolution

3.4 Version Extraction

Introduce section by covering what information we need to extract for our study

Cover, broadly, how we intend to obtain this information from the choice of data we opted for

Introduce Mutations as being the tool to solve all of our woes

Broadly cover what Mutations does and what it is capable of

Mutations works by utilising the ASM bytecode manipulation framework to extract various properties from binary Java class (.class) files. Such properties include x, y and z. The tool, by default, also extracts the names of each of the fields, methods, interfaces and other such information associated with a class. All of this information is stored as

JSON formatted data in plain-text files.

Detail how Mutations goes about extracting the information from input JAR files

Cover in some depth, each stage Mutations goes through to obtain the data

3.5 Vocabulary Definition

Justify the definition of a vocabulary (i.e. why do we need to be precise about vocabulary? Need a consistent representation for comparison!)

Cover the definitions of others, what they lacked and what we have drawn upon from them.

Cover what we need from a definition (might want to shift this above previous paragraph). i.e. What do natural language vocabularies have relating to mental models that needs to be captured in programmer language.

Explain (at a high-level) our approach to defining vocabulary for our study, in light of the constraints outlined in previous paragraphs. (i.e. we wanted a representation of vocabulary that was quasi-unique to individual system – containing only terms that are likely to be the cause of cognitive overload)

We defined vocabulary as being being composed of the complete set of class names, method names and field names accumulated from each class within a given release of a software system.

The use of compound identifiers within software systems is common. While the compounding of words is often used to merge together to concepts within a given context, e.g. or even take on new meanings altogether TODO: Example of these. (can list this as a limitation...in general, our vocabulary definition has glaring

holes). We made the assumption that terms compounded together within programmer language adhere to the former, and thus, selected to treat each single word within the language as a distinct element of the vocabulary.

TODO: Be more specific about terminology – identifiers, words, tokens, terms, lexicon

3.6 Vocabulary Extraction

The class metric data extracted for each version of the systems used for our analysis provides the basis for forming the representation of vocabulary.

In order to build our vocabulary, we need terms. The vocabulary that we will use will be composed of various linguistic elements of the source code, including the following:

1. Class names
2. Method names
3. Field names

From these elements, we extract individual words. Precise/consistent extraction of distinct keywords from compound identifiers is a non-trivial task (even understanding these keywords can be difficult). **TODO: Need examples of coding styles or lack thereof producing hard to understand**. The use of adopted and well-adhered to coding standards aids in presenting identifiers in a clear and consistent manner. The coding standard for Java which has been prescribed by Sun **TODO: Is this in fact Sun's coding standard? Can we cite it? TODO: A compare and contrast example of using the Java coding standard and not using it** (and adapted by many projects) is conformed to by a number of open-source projects. In addition to this, there are popular tools which integrate into the build process which allow projects to enforce the use of such standards.

Note: the above points add weight to the notion that splitting based on the Java coding standard is somewhat sufficient – lots of projects use it + there are tools that are capable of enforcing it – we basically want to sell that it is adhered to well enough that we can use it as a basis for splitting identifiers into appropriate words

TODO: Might want to sneak this in here...or in limitations Need more sophisticated means of extracting distinct terms from identifiers to build a more representative vocabulary.

Because of this, we use the Java coding standard as the basis for separating identifiers into individual elements of the vocabulary.

Outline the standards for the elements we will be extracting terms from and what they would yield – examples

- Camel case
- Pascal case
- Upper-case and underscore separated

Report|Writer report|Writer REPORTWRITER

Outline how we actually go about extracting the terms from the data we have

Extraction process diagram JAR Extractor -> Class Metric Extraction -> Class Vocabulary Extraction -> Class Vocabulary Merging -> Vocabulary Metric Extraction

For building our representation of vocabulary for the systems we analysed, we extended the capabilities of an existing software evolution analysis tool, *Mutations*.

Using Mutations, we get programmatic access to this information through the ClassMetricData abstraction, which encapsulates the metadata associated with classes highlighted in the previous paragraph.

