Possible Organization for Writing a Thesis including a LaTeXFramework and Examples

by

A Graduate Advisor
B.Sc., University of WhoKnowsWhere, 2053
M.Sc., University of AnotherOne, 2054

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Whichever

Possible Organization for Writing a Thesis including a LaTeXFramework and Examples

by

A Graduate Advisor
B.Sc., University of WhoKnowsWhere, 2053
M.Sc., University of AnotherOne, 2054

Supervisory Committee

Dr. R. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

Supervisory Committee

Dr. R. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

ABSTRACT

This document is a possible Latex framework for a thesis or dissertation at UVic. It should work in the Windows, Mac and Unix environments. The content is based on the experience of one supervisor and graduate advisor. It explains the organization that can help write a thesis, especially in a scientific environment where the research contains experimental results as well. There is no claim that this is the *best* or *only* way to structure such a document. Yet in the majority of cases it serves extremely well as a sound basis which can be customized according to the requirements of the members of the supervisory committee and the topic of research. Additionally some examples on using LaTeXare included as a bonus for beginners.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my cat, Star Trek, and the weather, for supporting me in the low moments.

Supervisor Main, for mentoring, support, encouragement, and patience.

Grant Organization Name, for funding me with a Scholarship.

I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.

Edith Wharton

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

1.1 How to Start an Introduction

1.2 Is a Review of All Previous Work Necessary Here?

Chapter 2

The Problem

Introduction paragraph about the problem and the various motivations.

2.1 Study 1: Failure Inducing Developer Pairs

Technical dependencies in a project can be used to predict success or failure in builds or code changes [14, 22]. However, most research in this area is based on identifying central modules inside a large code base which are likely to cause software failures or detecting frequently changed code that can be associated with previous failures [10]. This module-based method also results in predictions at the file or binary level of software development as opposed to a code change level and often lack the ability to provide recommendations for improved coordination other than test focus.

With the power of technical dependencies in predicting software failures, the question I investigated in this study was : "*Is it possible to identify pairs of developers whose technical dependencies in code changes statistically relate to bugs?*"

This study explains the approach used to locate these pairs of developers in developer networks. The process utilizes code changes and the call hierarchies effected to find patterns of developer relationships in successful and failed code changes. As it will be seen, we found 27 statistically significant failure inducing developer pairs. These developer relationships can also be used to promote the idea of leveraging socio-technical congruence, a measure of coordination compared to technical dependencies amongst stakeholders, to provide coordination recommendations.

2.1.1 Technical Approach

Extracting Technical Networks

The basis of this approach is to create a technical network of developers based on method ownership and those methods' call hierarchies effected by code changes. These networks will provide dependency edges between contributors caused by code changes which may be identified as possible failure inducing pairings (Figure 2.1). To achieve this goal, developer owners of methods, method call hierarchies (technical dependencies) and code change effects on these hierarchies must be identified. This approach is described in detail by illustrating its application to mining the data in a Git repository although it can be used with any software repository.

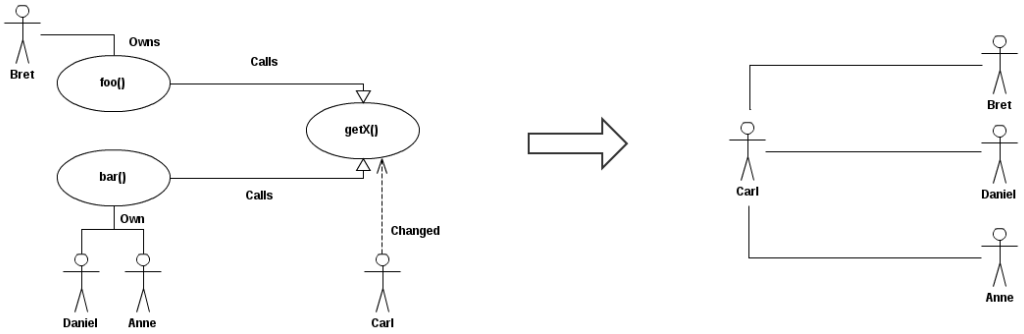

Figure 2.1: A technical network for a code change. Carl has changed method getX() which is being called by Bret's method foo() as well as Daniel and Anne's method bar().

To determine which developers own which methods at a given code change, the Git repository is queried. Git stores developers of a file per line, which was used to extrapolate a percentage of ownership given a method inside a file. If developer A has written 6/10 lines of method foo, then developer A owns 60% of said method.

A method call graph is then constructed to extract method call hierarchies in a project at a given code change. Unlike other approaches such as Bodden's et al. [3] of using byte code and whole projects, call graphs are built directly from source code files inside of a code change, which does not have the assumptions of being able to compile or have access to all project files. It is important to not require project compilation at each code change because it is an expensive operation as well as code change effects may cause the project to be unable to compile. Using source files also allowed an update to the call graph with changed files as opposed to completely rebuilding at every code change. This creates a rolling call graph which is used to show method hierarchy at each code change

inside a project opposed to a static project view. As some method invocations may only be determined at run time, all possible method invocations are considered for these types of method calls while constructing the call graph.

The code change effect, if any, to the call hierarchy is now found. The Git software repository is used to determine what changes were made to any give file inside a code change. Specifically, methods modified by a code change are searched for. The call graph is then used to determine which methods call those that have been changed, which gives the code change technical dependencies.

