

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO



ANÁLISIS DE PATRONES DE
TRABAJO DE PROGRAMADORES
CON DATOS DE INTERACCIÓN

TESIS

QUE PARA OBTENER EL TÍTULO DE
MAESTRO EN CIENCIAS EN COMPUTACIÓN

PRESENTA

LUIS CARLOS CRUZ DÍAZ

ASESORES

DR. VÍCTOR MANUEL GONZÁLEZ Y GONZÁLEZ
DR. ROMAIN ROBBES

MÉXICO, D.F.

2016

“Con fundamento en los artículos 21 y 27 de la Ley Federal de Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada **“ANÁLISIS DE PATRONES DE TRABAJO DE PROGRAMADORES CON DATOS DE INTERACCIÓN”**, otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la biblioteca Raúl Baillères Jr., autorización para que fijen la obra en cualquier medio, incluido el electrónico, y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por tal divulgación una prestación”

LUIS CARLOS CRUZ DÍAZ

Fecha

Firma

Abstract

Este documento presenta una plantilla para usar en las tesis y tesinas del ITAM.
Se provee de manera gratuita y sin ninguna responsabilidad bajo la licencia
creative commons BY-SA 3.0.

Abstract

In this work we present a template for thesis and titulation works presented at ITAM. It is provided freely and without any responsibility under the *creative commons BY-SA 3.0*.

Contents

1. Introduction	1
2. Related work	2
2.1. Mining Software Repositories	2
2.2. Predicting errors	3
2.3. Interaction data programmer-IDE	4
2.4. Capturing usage data	4
2.5. Productivity metrics	5
2.6. Research with usage data	5
3. Data and Methodology	8
3.1. Description of the data	8
3.1.1. Eclipse Usage Data Collector	8
3.1.2. Codealike and ABB	10
3.2. Methodology	11
3.2.1. Preprocessing	11
3.2.2. Classification of events	12
3.2.3. Transformation	14
4. Results	17
4.1. Interruptions of work and productivity	17

CONTENTS

4.1.1. Relationship between the number of interruptions and productivity	18
4.1.2. Relationship between duration of interruptions and productivity	20
4.1.3. Local relationships between interruptions and productivity	23
4.1.4. Recovery Time after an interruption	27
4.2. Activities and working patterns	31
4.2.1. Identifying activities	31
4.2.2. Comparing activities between datasets	35
4.2.3. Identifying patterns of sessions	36
4.2.4. Comparison of working sessions patterns	36
4.2.5. Validation of the results	38
References	41

List of Figures

4.1. Boxplots showing the relation between the number of edits and selection events per minute, the edit ratio, and the number of interruptions.	19
4.2. Median values of edits and selection events per minute, and the edit ratio according to the number of interruptions and proportion of prolonged interruptions.	22
4.3. Local effect of an interruption in the user activity.	24
4.4. Cluster of sessions in UDC (left) and ABB (right).	38
4.5. Silhouette analysis of the clustering of chunks for UDC (top) and ABB (bottom).	39
4.6. Silhouette analysis of the clustering of sessions for UDC (top) and ABB (bottom).	40

List of Tables

3.1. Attributes of the UDC dataset.	9
3.2. Attributes of the ABB dataset.	11
3.3. Description of the detailed classification of events.	13
3.4. Statistics of sessions in UDC and ABB.	16
4.1. Effect size and significance of the relationship between number of editions per minute, selections per minute, edit ratio, and the number of interruptions on UDC data	20
4.2. Effect size and statistical significance of the relationships between the groups of sessions by proportion of prolonged interruptions and the number of interruptions in UDC	23
4.3. Local Effect of Interruption.	27
4.4. Weighted frequency of execution after positive or negative inter- ruptions.	30
4.5. Activities found via clustering for UDC (left) and ABB (right). .	34

Chapter 1

Introduction

Chapter 2

Related work

2.1. Studies on programmers' activities

2.2. Mining Software Repositories

The core of this work is within the Mining Software Repositories research area, which can be defined as the extraction of information from different artifacts (e.g. source code, control version logs, documentation and bug reports) that are produced throughout the software development cycle [?]. This term is not alien to data mining, so the intentions are extracting knowledge and discovering patterns in the data using a set of techniques and algorithms for this goal [?].

The software repositories can be found in different formats:

- **Historical repositories**, with information obtained throughout the evolution of a project like bug reports, emails and control version logs.
- **Runtime repositories**, like deployment information, performance reports and application usage data.
- **Code repositories**, which is access to the source code from a control version tool like SourceForge or Github.

Although some of this repositories are used to keep control of changes and

procedures, they are rarely used to make decisions. One of the goals of the Mining Software Repositories field is allowing static repositories to be a guide for decision making, for it is possible to discover important information and patterns that can help predict the future performance of the team and the quality of the product in development [?]. Each stage of the development of software releases data that can be exploited to make timely decisions. Recently, it was appointed the name of *Software Intelligence* for all the practices involving Mining Software Repositories within companies [?].

2.3. Predicting errors

There are tasks that have a greater impact with these practices. For example, the project administration can make decisions based on facts and tendencies reflected in the data that can be visualized with metrics, or even predict future events like modules prone to errors based on recent activity [?]

Related to the latter, there is a great effort in using this data for quality assurance. Khomf et al. (2012) analyzed product release data of the Firefox navigator while the developers were migrating from a traditional release scheme to an agile methodology and found that, though there are the same amount of errors, the users tend to identify them early and the failures are fixed quicker.

Error prediction is another example of the interest in quality. The work made by Nagappan et al. (2006) and Finaly (2014) are similar approaches of error prediction using complexity metrics, which are measures to the source code like the number of lines, functions, classes and more. Both works agree about the difficulty of predicting errors using solely this metrics and in the low precision of the proposed models. Zimmerman and Nagappan (2008) obtained better results considering the structure of the system and the relation between classes, creating a network and obtaining metrics from the related graph. The graph was built identifying the code dependencies of the program, where the nodes are classes and the edges the dependencies between them. From the graph they obtained metrics like the number of nodes, the amount of dependencies in relation to a node and the distance to a node. They found that, although the complexity metrics have better correlation with the amount of errors, the graph metrics

can predict well with regression models.