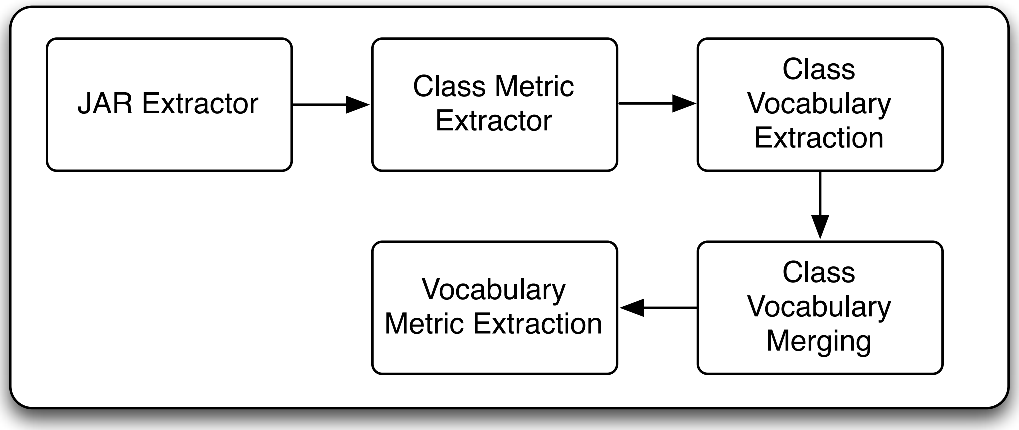

Figure 3.1: Vocbulary metric extraction process.

First, we use Mutations to extract the class metric data, as described in Section **TODO: Refer to section here**. Once we have extracted each class in the version, we begin to extract the vocabulary from each of the ClassMetricData objects. This entails the mining of words from the set of both method names and field names for each class within the version. In order to keep track of our vocabulary whilst processing the sets of identifiers, we maintain a map of class name -> token -> occurrences (i.e. the number of times a token has been used). With each occurrence within an identifier, we increment the occurrence count for the token encountered.

TODO: Need trim list justification/description somewhere above

TODO: Borrow from Raj's thesis for a description of what Mutations does

Token extraction algorithm for different elements – include reg-ex

Merging of vocabulary once the various sets have been established

TODO: How do we generate the measures used once we have extracted the vocabulary?

```java
/** **/
public void extractVocabulary(History history)
{
  for(Version version : history.getVersions())
  {
    for(ClassMetricData classMetricData : version.getVersions())
    {
      extractClassNameTokens(classMetricData.getClassName());
      extractMethodTokens();
      extractFieldTokens();
    }
  }
}

private void extractClassNameTokens(String className)
{

}

private void extractMethodTokens(Set<String> methodSignatures)
{
  for(String methodSignature : methodSignatures)
  {

  }
}
```

Listing 3.1: Same of bytecode generated for a simple Hello World method

Type	**Token**
Primitives	class, string, char, int, long, float, double, byte, object
Unambiguous Terms	get, set, val, value, name, impl, listener, event, this

Chapter 4

Findings

Start by re-introducing the findings of others that have performed similar studies (only cover what is closely related to what will be investigated within this chapter) – provides context/background for the our investigation and allows us to clearly differentiate

TODO: Need to find an appropriate place to identify the method of

[1] observed that most new identifiers are composed of existing terms, rather than new terms

[5] observed that the lexicon is more stable than the system as a whole and that changes to the lexicon are unlikely

Contrast their work to what will be investigated within this chapter

Identify why this contrast has been applied for our study (i.e. what new and valuable information does it potentially yield)

Clearly summarise what questions will be addressed within this chapter

TODO: Trim the questions down

- What are the trends for growth in total size of the vocabulary across evolving software systems? Does this match findings for the growth of software systems as a whole?
- How stable is the growth of vocabulary across? Does the amount of growth vary from version to version and is there more volatility in earlier versions?
- How does the growth in vocabulary size compare to that of system size?
- What is the distribution of term usage across systems?
  - Is there a tendency to use particular terms with greater frequency than others?
  - Is the distribution profile preserved throughout evolution?
- What kinds of terms are used with greatest frequency?
  - Is there a bias towards using either nouns or verbs and is this preserved?
  - What do the most frequently used terms refer to?
  - Do the most popularly used terms maintain their status across the entire or do new terms take their place?

In this chapter, we address the above questions and discuss the implications of our findings. **TODO: Give a neat stat that summarises how much data we used for the analysis.**

4.1 Growth in Vocabulary

4.1.1 Measuring Growth in Vocabulary

Plots of the total token count against the number of days since the initial version for each version, which is obtained through the extraction of tokens from each version of software

Regression for the plot is generated to summarise the growth. From this we can determine if the growth is sub-linear, linear or super-linear

Using these plots we assess the stability of growth by observing how the versions fit this model (i.e. if the residual values closely fit the model, then the growth is stable, otherwise it is volatile) **TODO: Clean this up, be more precise with how stability is defined**

Growth of system size compared to vocabulary size is determined by plotting the system's raw size in the same manner as the vocabulary size and observing similarities in their growth patterns.