These procedures result in a technical network based on contributor method ownership inside a call hierarchy effected by a code change (Figure 2.1 left hand side). The network is then simplified by only using edges between developers, since I am only interested in discovering the failure inducing edges between developers and not the methods themselves (Figure 2.1 right hand side). This is the final technical network.

Identifying Failure Inducing Developer Pairs

To identify failure inducing developer pairs (edges) inside technical networks, edges in relation to discovered code change failures are now analysed. To determine whether a code change was a success or failure (introduce a software failure), the approach of Sliwerski et al. [19] is used. The following steps are then taken:

1. Identify all possible edges from the technical networks.
2. For each edge, count occurrences in technical networks of failed code changes.
3. For each edge, count occurrences in technical networks of successful code changes.
4. Determine if the edge is related to success or failure.

To determine an edge's relation to success or failure, the value FI (failure index) which represents the normalized chance of a code change failure in the presence of the edge, is created.

$$\text{FI} = \frac{\text{edge}_{failed}/\text{total}_{failed}}{\text{edge}_{failed}/\text{total}_{failed} + \text{edge}_{success}/\text{total}_{success}} \tag{2.1}$$

Developer pairs with the highest FI value are said to be failure inducing structures inside a project. These edges are stored in Table 2.1. A Fisher Exact Value test is also preformed on edge appearance in successful and failed code changes, and non-appearance

in successful and failed code changes to only consider statistically significant edges (Table 2.1's p-value).

2.1.2 Results

To illustrate the use of the approach, I conducted a case study of the Hibernate-ORM project, an open source Java application hosted on GitHub[1] with issue tracking performed by Jira[2].

This project was chosen because the tool created only handles Java code and it is written in Java for all internal structures and control flow and uses Git for version control. Hibernate-ORM also uses issue tracking software which is needed for determining code change success or failure [19].

In Hibernate-ORM, 27 statistically significant failure inducing developer pairs (FI value of 0.5 or higher) were found out of a total of 46 statistically significant pairs that existed over the project's lifetime. The pairings are ranked by their respective FI values (Table 2.1).

Pair	Successful	Failed	FI	P-Value
(Daniel, Anne)	0	14	1.0000	0.0001249
(Carl, Bret)	1	12	0.9190	0.003468
(Emily, Frank)	1	9	0.8948	0.02165

Table 2.1: Top 3 failure inducing developer pairs found.

2.1.3 Analysis

Technical dependencies are often used to predict software failures in large software system [10, 14, 22]. This study has presented a method for detecting failure inducing pairs of developers inside of technical networks based on code changes. These developer pairs can be used in the prediction of future bugs as well as provide coordination recommendations for developers within a project.

This study however, did not consider the technical dependencies themselves to be the root cause of the software failures. This study focused purely on developer ownership of software methods and the dependencies between developers as the possible root cause of

[1] https://github.com/

[2] http://www.atlassian.com/software/jira/overview

the failures. To study this root cause futher, a study of indirect conflicts and their relationship to developer code ownership will be conducted.

2.2 Study 2: Awareness with Impact

In response to Study 1, a second investigation was conducted. Study 1 revealed that pairs of developers can be used around technical dependencies in order to predict bugs. The natural follow up to these findings was to conduct a study of indirect conflicts surrounding source code changes that involved the source code being changed and the developers involved in those changes. These indirect conflicts were primarily studies through the notion of task awareness.

Tools have been created to attempt to solve task awareness related issues with some success [2,9,16,21]. These tools have been designed to solve task awareness related issues at the direct conflict level. Examples of direct conflict awareness include knowing when two or more developers are editing same artifact, finding expert knowledge of a particular file, and knowing which developers are working on which files. On the other hand, task awareness related issues at the indirect conflict level have also been studied, with many tools being produced [1,15,18,20]. Examples of indirect conflict awareness include having one's own code effected by another developer's source code change or finding out who might be indirectly effected by one's own code change. Previous interviews and surveys conducted with software developers have shown a pattern that developers of a software project view indirect conflict awareness as a high priority issue in their development [1,4, 7,17].

Indirect conflicts arising in source code are inherently difficult to resolve as most of the time, source code analysis or program slicing [**?**] must be performed in order to find relationships between technical objects which are harmed by changes. While some awareness tools have been created with these indirect conflicts primarily in mind [1,20], most have only created an exploratory environment which is used by developers to solve conflicts which may arise [18]. These tools were not designed to detect indirect conflicts that arise and alert developers to their presence inside the software system. Sarma et al. [15] has started to work directly on solving indirect conflicts, however, these products are not designed to handle internal structures of technical objects.

In this study, I report on research into supporting indirect conflicts and present the design, implementation, and future evaluation of the tool *Impact*, a web based tool that aims at detecting indirect conflicts among developers and notifying the appropriate members in-

volved in these conflicts. By leveraging technical relationships inherent of software projects with method call graphs [12] as well as detecting changes to these technical relationship through software configuration management (SCM) systems, *Impact* is able to detect indirect conflicts as well as alert developers involved in such conflicts in task awareness while limiting information overload by using design by contract [13] solutions to method design. While this study outlines *Impact's* specific implementation, its design is rather generic and can be implemented in similar indirect conflict awareness tools.

2.2.1 Impact

This section will proceed to give an outlined detail of *Impact* in both its design and implementation. The design of *Impact* was created to be a generic construct which can be applied to other indirect conflict awareness tools while the implementation is specific to the technical goals of *Impact*.