Meneely et al. (2008) also used networks to predict errors in modules based on product release data, but instead of representing classes as nodes, they represent the programmers that worked on the project and the edges are created when two of them work in the same piece of code. The authors also used graph metrics to train a predictive model, obtaining a precision of 81%.

2.4. Interaction data programmer-IDE

Another example of data that can be captured is the interaction data (also called usage data) between the programmer and the development tools [26], which are basically a log of the execution of events within the IDE (Integrated Development Environment). The IDE provide tools to execute tasks effectively, like navigation between classes and methods, continuous compilation, refactoring, automated testing and debugging, all designed to assist the programmers' productivity.

The usage data is composed of the interactions between the programmer and the IDE, and include the description of executed commands (e.g. copy, paste, delete and save); the opening or closing of files; clicks, change of line and navigation around text; usage of tools, and more. Generally, everything is stored in a log with date, time and an identifier for the user. As a complement it can contain the name of the class and/or the function where it was executed and the type of the event.

2.5. Capturing usage data

It is possible to command the IDE to capture usage data to have better understanding, to a low level, of the activities of the programmer [?]. Eclipse and Visual Studio are examples of IDEs that provide an API (Application Programming Interface) that allows to register all the commands and actions that are begin executed. It is not necessary to build an application to capture the data, for there are several options to collect it like Eclipse Usage Data Collector, Mylyn Monitor and Codealike. It is possible to start a study with existing data

from previous projects. An example is the available information in the Eclipse archives about the Usage Data Collector, a dataset of more than 1 million users and 2,000 million registered events.

2.6. Productivity metrics

Given that the available information about usage data can tell the history of activity of the programmer, one of the most used metrics is the total active time, without the time consumed in interruptions, like the study by Sanchez et al. (2015). The active time is also considered by the focus function of Codealike, which increases when more activity is registered. The focus level is a metric used by this tool to measure the productivity and tries to model the concentration of the user. Prolonged time without registered activity or time outside the IDE are identified as interruptions, being the number of interruptions another metric. When it is possible to classify the events by its type it can be used metrics like the editions per minute or selections per minute. It is also common to use debugging per minute as metric, like the research by Carter and Dewan (2010). Moreover, Sanchez et al. (2015) identified that during sessions without interruptions the proportion of edition events is superior than navigation events, so one of their conclusions is that it is a good indicator of productivity. This tell us that a productive programmer spends more time coding and less time exploring the code. Codealike also uses a classification of events to quantify the technical debt when a class or function has more navigation events and debugging than edition events, which indicates that the element can be a bottle neck for the project.

2.7. Research with usage data

Kersten and Murphy (2006) propose a tool that keep the context of the task being performed by the programmer and make it visible to help in the navigation around elements. The context is a graph of classes, modules and functions that are relevant for the current task and that are needed to complete it. Each element related to the task has a weighted value that is created with a model

CHAPTER 2: RELATED WORK

of the degree of interest on that element according to the task, and it is shown to the programmer a list of elements ordered by the value. In a field study with programmers, the authors obtained qualitative and quantitative results that indicate an increase of productivity when having context of the task in progress, specially in big systems with many programmers collaborating.

Fritz et al. (2007) did a research about the possibility of inferring whether the programmer has knowledge of the code or not by quantifying the degree of interest, which is the amount of recent activity that the programmer made on an element, similar to the work done by Kersten and Murphy (2006). The degree of interest increases with the frequency of interactions on an element and decreases when the interactions stop. In a field study, they monitored the activity of 19 programmers and applied a weekly survey for three weeks. They concluded that when the programmer has knowledge of code (according to the surveys) there is a high degree of interest, concluding that the usage data can be used to identify this phenomenon.

Carter and Dewan (2010) inquire about the possibility of identifying a programmer stuck or having problems, which is when there is no progress despite of the effort, under the hypothesis that the activity of the programmer will give out evidence of this state. During an experiment with students in a programming course, the participants were asked to indicate when they were facing problems during programming tasks and while usage data was being captured. From the data they obtained metrics of the number of edition, navigation and debugging events per minute, and with the data labeled where the programmer was stuck, they trained several machine learning algorithms. The best result was obtained with a decision tree, which correctly predicted the "in problems" state the 92% of the times, concluding that it is possible to identify this state with usage data.

Minelli et al. (2004) performed an analysis about the process of program comprehension and compared the results with the literature. Based on interaction data, they labeled the segments of the working sessions of programmers according to the activity, specifically in sectors where the programmer was navigating around classes, editing code, inspecting objects and reading the code. Then, they split the sessions in segments and quantified them according to the activity performed and concluded that the program comprehension activity was underestimated by previous research, for the results indicated that this activ-

CHAPTER 2: RELATED WORK

ity covers (in average) 65% to 90% of the working sessions, against a previous estimation of 35%.

Sanchez et al. (2015) used interaction data to perform an empirical analysis of the impact of work fragmentation on productivity, identifying lapses of time without recorded activity as interruptions and quantifying the productivity according to the number of edition and selection events, and the proportion of edition events against selections. They found that productivity decreases as the number of interruptions increases in a working session, and that in sessions with prolonged interruptions the productivity tends to be lesser in comparison with sessions with only short or none interruption. In session without interruptions the productivity is triplicated, but they are only the 2% of all the sessions of a programmer.

Chapter 3

Data and Methodology

In this Chapter we describe the data used in this work and the methodology used for knowledge extraction.

3.1. Description of the data

It is important to first describe the data in terms of the attributes, magnitude and context of extraction. In the following subsection we describe in detail the characteristics of the datasets, limitations and some inferences we can obtain.