4.1.2 Observations

Patterns of Growth

What are the trends for growth in total size of the vocabulary across evolving software systems? Does this match growth exhibited by evolving software systems as a whole?

Explain how we applied the measurements defined above to assess the patterns of growth

Indicate how many of the systems that we analysed were of a particular pattern of growth for each type of growth. Sentence or two or a table.

Highlight the differences in the results for growth patterns. Systems were mostly sub-linear, particular those which were more mature. Some systems show much stronger sub-linearity, to the point where the number of terms being introduced is a very small number.
Some systems even exhibit a substantial decrease in the total number of tokens as time goes on. We speculate the cause of this may be the removal of functionality that was previously included as part of the main development branch which is now distributed as a separate module.
Strongly sub-linear JChemPaint - flattens out after release at around 1150 days, Jena - Flattens out at after release around 900 has a small jump, then has a couple of large decreases
Linear Castor, Groovy, Jasperreports, KoLMafia, JRuby, RSSOwl (one release makes it linear...)
Super-linear ASM - small, Cassandra - new and volatile, Tapestry - very strange growth pattern (investigate further!)

Chart highlighting a stable, sub-linear growth pattern – use Azureus for this

Chart highlighting a linear growth pattern – use JRuby

Growth Stability

How stable is the growth of vocabulary across? Does the amount of growth vary from version to version and is there more volatility in earlier versions?

Vocabulary Size vs. System Size

How does the growth in vocabulary size compare to that of system size? Is there any indication that the two are related or are they on different tangents?

Table comparing growth types for system size and terms

Chart comparing a stable and unstable growth pattern – use Azureus and Hibernate

Speculate as to what is most likely the contributing factor to identified growth patterns (relating to stability of growth)

Speculate as to why most systems are exhibiting sub-linear growth, while some are linear or even super-linear. This could be related to age of the software, number of releases, architecture, etc. Are there are any cases in which the system size is clearly following vocabulary size? Is this common? Reason as to what the relation (if any) between the two may be and why they may be fundamentally different.

TODO: Need to generate growth plots for system size based on bytecode, rather than class count (or some other metric)

Use example of late total token growth for Hibernate – this growth is caused by introduction of an associated module, which was included as part of the core distribution. Can probably reason that this kind of growth may not be as significant as growth which is including functionality that has been only just been developed. **TODO: See if more insight can be gained into what contributes to large jumps in growth and general instability in growth – is this something that is caused by process (e.g. the introduction of existing term-rich modules or the introduction of new functionality, or does it occur through refinement. Note: This level of investigation is most likely out of scope, though we can probably speculate in order to provoke)**

Explain the implications of these findings – should we be worried about the rate at which vocabulary tends to grow? Well, examining a broad picture of growth alone does not really give us much insight into specifics of the vocabulary. Why should we care about total term growth if none of the terms accounting for growth form part of the vocabulary (i.e. are semantically rich terms that require consistent understanding and therefore require documentation)?

To get a better idea of this, we really need to examine the application of the terms that make up the vocabulary. This will be a nice segue into the following chapter, which will examine just that

4.2 Distribution of Vocabulary

4.2.1 Measuring Distribution of Vocabulary

Frequency distributions (to plot term usage distribution profile)

Gini coefficient (to determine distribution of wealth amongst terms)

4.2.2 Observation

Distribution Profiles

Distribution of Term Usage

Types of Terms

TODO: Think of a much better title than this

Popular Terms

Frequency distribution charts which show the similarity in application of terms across systems

Discuss why the vocabulary is consistently distributed in the same manner across systems

Might be a nice time to bring up Zipf's law here

Gini charts which show the trend for term wealth distribution over time

Highlight the continually increasing Gini over time across systems

Discuss cases in which there are versions that match in Gini vs. Growth. Reason as to what this might be attributed to. Is it safe to say that in cases where there is this parallel, these versions are significant in terms of the vocabulary. What explanations may or may not support this reasoning?

4.3 Implications

Open up the discussion here to go into further detail for what was covered in the previous sections. What could our findings mean and why should people care?

Discuss the findings of growth and distribution in the same context. What information were we able to reveal about the combination of these two facets that is meaningful to people?