Design

Compared to tool design for direct conflicts, the major concern of indirect conflict tools is to relate technical objects to one another with a "uses" relationship. To say that object 1 uses object 2 is to infer a technical relationship between the two objects which can be used in part to detect indirect conflict that arise from modifying object 2. This kind of relationship is modeled based on directed graphs [8]. Each technical object is represented by node while each "uses" relationship is represented by a directed edge. This representation is used to indicate all indirect relationships within a software project.

While technical object relationships form the basis of indirect conflicts, communication between developers is my ultimate goal of resolving such conflicts (as was seen in Study 1). This being the case, developer ownership must be placed on the identified technical objects. With this ownership, we now infer relationships among developers based on their technical objects "uses" relationship. Developer A, who owns object 1, which uses object 2 owned by developer B, may be notified by a change to object 2's internal workings. Most, if not all, ownership information of technical objects can be extracted from a project's source code repository (CVS, Git, SVN, etc.).

Finally, the indirect conflict tool must be able to detect changes to the technical objects defined above and notify the appropriate owners to the conflict. Two approaches have been proposed for change gathering techniques: real time and commit time [5]. I propose the use of commit time information gathering as it avoids the issue of developers overwriting

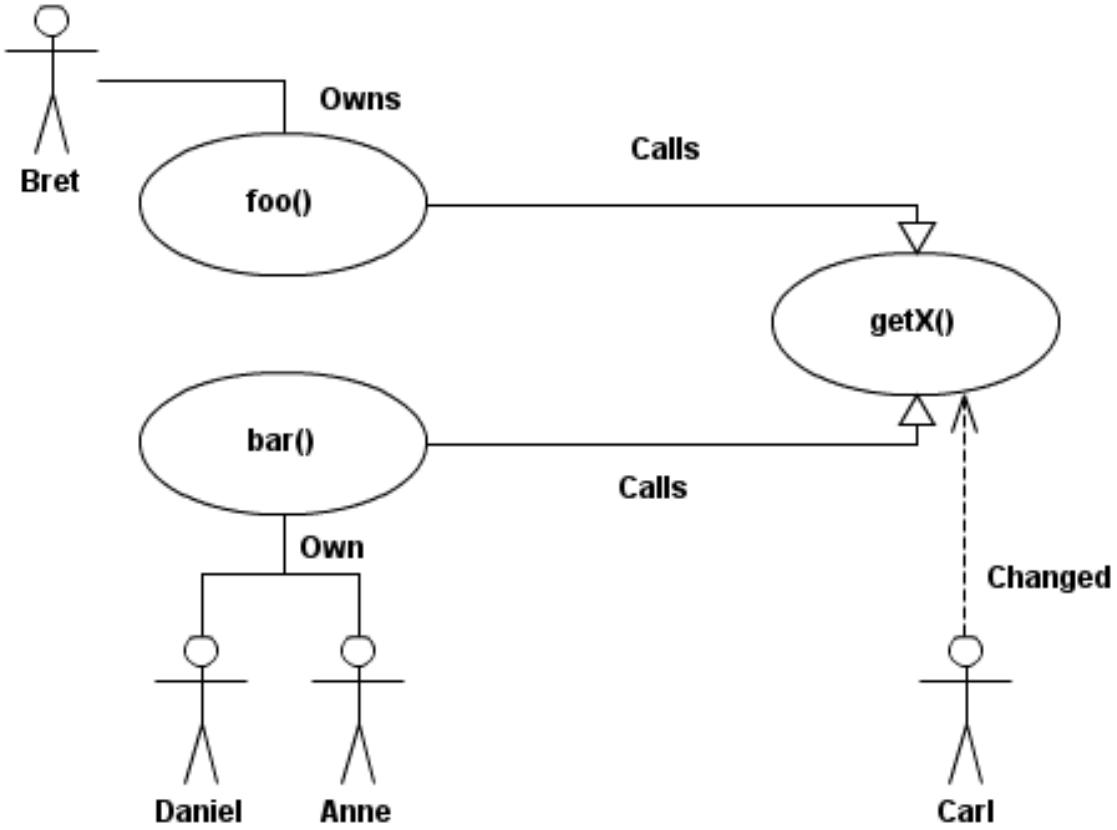

Figure 2.2: Technical object directed graph with ownership

previous work or deleting modifications which would produce information for changes that no longer exist. However, the trade off is that indirect conflicts must be committed before detected, which results in conflicts being apart of the system before being able to be dealt with as opposed to catching conflicts before they happen. At commit time, the tool must parse changed source code in relation to technical artifacts in the created directed graph detailed above. Where *Impact's* design differs from that of Palantir's is that the object's entire body (method definition and internal body) is parsed, similar to that of CASI [18], at commit time, as opposed to real time, to detect changes anywhere in the technical object. This is a first design step towards avoiding information overload. Once technical objects are found to be changed, appropriate owners of objects which use the changed object should be notified. In Figure 2.2, Carl changes method (technical object) 1, which effects methods 2 and 3 resulting in the alerting of developers Bret, Daniel, and Anne. I have opted to alert the invoking developers rather than the developer making the change to potential solutions as my conflicts are detected at commit time and this supports the idea of a socio-technical

congruence [11] from software structure to communication patterns in awareness systems.

With this three step design: (i) creating directed graphs of technical objects, (ii) assigning ownership to those technical objects, and (iii) detecting changes at commit time and the dissemination of conflict information to appropriate owners, I believe a wide variety of indirect conflict awareness tools can be created or extended.

Implementation

For *Impact's* implementation, I decided to focus on methods as my selected technical objects to infer a directed graph from. The "uses" relationship described above for methods is method invocation. Thus, in my constructed dependency graph, methods represent nodes and method invocations represent the directed edges. In order to construct this directed graph, abstract syntax trees (ASTs) are constructed from source files in the project.