3.1.1. Eclipse Usage Data Collector

The Usage Data Collector (UDC) dataset is a large compendium of information about interaction data from users of Eclipse, collected from December 2008 to August 2010, with the intention to keep track of how programmers are using the IDE. The framework listens to the events triggered by the user or the system, such as: edition and navigation commands; the startup of a plug in; or the closing of the platform. To be more specific, UDC collects information about loaded bundles, commands accessed via keyboard shortcuts, actions invoked via menu or tool-bars, perspective changes, view usage and editor usage. The UDC is a large dataset that contains information of around 1,800,000 users, and has a total of 2,323,233,101 unique rows with 5 attributes each. Table 3.1 shows a

description of the attributes.

This dataset only contains information of the execution of commands within the IDE and we do not have more context about the programmers or environments. Judging by the registry of events, there is a mix of programmers of different nature like Java SE, Java EE and Web developers. There are also programmers that use other kind of languages like PHP, SQL and Ruby. We assume that this data corresponds to programmers of all levels of expertise, from students to professionals.

Table 3.1: Attributes of the UDC dataset.

<i>Attribute</i>	<i>Description</i>
UserId	Unique number that identifies a user
What	The action of the event (deactivated, activated, opened, executed, etc.)
Kind	What kind of event was executed (workbench, view, command, etc.)
BundleId	Description of the event's package
BundleVersion	Version of the bundle's event
Description	Description of the event
Datetime	Date and time in UNIX format

We used the pre-processed version of the data that is published on Google BigQuery, by Murphy-Hill et al. [26]. This is an alternative version of the original UDC dataset, which is cleaned and preprocessed, so that the cleaning phase is simpler and focused on our needs. Due to the magnitude of the dataset we only worked on a fragment of it.

We took a subset of the data from 1,000 random users. We delimited the query to obtain only those events dispatched by the user, ignoring system events. We also ignored the BundleId and BundleVersion, leaving only the attributes UserId, What, Kind, Description and Datetime. From this query we extracted 4,321,349 unique events, which are around 0.18% of the whole dataset.

3.1.2. Codealike and ABB

Codealike [5] is a tool that monitors the activity of the user and later offers analytics and insight about the programmer’s productivity and working patterns. This tool is installed in the IDE (Eclipse and Visual Studio) and listens to the events executed by the user and system, similar to UDC. It captures almost the same kind of events like edition, navigation and tool usage. Shortly after the capture of events, the user can observe information derived from his activity through a website.

Corvalius (the Argentinian company that created Codealike) is collaborating with ABB, a multinational corporation operating mainly in robotics, power and automation technology areas. The ABB’s Software Engineering Research Group is using Codealike to obtain information from its developers with the objective of improving productivity and software quality. We had access to a dataset corresponding to the monitoring of 87 programmers between May and October of 2015 comprising 15,597,697 unique events. The main dataset that contains the registry of executed commands has the attributes described in the Table 3.2.

The data was extracted from Visual Studio via Codealike and the events seems to correspond to .NET programming languages, like C# or Visual Basic. Similar to UDC, it only contains a registry of executed commands and no information about the developers, except for the country of origin judging by the email domain that was provided. The programmers were invited to use Codealike at will, so the amount of data collected varies among users. Additionally, we had access to the *focus* level of the user, a feature of Codealike that measures the focus or concentration of programmers based on the amount of activity registered within the IDE. We assume that this data corresponds to professional programmers working for the same company and under similar circumstances. The latter meaning that they all have the same or similar equipment, methodology, tools and working hours. However, we can not be certain about any of these assumptions.

The actual description of the events is stored on a different file, so we use the values of the Category and Events attributes to extract it. In this case, all the attributes are needed. From hereafter this dataset will be referred as ABB.

Table 3.2: Attributes of the ABB dataset.

<i>Attribute</i>	<i>Description</i>
Username	Unique identifier of the user
Timestamp	Date and time of the execution
Event	Identifier of the event’s description
Category	Unique identifier of the event’s category

3.2. Methodology

We followed the steps described by the *Knowledge Discovery in Databases* process [7], a set of iterative steps composed of Selection, Preprocessing, Transformation, Data Mining and Interpretation. The Selection was covered by the last two sections and the following sections will describe the tasks performed during the Preprocessing and Transformation steps. In the latter Chapters we will give details about the Data Mining and Interpretation steps.

3.2.1. Preprocessing

During the Preprocessing stage we performed mainly two tasks: data cleaning and classification. The cleaning tasks are trivial, for the data is in overall good shape. However classifying the events is a complicated task that requires careful inspection and multiple iterations. The tasks involving data cleaning diverge between datasets, so we give details about this step separately. But the classification of events follows the same process for both datasets and is described afterwards.

I. UDC

First, we added a field with the duration of every event (time elapsing between one event and the next one, used to determine where interruptions take place and sessions end) and an ID to identify the different working sessions present on the data. For the latter, we sorted the data by UserId and Datetime. This was required because, by default, the user’s data is mixed and we need it not only chronologically correct but also sorted by users to tag the working sessions

of every user without interferences. We also filled the description for the events that indicate the activation or deactivation of the Eclipse’s workbench, which were the only ones with that issue.

II. ABB

The Preprocessing for ABB is also simple. As with UDC, we added a duration in seconds and an ID to every event. Then, we extracted the Description from a second dataset, according to the Event and Category, and created a new attribute. We changed the Description to lowercases and removed curly braces and other special characters.

An interesting feature of this dataset is that plenty of data seems to be somewhat systematic, with a certain event executed every 5 minutes or less. We do not know in what scenarios this kind of activity is recorded (it could be executed by the IDE) so we decided to remove those observations. Finally, we ordered the data by Username and Datetime.

3.2.2. Classification of events

To classify the events, we look into the description to see what it can tell us about the event. We worked with two kinds of classifications:

- A detailed classification to tag the nature of every event.
- A general classification to identify between Edition and Selection events.