Chapter 5

Conclusions

[1–44]

Appendix A

Meta Data Collected for Software Systems

References

[1] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, G. Antoniol, and T. FBK-irst. Analyzing the Evolution of the Source Code Vocabulary. In *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR'09*, pages 189–198, 2009.

[2] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus. Lexicon bad smells in software. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 95–99. IEEE Computer Society, 2009.

[3] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 4. IBM Press, 1998.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[5] G. Antoniol, Y. Gueheneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during sofware evolution. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*, pages 14–23, 2007.

[6] D. Biber, S. Conrad, and R. Reppen. *Corpus linguistics: Investigating language structure and use.* Cambridge Univ Pr, 1998.

[7] T. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, page 498. IEEE Computer Society Press, 1993.

[8] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, page 112. IEEE Computer Society, 1999.

[9] B. Caprile and P. Tonella. Restructuring program identifier names. In *International Conference on Software Maintenance, 2000. Proceedings*, pages 97–107, 2000.

[10] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, pages 446–465, 2005.

[11] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[12] D. Delorey, C. Knutson, and M. Davies. Mining Programming Language Vocabularies from Source Code. In *21st Annual Psychology of Programming Interest Group Conference*, 2009.

[13] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining Source Code to Automatically Split Identiers for Software Analysis. 2009.

[14] L. Etzkorn, L. Bowen, and C. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(3):236, 1999.

[15] L. Etzkorn, C. Davis, and L. Bowen. The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics*, 33(11):1731–1756, 2001.

[16] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*. Citeseer.

[17] B. Fluri, M. W
"ursch, and H. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Working Conference on Reverse Engineering*, pages 70–79. Citeseer, 2007.

[18] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension-Volume 00*, pages 113–122. IEEE Computer Society, 2008.

[19] J. Hoc. Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies*, 9(1):87–105, 1977.

[20] E. Høst and B. Ostvold. The Programmer's Lexicon, Volume I: The Verbs. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, 2007. SCAM 2007*, pages 193–202, 2007.

[21] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[22] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006. SCAM'06*, pages 139–148, 2006.

[23] D. Lawrie, H. Feild, and D. Binkley. An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229, 2007.

[24] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. 2006.

[25] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[26] M. Lehman and L. Belady. *Program evolution: processes of software change*. 1985.

[27] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *metrics*, page 20. Published by the IEEE Computer Society, 1997.

[28] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Citeseer, 2006.

[29] E. Linstead, L. Hughes, C. Lopes, and P. Baldi. Exploring Java software vocabulary: A search and mining perspective. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 29–32. IEEE Computer Society, 2009.

[30] J. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, page 112. IEEE Computer Society, 2001.

[31] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 289, Washington, DC, USA, 2002. IEEE Computer Society.

[32] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE Computer Society, 2003.

[33] T. McEnery and A. Wilson. *Corpus linguistics*. Edinburgh, 2001.

[34] P. Naur. Programming languages, natural languages, and mathematics. *Communications of the ACM*, 18(12):676–683, 1975.

[35] M. Ohba and K. Gondow. Toward mining concept keywords from identifiers in large software projects. *ACM SIGSOFT Software Engineering Notes*, 30(4):5, 2005.

[36] D. Perry, A. Porter, and L. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM, 2000.

[37] D. Pierret and D. Poshyvanyk. An empirical exploration of regularities in open-source software lexicons. In *Proc. of the Int'l Conf. on Program Comprehension*, pages 228–232. IEEE, 2009.

[38] M. Porter. An algorithm for suffix stripping. 1997.

[39] M. Ramal, R. de Moura Meneses, and N. Anquetil. A disturbing result on the knowledge used during software maintenance. In *wcre*, page 0277. Published by the IEEE Computer Society, 2002.

[40] S. Sim and R. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *International Conference on Software Engineering*, volume 20, pages 361–370. Citeseer, 1998.

[41] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

[42] R. Vasa, M. Lumpe, and A. Jones. Helix - Software Evolution Data Set. `http://www.ict.swin.edu.au/research/projects/helix`, 2010.

[43] T. Winograd. Understanding natural language* 1. *Cognitive Psychology*, 3(1):1–191, 1972.

[44] G. Zipf. Selective Studies and the Principle of Relative Frequency in Language (Cambridge, Mass, 1932). *Human Behavior and the Principle of Least-Effort (Cambridge, Mass, 1949*.