Once the directed graph is constructed, I must now assign ownership to the technical objects (methods) as per the design. To do this, I simply query the source code repository. In this case I used Git as the source code repository, so the command *git blame* is used for querying ownership information. (Most source code repositories have similar commands and functionality.) This command returns the source code authors per line which can be used to assign ownership to methods.

To detect changes to technical objects (methods), I simply use a commit's *diff* which is a representation of all changes made inside a commit. I can use the lines changed in the *diff* to find methods that have been changed. This gives cause of potential indirect conflicts. I now find all methods in the directed graphs which invoke these changed methods. These are the final indirect conflicts.

Once the indirect conflicts have been found, I use the ownership information of technical objects to send notifications to those developers involved in the indirect conflict. All owners of methods which invoke those that have been changed are alerted to the newly changed method. Impact can been seen in Figure 2.3, the user interface of *Impact*. Here, in an RSS type feed, the changing developer, time of change, changed method, invoking methods, and commit message are all displayed. The weight provided is the percent changed of changed method multiplied by ownership of the invoking method. This allows developers to filter through high and low changes affecting their own source code.

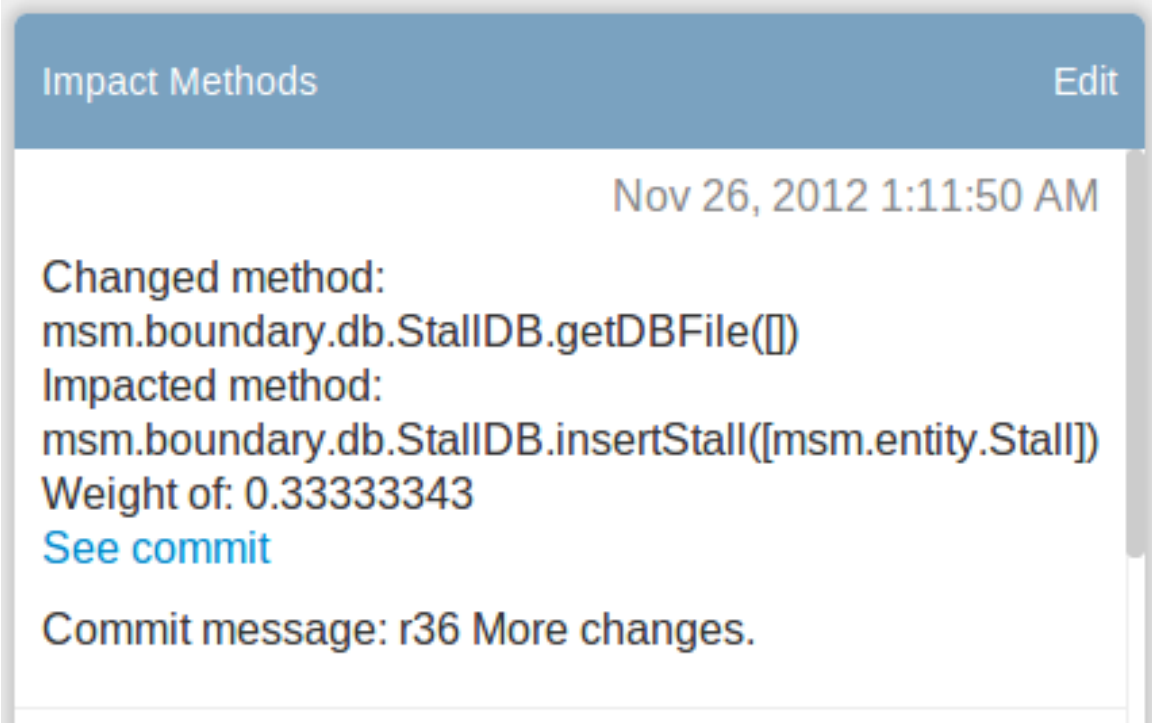

Figure 2.3: *Impact's* RSS type information feed.

2.2.2 Evaluation

To fully evaluate both the generic design of detecting and resolving indirect conflicts as well as *Impact*, extensive testing and evaluation must be performed. However, I felt that a simple evaluation is first needed to assess the foundation of *Impact*'s design and claims about indirect conflicts at the method level.

I performed a user case study where I gave *Impact* to two small development teams composed of three developers. Each team was free to use *Impact* at their leisure during their development process, after which interviews were conducted with lead developers from each development team. The interviews were conducted after each team had used *Impact* for three weeks.

I asked lead developers to address two main concerns: do indirect conflicts pose a threat at the method level (e.g. method 1 has a bug because it invokes method 2 which has had its implementation changed recently), and did *Impact* help raise awareness and promote quicker conflict resolution for indirect conflicts. The two interviews largely supported the expectation of indirect conflicts posing a serious threat to developers, especially in medium to large teams or projects as opposed to the small teams which they were apart of. It was also pointed out that method use can be a particularly large area for indirect conflicts to

arise. However, it was noted that any technical object which is used as an interface to some data construct or methodology, database access for instance, can be a large potential issue for indirect conflicts. Interview responses to *Impact* were optimistically positive, as interviewees stated that *Impact* had potential to raise awareness among their teams with what other developers are doing as well as the influence it has on their own work. However, *Impact* was shown to have a major problem of information overload. It was suggested that while all method changes were being detected, not all are notification worthy. One developer suggested to only notify developers to indirect conflicts if the internal structure of a method changes due to modification to input parameters or output parameters. In other words, the boundaries of the technical objects (changing how a parameter is used inside the method, modifying the return result inside the method) seem to be more of interest than other internal workings. More complex inner workings of methods were also noted to be of interest to developers such as cyclomatic complexity, or time and space requirements.