Having two classifications will help us provide better answers to our research questions, for some of them require doing analyses in detail and others can be seen from a general perspective. The general classification is derived from the detailed classification, so we describe the latter first in the Table 3.3.

To assign a detailed classification we use the description. In both datasets the description has the format *path.to.class.ListenerClass*. It contains the path to the class that works as listener of the event and executes the required task. The packages and class name give out information about the nature of the event.

With that in mind, we created a set of rules that adds a classification to every

Table 3.3: Description of the detailed classification of events.

<i>Classification</i>	<i>Description</i>	<i>Examples</i>
Edit-Text	Text edition events	Copy, Paste and Delete.
Text-Nav	Events executed when navigating around text.	LineUp, LineDown and LineEnd.
High-Nav	Navigation of high level, like around classes and views.	GoToDefinition, NextTab and GoToSuperclass
Debug	Events executed during debugging sessions.	StepInto and StepOver.
Search	Events executed when searching for objects and text.	Search, Find and FindReplace.
Refactoring	Events executed when restructuring code.	Encapsulate, Rename and MoveField.
Testing	When testing (e.g. unit tests) is executed through a framework.	ExecuteTest and TestResults.
Control	Execution of version control tasks.	Compare, ViewChanges and CommitChanges.
Clean-Build	Tasks performed by the IDE to build and execute a solution.	Build and Run.
File	Events executed during the management of files.	Open, Close and SaveChanges.
Tools	Execution of specialized tools and plugins.	DatabaseDesigner, Codealike and UIDesigner.

event according to the name of the class and/or the name of the path. For example, in UDC we label as Text-Nav all the events that have *ui.edit.scroll* in the description, and in ABB we label as High-Nav all the events whose class name is *NextTab*. Sometimes it was required to create special rules for certain events but most of the time we were able to set rules for several of them. This was an iterative process that required careful inspection of the results.

Once we added the detailed type to the events, we proceeded to assign the general type. This is easily performed by setting as Selection all the events except those who has Edit-Text, Text-Nav or Refactoring in the detailed type; this set of events describe the kind of activities performed when editing code, while the rest involve navigation around classes, the opening of views and selection of

graphical elements. After we assigned a general and detailed type to the events we followed to the next step of the process.

3.2.3. Transformation

From the Transformation phase and on, the activities are the same for both datasets. The objectives of the transformation phase are identifying the working sessions and calculating a set metrics and time series for each. Then every session is decomposed into chunks or segments of smaller time than sessions.

First, we identify interruptions using the duration or time interval of every event. This is an important tasks and it is convenient to define the two kinds of interruptions we use in this work:

- *Interruption.* We define empirically an interruption as a pause of activity of duration ≥ 3 minutes. This is based on previous work where we observed that short interruptions lasted usually this long [9]. Shorter values would risk classifying periods of inactivity (such as a programmer reading source code) as interruptions.
- *Prolonged interruption.* Based on additional observations from this work, we defined a prolonged interruption as one lasting for more than 12 minutes. These thresholds are also supported by the Activity Theory models of Kaptelinin and Nardi [11]. This study presented work fragmentation at two different levels: actions and activities. Interruptions originated after a period of around three minutes of sustained attention to the previous action were considered when people were switching at the level of interactions with artifacts of people. Interruptions originated after a period of twelve minutes of sustained attention to a previous activity were considered when people were switching at the level of interactions with projects or topics.

To identify working sessions we look for interruptions as well, but only those that last for more than 4 hours. Any segment of activity surrounded by interruptions with that duration is labeled as a session. However, in order for the segment to be valid it must be of at least 30 minutes of duration.

Then, for every session we created several time series representing the execution of events by detailed classification. Taking the minute as time unit, we counted the number of events executed on every minute and created eleven time series, where the amplitude is the number of events. The interruptions were treated differently; they are also contained in a time series of its own but the amplitude is the duration of the interruption and every observation represents the minute of occurrence.

After that, the next step is to calculate a number of metrics for every session that measure the activity of the programmer. They can be seen from two perspectives, first the metrics for the characterization of interruptions:

- *Number of interruptions*: counts all the interruptions that occur in a development session.
- *Duration of interruption*: it is the time duration in minutes of the each interruption.

And the metrics that describe the productivity and activity:

- *Productive work time*: the duration of a development session, subtracting the duration of all the interruptions present in the session, to control for inactivity.
- *Number of edits per minute*: the total number of edits events, divided by the productive work time to control for length of the session. This is an indicator of user activity during the session.
- *Number of selections per minute*: it is the total number of selection events, divided by the productive work time. Also an indicator of activity.
- *Edit ratio*: the number of edits divided by the sum of edits and selections, as used by Kersten and Murphy [13]; an efficient developer spends less time exploring code and more time editing it.
- *Proportion of events*: it is the count of events by detailed type divided by the total of events executed in the session. There are a total of 11 values for this metric.

The last task of the transformation phase is to split the sessions into smaller activity frames to do analyses of the programmer’s activity in detail. For that we split the time series of every session into time segments of 3 to 5 minutes of activity that we will call chunks. It was necessary to recreate the time series and metrics for the chunks. After this, every user has a set of sessions (with their respective time series) and every session has a series of chunks.

We can observe some statistics about the resulting sessions in the Table 3.4. UDC contains much more sessions, but in average they are shorter (4.11 hrs) than the sessions in ABB (7.33 hrs). However, in both cases the actual time spent working in the IDE is much less, being in average of 64.29 minutes for UDC and 166.08 minutes in ABB. The average number of interruptions per session is larger in ABB (25.40) than in UDC (9.52), which is in relation to the size of the sessions, but in ABB the interruptions tend to be shorter (10.74 minutes).

Table 3.4: Statistics of sessions in UDC and ABB.