These two studies have shown that my design and approach to detecting and alerting developers to indirect conflicts appear to be on the correct path. However, *Impact* has clearly shown the achilles heel of indirect conflict tools, which is information overload because of an inability to detect "notification worthy" changes.

2.2.3 Analysis

In this study, I have proposed a generic design for the development of awareness tools in regards to handling indirect conflicts. I have presented a prototype awareness tool, *Impact*, which was designed around the generic technical object approach. However, *Impact* suffered from information overload, in that it had too many notifications sent to developers.

A potential solution to information overload comes from the ideas of Meyer [13] on "design by contract". In this methodology, changes to method preconditions and postconditions are considered to be the most harmful. This includes adding conditions that must be met by both input and output parameters such as limitations to input and expected output. To achieve this level of source code analysis, the ideas of Fluri et al. [6] can be used on the previously generated ASTs for high granularity source code change extraction when determining if preconditions or postconditions have changed.

Aside from better static analysis tools for examining source code changes, the results of this study potentially imply a lack of understanding into the root causes of indirect conflicts. A theme of information overload to developers continues to crop up in indirect conflicts, of which the root cause should be examined in future studies.

2.3 Problem Description

Use the two case studies as motivating examples to show the full problem.

Chapter 3

The New Approach and Solution

This is where you go all out and tell us all about your new discovery and research related to the problem in the previous chapter. No arrogant sweeping statements which cannot be fully justified, but no false modesty either. You must impress your reader that you have accomplished something.

Simply summarized, this chapter should be comprised of at least two main sections, each with appropriate subsections. The first section should describe:

- what the new approach is;
- what is really totally new;
- what is incrementally new;
- what you built upon.

The second part should describe fully how the new approach works, both with the overall theoretical exposition (e.g. an algorithm) and with as many examples as necessary for clarity. Remember that if the reader does not understand fully, you will get a lot of questions and doubts. Good examples, good figures, good diagrams with super clear tutorial explanations can be a joy to read and make even a small contribution appear to be more impressive. Are you afraid that if you are too tutorial your work will not seem as deep and difficult? Only shallow people will make such a superficial evaluation, have trust instead in the wisdom of your supervisory committee.

Use at least one good example throughout, and even better if this is one of the examples you used in Chapter 2 to describe the original problem.

By the way, this would be the first chapter I would write. This is what I know best right now, as I just finished working on it. It is clear to me and on the tip of my fingers. Start with your strengths! The second chapter I would write is the next one about the experiments, followed closely by chapter 2 describing the problem. It may not seem intuitive to you, but it works and it is the most productive way I ever found to finish a document.

3.1 Some LaTeXExamples

A Latex document is composed of two parts: the Preamble, and the Document Body. The *Preamble* is the site for inclusion of all document set up commands: definition of new commands, inclusion of prebuilt packages, template declaration, etc. The *Body* is where the document content is placed.

3.1.1 Preamble

The Preamble refers to the input which precedes the documents contents. It is the area where the author determines the general template for the document using the **documentclass**$[options]${*do style*} command. For example, \documentclass$[11pt]${*article*} declares that a document will follow the *article* document class, and have 11pt font.

If the document requires support of any library packages they must be included in the preamble using the **usepackage**{*package name*} command. For example, \usepackage{graphicx} is the command needed to include the graphicx package.

3.1.2 Document Body

The document body is the area which follows the Preamble. It is defined by the **begin**{*document*} and **end**{*document*} commands. The content of a Latex document is declared in the document body. Input which appears after the \end{document} command is ignored.

3.2 How to Number Pages

To number the pages of a document use the **pagenumbering**{*style*} command. Numbering is defined in the documents preamble. There are several different *styles* to choose from.

Numbering Style	Output
\pagenumbering{arabic}	1, 2, 3, ...
\pagenumbering{roman}	i, ii, iii, ...
\pagenumbering{alph}	a, b, c, ...
\pagenumbering{Roman}	I, II, III, ...
\pagenumbering{Alph}	A, B, C, ...

Table 3.1: Page Numbering Styles

The numbering of pages for a thesis is, however, much more complex than for an article and, in fact, the *book* class has been adopted. Make changes to those settings only if you are really familiar with LaTeX.

3.3 How to Create a Title Page

A title page can be either on a separate page or integrated directly into the first page of the document. It is defined by three declarations, followed by the **maketitle** command as illustrated below.

```
\title{Title of Paper}
\author{Author(s) of Paper}
\date{Publication Date}
\maketitle
```

The article document class defaults on an integrated title page. To make a separate title page, use the **titlepage** option with the \documentclass$[titlepage]${*doc style*} command.

For this thesis style the title page has been completely formatted for you. Just insert the various names of people in the supervisory committee, the title, your name and so on in the location where the *dummy* entries exist right now and you will be done. I would suggest to avoid doing any other changes unless you are absolutely sure!

3.4 How to Create an Abstract

To create an abstract, place contents of abstract between the **begin**{*abstract*} and **end**{*abstract*} commands.

3.5 How to Create a Table of Contents

The **tableofcontents** command automatically generates a table of contents from all section headers. The default behavior for the article document class is to produce an integrated table of contents. However, the document can be altered to generate the table of contents on a separate page using the **newpage** command (see section Formatting Extras).

For this thesis template a special command has been added, namely the **textTOCadd**. You can find it in the file *macros/style.tex*. It has to be explicitly called for an insertion into the Table of Contents and it is already in place appropriately for the existing sections and subsections.

3.6 How to Create Sections

Creating sections, subsections, and subsubsections is completed using the \section{Section Name}, \subsection{Subsection Name} and \subsubsection{Subsubsection Name} commands, respectively. Each sectional division is numerically labeled with respect to it's placement in the section hierarchy. For example, this section was defined with the code:

```
\section{How to Create Sections}
Creating sections, subsections, and ...
```

It is useful to give a label using the **label** command to a section or subsection if a reference to it is made, so that the reference will be automatically updated should the structure of the document change.

3.7 How to Create a List

Lists can be either enumerated, non enumerated, or descriptive. Each element of a list is termed an 'item'.

1. enter the list environment with the \begin{*list style*} command.
2. define each item with the \item command for non\enumerated lists, or \item[*label*] for descriptive lists.
3. terminate list environment with the \end{*list style*} command.

loc	Purpose
l	left justified column
r	right justified column
c	centered column
|	vertical rule

Table 3.2: Table Example

3.8 How to Insert Tables, Figures, Captions, and Footnotes

The table and figure environments contain input blocks which cannot be split across pages. Rather than divide the input of either of these environments, the contents are relocated, or floated, to a location in the document which optimizes page layout with the surrounding document content.

3.8.1 Tables

Tables are created in the tabular environment. A single parameter is used to define the number of columns and item justification pertaining to each column. The single parameter is a combination of the following ones shown in Table 3.2.

\\ and & are used to define rows and columns, respectively. A table can either have the contents of its rows and columns lined or not. Each line used to construct the table must be individually specified, using | and \hline for vertical and horizontal lines, respectively.

Table 3.2 was generated with the following input:

```
\begin{center}
    \begin{tabular}{|l|l|} \hline
    l & left justified column   \\ \hline
    r & right justified column  \\ \hline
    c & centered column         \\ \hline
    $|$ & vertical rule         \\ \hline
 \end{tabular}
 \end{center}
```

You will want to include your table in the ”List Of Tables” section at the beginning of your thesis. To do this you enclose the above table inside a table environment like so:

```
\begin{table}
    \begin{center}
    ...
    \end{center}
    \caption{Sentence describing table.}
    \label{unique:label}
\end{table}
```

The caption is the text that appears underneath the table. It should be short and precise. The label is a unique label that you can use to refer to the table within your document. You can use the `\ref{label}` to insert the table number into your text as in Table 3.2. In the example above you would use as in:

```
I am referring to Table \ref{unique:label}.
```

3.8.2 Figures

The first step to including an externally prepared image into a document, is to declare the graphixs package into the documents preamble. Integrating the image can be done using the figure environment. Enter and exit the figure environment with the **begin**{**figure**}[loc]**]** and **end**{**figure**} commands, respectively. The *loc* dictates the placement of the included image, and can be any of the following:

h here: location in text where the environment appears

t top: top of the page

b bottom: bottom of the page

p page of floats: on a separate page with no text

For organizational purposes, it is best to have keep all figures in a folder together. I usually label the folder as "*Figures*" (with great creativity) and I placed it in the same directory as the topmost main *.tex* file. Include the image into the document with the **includegraphics**[*dim*]{path to image} command. *dim* dictates the magnitude of the **`height`** or **`width`**. The image is scaled proportionally. An example and its resulting output follow below.

```
\begin{figure}[h]
\centering
\includegraphics[height = 1in]{LinuxPenguin.eps}
\caption{The Linux Penguin}
\end{figure}
```

Why is the output for the figure not shown? Because inserting figures into LaTeXis not that simple and it is highly dependent on the system you are using together with the type of figure. This is not the place to dwell upon the inconsistencies which can make your life difficult. Suffice it to say that the original LaTeXand its tools was geared to accept *.eps* files for figures and it still maintains that expectation if one compiles using a *Latex to dvi to (pdf or ps)* series of commands. On the other hand, if one uses the *Latex to pdf* direct path, then files of other types are perfectly fine (e.g. *pdf, jpg, gif, etc.*).

If you are interested, look at the actual file for this section namely "sec_latexhelp.tex" and consider the set of lines commented out just above this paragraph. There are two examples of insertion of figures, the first with the *.eps* version and the second with the *.pdf* version of the same picture (of a penguin). Delete the comments from one of the two sets and use the appropriate tools.

To refer to a figure, the same approach used for tables should be used, namely a `\ref{label}` command which includes the unique identifier label for that figure, as in:

```
I am referring to Figure \ref{unique:label}.
```

3.8.3 Captions

Captions for tables and figures are created using the **caption**{caption goes here}. Captions are automatically numbered with separate counters for tables and figures. **caption**{caption contents} can only be used in the Figure or Table environment.

3.8.4 Footnotes

Footnotes are inserted with the **footnote**{footnote contents} command. This footnote[1] is generated as follows:

```
...This footnote\footnote{this is a footnote} is
                    generated...
```

[1] this is a footnote

3.9 How to Alter Font

3.9.1 Type Style

Roman Family is the default type style. The types style can be modified using the following commands.