<i>Statistic</i>	<i>UDC</i>	<i>ABB</i>
Number of sessions	6,405	1,182
Number of observations	2,848,270	2,449,227
Number of users	621	69
Number of chunks	43,769	23,624
Avg. duration	4.11 hrs.	7.33 hrs.
Avg. productive time	64.29 min	166.08 min
Avg. of interruptions	9.52	25.40
Avg. duration of inte.	18.94 min	10.74 min

Chapter 4

Results

This section corresponds to the phases Data Mining and Interpretation of the Knowledge Discovery in Databases process. Here, we present two different analyses using the UDC and ABB datasets:

1. Analysis of the impact of interruptions of work in the programmers' performance and activities
2. An identification and characterization of the programmers' activities at low level (activities within 5 minutes) and high level, covering working sessions.

First, we present the analysis of the impact of interruptions in the following section.

4.1. Interruptions of work and productivity

In this section we present an analysis of the effects of interruptions of work in programmers' productivity, a phenomenon that has been previously investigated via field studies and to a wider spectrum of workers [9, 15, 6, 2] instead of focusing on the population of programmers like this work. These studies agree that interruptions of work can be detrimental to the programmer performance,

for every interruption demands a recovery or resumption time to restore the lost context.

To assess the impact of interruptions we use a set of metrics described in Chapter 3: edits per minute, selections per minute and edit ratio. We measure the effects on productivity using mainly the edit and selection events per minute as indicators. The edit ratio [13] might be also a good indicator, under the assumption that a productive programmer will execute more editing activities than selection or navigation events. However, this is not always the case, given that depending on the role of the worker, selection events can be equivalent to the usage of a designing or reporting tool.

The results presented in this section correspond to a replica of a published paper [25] in the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. That paper included the same analyses described in the next three subsections (4.2.1 to 4.2.3) but with a different dataset (Mylyn). During this work we improved some of the methodology and extended the results to include a detailed analysis of the Recovery Time (subsection 4.2.4). The extended version is now being submitted to the Journal of Software: Evolution and Process.

4.1.1. Relationship between the number of interruptions and productivity

As mentioned above, we use the metrics of edit, selection, and edit ratio as indicators of productivity. We first examine the number of edits and selections, and how their distribution varies in function of the number and type of interruptions.

We split the data in five groups: The first group contains all the sessions without interruptions (*none*). For the others groups, we have considered four ranges of number of interruptions delimited by their quartiles, being 2, 4 and 7 the first, second and third quartile respectively.

We observe that when facing zero interruptions the editions and selections per minute are greater than in sessions with one or more interruptions, and as more interruptions occur the metrics gradually decrease. There is a big difference between sessions with no interruptions and at least one in the edits per minute

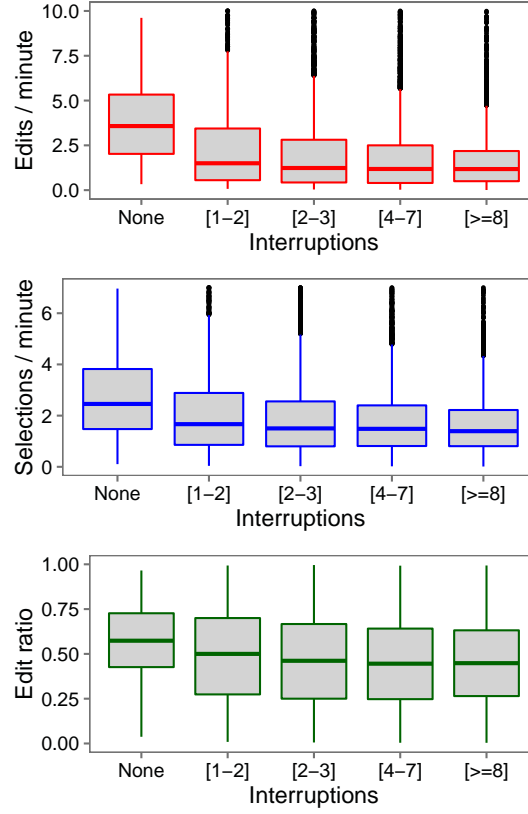


Figure 4.1: Boxplots showing the relation between the number of edits and selection events per minute, the edit ratio, and the number of interruptions.

metric.

As for the edit ratio, the median changes accordingly with the number of interruptions, decreasing when there are more and reaching the maximum when there are none. In addition, the size and form of the boxes indicate more variation in these results and the differences between groups is more subtle, judging by the small changes in the mean values.

Beyond visual inspection, we also quantify the statistical and the practical significance of these observations. First, all the differences observed are statistically significant with very low p-values (see Table 4.1) according to the Mann-Whitney U-test. This is not surprising, given the shape of the boxplots and the

CHAPTER 4: RESULTS

size of the samples. It is important to mention that the values from the effect size in the Table 4.1 are obtained by comparing the *none* group with the rest of the groups.

More importantly, we used *Cliff's delta* to measure the practical significance of these results in term of effect size [3]. *Cliff's delta thresholds* are defined as follows: negligible (< 0.147), small (< 0.33), moderate (< 0.476) and large effect otherwise. As shown in Table 4.1, we note that the effect size of the interruptions over the number of edits per minute is overall moderate, being larger for edits per minute and small in the edit ratio. Note that despite the differences in effect sizes, the differences are still significant and it is possible to reach the same conclusions.