Command	Output
\textit{Italic Characters}	*Italic Characters*
\textsl{Slanted Chartacters}	*Slanted Characters*
\textsc{Small Cap Characters}	SMALL CAP CHARACTERS
\textbf{Boldface characters}	**Boldface characters**
\textsf{Sans Serif Characters}	Sans Serif Characters
\texttt{Typewriter Characters}	`Typewriter Characters`

3.9.2 Type Size

The font size can be modified using the following commands.

Command	Output
\tiny{tiny font}	tiny font
\scriptsize{scriptsize font}	scriptsize font
\small{small font}	small font
\normalsize{normalsize font}	normalsize font
\large{large font}	large font
\Large{Large font}	Large font
\huge{huge font}	huge font
\Huge{Huge font}	Huge font

3.10 Math Mode

To incorperate mathematical content into a document, Latex provides three different environments: Displaymath, Math, and Equation. Brief descriptions for each environment, and environment short cuts are displayed in the table below.

Environment	Function	Shortcut
`math`	`displays an in-text formula`	`\ ( ...\ )`
`displaymath`	`displays an unnumbered formula`	`\ [ ...\ ]`
`equation`	`displays a numbered formula`	`N/A`

The following examples, using Einstein's famous $e \doteq mc^2$ equation, illustrate how to include a formula into a document.

```
...Einstein's famous \( e \doteq mc^{2} \) equation, illustrate...

\[e \doteq mc^{2}\]


\begin{equation}
   \doteq mc^{2}
\end{equation}
```

$$e \doteq mc^2 \tag{3.1}$$

3.11 Formatting Extras

The following table illustrates some formatting tips for perfecting the layout of a Latex document.

Command	Purpose
`\hspace{`*len*`}`	`insert a horizontal space of length` *len*
`\vspace{`*len*`}`	`insert a vertical space of length` *len*
`\mbox{`*text*`}`	`ensure that` *text* `is not split over multiple lines`
`\\`	`new line`
`\newpage`	`start new page`
`\pagebreak`	`insert a page break`
`%`	`precedes comments`

Chapter 4

Experiments

Assuming you have some experimental results to support your claims this is where all the data is reported. There are a few issues you should consider before dumping a lot of stuff here, or it will lose its effectiveness.

First of all you must describe precisely the experimental setup and the benchmarks you used. In any scientific discipline an experimental result is only good if it is reproducible. To be reproducible then somebody else must have sufficient details of the setup to be able to obtain the same data. Thus the first section in this chapter is a super precise history of the decisions made towards experimentation, including mentions of the paths which became infeasible. The setup must be valid and thus your description of it must prove that it is indeed sound. At times, terrifying times, when writing this section, both supervisor and student realize belatedly that something is missing and more work needs to be done!

The second portion of this chapter is dedicated the the actual results. At least two issues arise here:

1. Should all the data be reported here or should some be placed in the Appendix?
2. Should this be an exposition of the raw facts and data or should it include its analysis and evaluation?

There are no definite answers here, but I follow a few rules.
Should all the data be reported here or should some be placed in the Appendix?

- If there is a large number of tables of data, it might be better to present here only a handful of the most significant (”best”) results, leaving all the rest of the data in the Appendix with proper linkages, as it would make the chapter so much more easily

readable (not to mention limiting the struggle with a word processor for the proper placement of tables and text).

- Use an example throughout, call it a ”case study” to make it sound better, so that all the data and results are somehow linked in their logic, and even better if this is one of the examples you used in Chapter 2 to describe the original problem.

- Highlight in some manner the important new data, for example the column of your execution speed where all the numbers are much smaller. Make the results highly easy to read!

- It is normally expected that data should be presented only in one form and not duplicated, that is, you are not supposed to include both a table of raw numbers and also its graphical representation from some wonderful Excel wizard. I tend to disagree. I would not wish to see every results repeated in this manner, but some crucial ones need to be seen in different manners, even with the same information content, in order to show their impact. One good trick is to place the more boring tables in the Appendix and use wonderful graphs in this chapter.

- This is the one chapter where I would splurge and use colour printing where necessary, as it makes an *enourmous* difference.

Should this be an exposition of the raw facts and data or should it include its analysis and evaluation?

- Is the evaluation of the data really obvious? For example you have 10 tables to show that your chemical process is faster in development and gives purer material - you may simply need to highlight one column in each table and state the obvious.

- Most results are not that obvious even if they appear so. Moreover this is where you are comparing your *new* results to data from other people. I usually describe other people’s work at this point and make comparisons. That is why I prefer to talk about the analysis and evaluation of the results in a separate chapter.

- There is absolutely no clear structure here which is best.

Chapter 5

Evaluation, Analysis and Comparisons

For a Master's research this chapter represents the critical part where **you** are truly evaluated to determine whether you should be given your degree. Even more so for a PhD. Consider carefully what the University calendar states regarding the expectations for a master's thesis, paraphrased here.

1. *A Master's thesis is an original lengthy essay.* The main implication here is that the essay is original, that is, it is completely newly written by you and does not contain any writings from others unless precisely quoted. Any paraphrased items must be cited.

2. *It must demonstrate that:*

   - students understand research methods;
   - students are capable to employ research methods;
   - students demonstrate command of the subject.

3. *The work may be based on:*

   - original data;
   - original exercise from scholarly literature;
   - data by others.

4. *The work must show that:*

   - appropriate research methods have been used;
   - appropriate methods of critical analysis supplied.

5. *The work must contain:*

    - evidence of some new contribution;
    - evidence of a new perspective on existing knowledge.

Only the last point uses the attribute *new* and it refers almost entirely to giving a new perspective and analysis, even if based on data from others. This truly implies that this current chapter on evaluation and analysis of results is the most important and must be written with care. You are demonstrating here that, even if given data and methods from others, your skills of critical judgment and analysis are now at the level that you can give professional evaluations.

Things are slightly different for a PhD. According to the Graduate Calendar:
a doctoral dissertation must embody original work and constitute a significant contribution to knowledge in the candidate's field of study. It should contain evidence of broad knowledge of the relevant literature, and should demonstrate a critical understanding of the works of scholars closely related to the subject of the dissertation. Material embodied in the dissertation should, in the opinion of scholars in the field, merit publication.

The general form and style of dissertations may differ from department to department, but all dissertations shall be presented in a form which constitutes an integrated submission. The dissertation may include materials already published by the candidate, whether alone or in conjunction with others. Previously published materials must be integrated into the dissertation while at the same time distinguishing the student's own work from the work of other researchers. At the final oral examination, the doctoral candidate is responsible for the entire content of the dissertation. This includes those portions of co-authored papers which comprise part of the dissertation.

The second paragraph makes it clear that one must emphasize what is new and different from others, without arrogance, yet without being too subtle either. The first paragraph implies that for a PhD it is required that one approached an important open problem and gave a new solution altogether, making chapters 3, 4, 5 all part of the body of research being evaluated. In fact at times even the problem may be entirely new, thus including chapter 2 in the examination. This is in contrast to a Master's degree where the minimum requirement is for chapter 5 to be original.

Chapter 6

Conclusions

My first rule for this chapter is to avoid finishing it with a section talking about future work. It may seem logical, yet it also appears to give a list of all items which remain undone! It is not the best way psychologically.

This chapter should contain a mirror of the introduction, where a summary of the *extraordinary* new results and their wonderful attributes should be stated first, followed by an executive summary of how this new solution was arrived at. Consider the practical fact that this chapter will be read quickly at the beginning of a review (thus it needs to provide a strong impact) and then again in depth at the very end, perhaps a few days after the details of the previous 3 chapters have been somehow forgotten. Reinforcement of the positive is the key strategy here, without of course blowing hot air.

One other consideration is that some people like to join the chapter containing the analysis with the only with conclusions. This can indeed work very well in certain topics.

Finally, the conclusions do not appear only in this chapter. This sample mini thesis lacks a feature which I regard as absolutely necessary, namely a short paragraph at the end of each chapter giving a brief summary of what was presented together with a one sentence preview as to what might expect the connection to be with the next chapter(s). You are writing a story, the *story of your wonderful research work*. A story needs a line connecting all its parts and you are responsible for these linkages.

Appendix A

Additional Information

This is a good place to put tables, lots of results, perhaps all the data compiled in the experiments. By avoiding putting all the results inside the chapters themselves, the whole thing may become much more readable and the various tables can be linked to appropriately.

The main purpose of an Appendix however should be to take care of the future readers and researchers. This implies listing all the housekeeping facts needed to continue the research. For example: where is the raw data stored? where is the software used? which version of which operating system or library or experimental equipment was used and where can it be accessed again?

Ask yourself: if you were given this thesis to read with the goal that you will be expanding the research presented here, what would you like to have as housekeeping information and what do you need? Be kind to the future graduate students and to your supervisor who will be the one stuck in the middle trying to find where all the stuff was left!

Bibliography

[1] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[2] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1313–1322, New York, NY, USA, 2007. ACM.

[3] Eric Bodden. A high-level view of java applications. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 384–385, New York, NY, USA, 2003. ACM.

[4] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 81 –90, aug. 2007.

[5] Geraldine Fitzpatrick, Simon Kaplan, Tim Mansfield, Arnold David, and Bill Segall. Supporting public availability and accessibility with elvin: Experiences and reflections. *Comput. Supported Coop. Work*, 11(3):447–474, November 2002.

[6] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[7] Christine A. Halverson, Jason B. Ellis, Catalina Danis, and Wendy A. Kellogg. Designing task visualizations to support the coordination of work in software devel-

opment. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 39–48, New York, NY, USA, 2006. ACM.

[8] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[9] Himanshu Khurana, Jim Basney, Mehedi Bakht, Mike Freemon, Von Welch, and Randy Butler. Palantir: a framework for collaborative incident response and investigation. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, IDtrust '09, pages 38–51, New York, NY, USA, 2009. ACM.

[10] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Irwin Kwan and Daniela Damian. Extending socio-technical congruence with awareness relationships. In *Proceedings of the 4th international workshop on Social software engineering*, SSE '11, pages 23–30, New York, NY, USA, 2011. ACM.

[12] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 273–284, New York, NY, USA, 1993. ACM.

[13] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[14] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[15] Anita Sarma, Gerald Bortis, and Andre van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 94–103, New York, NY, USA, 2007. ACM.

[16] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 23–33, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Adrian Schröter, Jorge Aranda, Daniela Damian, and Irwin Kwan. To talk or not to talk: factors that influence communication around changesets. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1317–1326, New York, NY, USA, 2012. ACM.

[18] Francisco Servant, James A. Jones, and André van der Hoek. Casi: preventing indirect conflicts through a live visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 39–46, New York, NY, USA, 2010. ACM.

[19] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[20] Erik Trainer, Stephen Quirk, Cleidson de Souza, and David Redmiles. Bridging the gap between technical and social dependencies with ariadne. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, eclipse '05, pages 26–30, New York, NY, USA, 2005. ACM.

[21] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 2:1–2:1, New York, NY, USA, 2008. ACM.

[22] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.