Table 4.1: Effect size and significance of the relationship between number of editions per minute, selections per minute, edit ratio, and the number of interruptions on UDC data

	none	≤ 2	[3 – 4]	[5 – 7]	≥ 8
Edits					
median	3.58	1.5	1.23	1.15	1.18
mean	3.97	2.18	1.82	1.70	1.59
U-test (adjusted)	\hookrightarrow	$< 2.2\text{e-}16$			
Cliff's delta	\hookrightarrow	0.42	0.53	0.57	0.63
Selections					
median	2.54	1.66	1.5	1.5	1.4
mean	3.10	2.93	1.80	1.77	1.63
U-test (adjusted)	\hookrightarrow	$< 2.2\text{e-}16$			
Cliff's delta	\hookrightarrow	0.29	0.38	0.39	0.49
Edit ratio					
median	0.57	0.49	0.45	0.43	0.44
mean	0.68	0.56	0.55	0.54	0.57
U-test (adjusted)	\hookrightarrow	< 0.0001			
Cliff's delta	\hookrightarrow	0.19	0.24	0.28	0.26

4.1.2. Relationship between duration of interruptions and productivity

In this section we analyze whether or not the duration of interruptions is a factor in the relationship between interruptions and developer productivity. Under the hypothesis that prolonged interruptions require more time to recover from, we built two groups of sessions based on the proportion of long interruptions. As we

CHAPTER 4: RESULTS

described previously we consider that a prolonged interruption has a duration of ≥ 12 minutes, a decision based on the literature [9, 11].

For every session, we calculated the proportion of prolonged interruptions as the sum of interruptions with duration equal or greater than 12 minutes, divided by the total count of interruptions in that session. Then, we created 2 groups of sessions described as follows:

- *low*: the first group consists of sessions where the proportion of prolonged interruptions is < 0.50 ; this includes sessions for which the proportion of long interruption is zero, that is, sessions with only short interruptions. They are the 38.7% of all the sessions.
- *high*: the second group consists of sessions where half or more of the interruptions are prolonged (proportion ≥ 0.50). This represents 60.1% of the sessions.

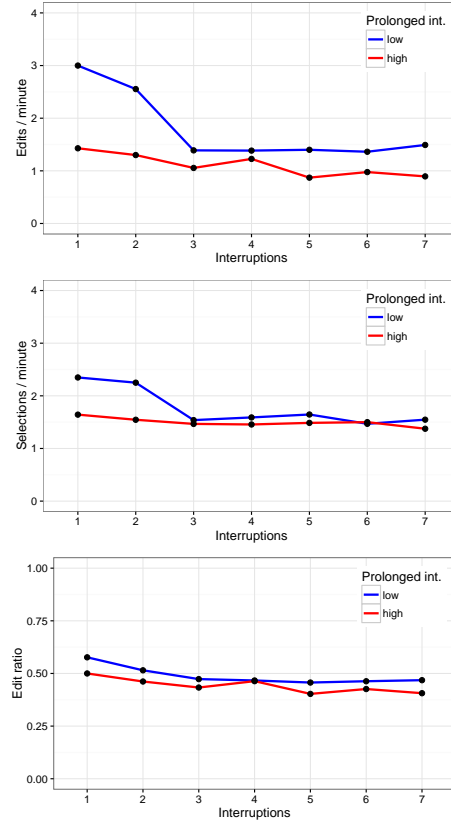
Following the conclusions on the last subsection, the number of interruptions has an impact on the metrics and should not be ignored. Furthermore, as the number of interruptions increases the difference between the groups minimizes. Thus we want to compare the proportions of prolonged interruptions among sessions that have the *same* number of interruptions. We hence split the *low* and *high* groups in subgroups of sessions which have the same number of interruptions. We consider groups of up to seven sessions, as larger groups tend to be smaller. We finally compare pairs of *low* and *high* groups that have the same number of interruptions, expecting higher productivity indicators in the *low* groups.

The Figure 4.2 shows the median of the metrics when grouping the sessions by the number of interruptions and the proportion of prolonged interruptions. According to our hypothesis, we expect sessions that have a low number of interruptions and a low proportion of prolonged interruptions to have higher rates of productivity, and the results mostly agree with this statement. First, we observe that when the number of interruptions increases the median of the metrics decrease until reaching a point (approximately between three and four interruptions) where the change is minimal and continues that way for the rest of the groups. Second, in the vast majority of groups the median value of the

CHAPTER 4: RESULTS

low group is greater than the *high* group.

Figure 4.2: Median values of edits and selection events per minute, and the edit ratio according to the number of interruptions and proportion of prolonged interruptions.



When having a small number of interruptions (between 1 and 3) the difference between sessions with a low or high proportion of prolonged interruptions is very noticeable, and the group of sessions with a low proportion has higher median values. As the number of interruptions increases the differences get smaller, but we observe that the group of low proportion of prolonged interruptions usually shows higher values. This applies on the edits and selections per minute, and to a lesser extent to the edit ratio, where the changes between groups (both by number of interruptions and proportion) are comparatively smaller.

We tested the statistical significance of this results and show the results in

CHAPTER 4: RESULTS

the Table 4.2. For every group according to the number of interruptions, we measured the effect between the low and high groups of sessions, with the Cliff's delta test and U-test. The groups with one or two interruptions have moderate size effects, in comparison with the other groups with more interruptions that have lower size effects; this matches our observations via visual inspection on a more defined difference between the low and high groups when having a small number of interruptions. Starting from (approximately) three interruptions, the distributions of the metrics are very similar.

Table 4.2: Effect size and statistical significance of the relationships between the groups of sessions by proportion of prolonged interruptions and the number of interruptions in UDC

Group	1	2	3	4	5	6	7
Edits							
U-test (adjusted)	1.9e-20	4.0e-13	0.0011	0.027	7.2e-08	0.055	6.5e-06
Cliff's delta	0.36	0.27	0.08	0.06	0.17	0.07	0.18
Selections							
U-test (adjusted)	2.9e-07	2.3e-11	0.32	0.12	0.0043	0.45	0.12
Cliff's delta	0.20	0.25	0.02	0.05	0.10	0.03	0.07
Edit ratio							
U-test (adjusted)	3.5e-05	0.033	0.033	0.20	0.004	0.14	0.0059
Cliff's delta	0.17	0.08	0.05	0.04	0.10	0.06	0.11

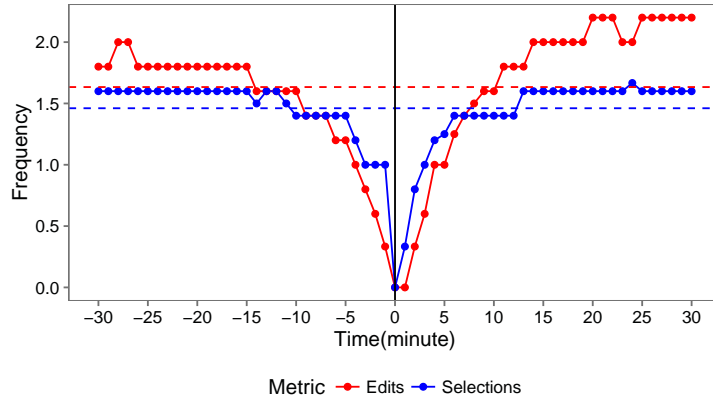
4.1.3. Local relationships between interruptions and productivity

The impact of work fragmentation could be more noticeable in the immediate minutes before and after an interruption occurs. On the one hand, after an interruption the programmer may carry out activities meant to recover the lost mental state, such as reading the code, debugging, reading notes and cues, and other resumption strategies [21]. Depending on the interruption, the recovery could last about 15 minutes, according to the reports from field studies [10, 28]. In the data from UDC, the sessions with duration of at least seven hours (commonly the working time of a software developer) have 11 interruptions in average; if we consider a recovery time of 15 minutes, the total time required to recover from the interruptions can be of two hours, which is about one quarter of the whole session.

On the other hand, before the interruption there is a preparation phase when the interruption is imminent or expected, e.g. lunch time, a meeting and other scheduled events. In this phase the programmer might leave notes or comments in the code to recover the context after the interruption [20]. Also, when the interruption happened because of a problem found by the programmer, he may try to solve it by reading the code, or using the debugger; after these resources are depleted, the next option is to ask to teammates or other external resources [14], which generates an interruption of the work. The activities prior to the interruption are commonly associated with selection and navigation tasks, such as reading the code and navigating to another classes or files to try to find an answer to the problem.

Having presented the global effect of the interruptions over the user productivity, we now focus on the local activity before and after interruptions. We take a maximum real time interval of 30 minutes around each detected interruption, obtaining a set of 97,984 time series. Then, we compute the median of all these subsequences as a generic local representation (Figure 4.3). We also plot with dashed lines the median values of edits and selections per minute in the sessions with interruptions in order to give more context to the observed values.

Figure 4.3: Local effect of an interruption in the user activity.



Below we describe some observations:

- *In the center*, we find the interruption point. There is clearly a negative effect on the time series, as the area before and after the interruption is

the area with the lowest activity. The activity is well below the median activity of time series with interruptions, showing that the effect is indeed more pronounced near interruptions.

- *On the right*, the trend of the time series increases steadily. We hypothesize that the programmer is immersing again into the programming task, increasing progressively the activity as represented by the number of edits and selections. It reaches the median activity (in average) 6 to 12 minutes after the interruption. Then rises further than the session median, which is not surprising, as we expect higher than median activity further away from interruptions.
- *On the left*, we observe that the number of events near the interruption also goes down well below the median value. This was more surprising at first to us. However we hypothesize that this might be because of two reasons: the programmer could have found a problem while coding, and at first would try to solve it by reading the code, switching out from the IDE, navigating the call stack and debugging, before going to ask another teammate (as documented e.g. by LaToza et al. [14]); this set of actions end up with an interruption and reduce the observed activity within the IDE. In the case when the interruption is imminent or expected, another possibility is that the programmer may make use of different suspension strategies such as writing physical notes, making a mental note or leaving a reminder cue on the code or window, as documented by Parnin and DeLine [20]. These activities reduce the activity before the gap and they are seen in the interaction data mostly as selection events. The latest can be better visualized in Figure 4.3, in which the UDC data has a significant decrease of edit events, being below the selection events approximately 10 minutes before the interruption.

I. Patterns of Interruptions

Given the overall activity pattern we noticed in the local analysis, we hypothesize that there are several kinds of interruptions, matching the scenarios observed in the literature: actual interruptions distracting the programmer from the task at hand, and switching tasks in case of being stuck in the current task.

CHAPTER 4: RESULTS

We hence looked for patterns in the interruptions. After applying clustering techniques over all the subsequences, we always found the formation of three recurrent patterns that show different effects of the interruption: neutral, positive and negative. The clustering was performed with the K-Medoids [27] technique, the Silhouette metric [24] to interpret and evaluate the results, the Dynamic Time Warping [12] as distance measure, and feature extraction techniques to reduce the dimensionality. For this reason, we classified empirically each interruption by its local effect.

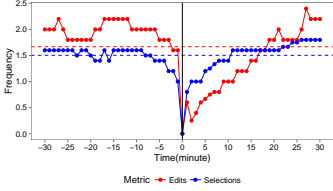
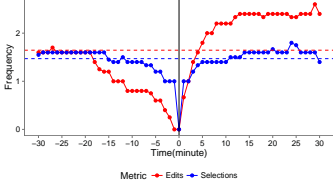
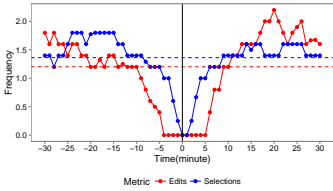
We did so by computing Cohen’s d on the quantity of edits before and after the interruption. To obtain a significant effect, we need the presence of activity both before and after the interruption. However, not all the interruptions meet this criteria: some are located close to the start or the end of a session, or too close to another interruption. In total, 53% of the interruptions had 30 minutes before and after the interruption and were selected for the analysis. The remaining 47% of the interruptions can not be used on this analysis due to the lack of time around them.

Table 4.3 shows the applied thresholds and the results. This local analysis shows that there are indeed three well-defined groups of interruptions, with the two largest of them having clear effects on the activity in the session. The 44% of the interruptions are positive and the negative interruptions constitute the 44%. The interruptions with a neutral pattern are the 12%.

The results in UDC indicate that after a positive interruption, both metrics tend to increase at a high rhythm until reaching the global average value. The edits per minute recover quickly (approximately 5 minutes after the interruption), but the selections per minute take a little longer to reach the global average, and this metric tend to be below the average for an extended period. It is important to notice that before the interruption happens the editions per minute metric is below the average for approximately 16 minutes prior to the interruption.

In contrast, when facing the effects of a negative interruption the metrics take longer to reach the median. Contrary to when facing a positive interruption, the editions per minute metric is below the selections per minute, and the former takes longer than the latter to reach the average value. Also, the editions per minute metric is above the average for the 30 minutes before the interruption.

Table 4.3: Local Effect of Interruption.

Effect	Pattern
<p><i>negative</i>: when the frequency of edit events decreases after the interruption (Cohen's $d < -0.2$)</p>	
<p><i>positive</i>: when the frequency of edit events increases after the interruption (Cohen's $d > 0.2$)</p>	
<p><i>neutral</i>: when there is no well defined effect before or after the interruption ($abs(d) \leq 0.2$)</p>	

In general, the effect of positive and negative interruptions seems to be reversed. The editions per minute describe better the effect and the selections do not show a major change. There is not a clear pattern during a neutral interruption.

4.1.4. Recovery Time after an interruption

In the last section we saw different patterns of activity around positive and negative interruptions. In particular, we observed that the edits per minute were higher after a positive interruption and the contrary after a negative interruption.

The presence of this pattern and the important difference observed in Figure 4.3 led us to investigate the effect more closely. To this aim, we did an exploratory study of the activities performed by programmers during the recovery time of an interruption. This is possible due to the high variety of events present in

CHAPTER 4: RESULTS

the UDC dataset, which can tell us a lot more about the type of activities that developers do. .

The goal of the exploratory study was to better understand whether the activity taken after an interruption were indicative of different types of behavior for each type of interruption (positive or negative), and also whether these different kinds of activities supported our earlier hypotheses explaining why the two patterns of interruptions were present. Assuming that different commands represented different types of activity, the hypothesis was that the frequency of a given activity (as represented by the commands used to carry it) would vary between the different types of sessions.

A first challenge that we encountered was that the high number of different commands made it hard to infer broader trends. Hence, we used the detailed classification described in Chapter 3. With these categories we continue with describing the hypotheses we made earlier when we observed the positive and negative patterns, namely that:

1. *Negative* interruptions (having higher overall activity before the interruptions rather than after it) could be indicative of actual interruptions. That is, a programmer is interrupted in his normal course of work, and needs to rebuild his or her mental context after the interruption, performing additional program comprehension before resuming programming.
2. On the other hand, *Positive* interruptions (with higher activity after the interruption) could be indicative of information-seeking behavior. Programmers, noticing that they are encountering a problem that they are not able to fix on their own may look for outside help (asking an expert, querying a question and answer site such as Stack Overflow, etc), and once a solution is found, resume their work in the IDE in a much more productive manner.

We would then expect that certain types of interruptions would feature some events more frequently than other ones. For instance, after a negative interruption, we would expect more program navigation events (indicative of program comprehension), while after a positive interruption, we would expect more programming activity since the problem that the programmer encountered was

CHAPTER 4: RESULTS

solved. In short, we expect that for some categories of events, there will be differences between the types of events. We put this hypothesis to the test by calculating the frequency of categories of events after positive and negative interruptions, and also compared it with the overall frequencies of the events. We describe how we computed these frequencies in the following.

First, we noticed that while some events are used by all users, other events are used heavily by a minority of users and not by others; this matches the observation of Murphy-Hill et al. [19] that many development tools are underused. For example, the event *copyLineDown* is more frequent than the *rename* event, but the former is only used by 2.4% of the users and the latest by 25.2% of them. As such, we calculated the frequency of event taking into account how commonly used they were.

We calculated the weighted average of every event based on the frequency of execution during the first phase of recovery time after a positive or negative interruption, and the percentage of users that make use of that event (so that events that are used by very few users do not distort the results). The weighted average was calculated as:

$$\bar{x} = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$

where x_i is the frequency of execution of the event i and w_i is the percentage of users that uses the event. This balances the average of execution for events used by a small number of users.

To get a baseline, the same frequency calculation was made for the entire dataset, without distinction of the location of events in the session. We then compared the weighted average changes to have evidence of what kind of events are more often executed after a positive or a negative interruption. The Table 4.4 shows the results, and is explained below.

After positive interruptions the events belonging to the Edit-Text, File and Refactoring categories are more frequent than in the average. In addition, these categories of events are less frequent than the average after a negative interruptions. Of these three, Edit-Text and Refactoring have the largest differences between positive and negative interruptions. This three categories are much more clearly associated with edition activities than program comprehension, and thus fit the previous hypothesis.

Table 4.4: Weighted frequency of execution after positive or negative interruptions.

Category	Positive	Negative	All
<i>Categories that are more frequent after a positive interruption</i>			
Edit-Text	34.75	22.35	26.62
File	10.56	9.53	10.01
Refactoring	1.29	0.84	1.01
<i>Categories that are more frequent after a negative interruption</i>			
High-Nav	7.84	10.14	8.74
Debug	3.08	7.81	6.35
Search	1.81	2.43	1.84
Clean-Build	0.42	0.62	0.56
Tools	0.40	0.59	0.50
<i>Categories that are less frequent during the recovery time</i>			
Text-Nav	14.20	12.89	16.13
Control	1.21	1.75	1.82

After a negative interruption, Debug, Search, High-Nav, Search and Clean-Build categories are all more common than average. The categories of Debug, Search, and High-Nav are clearly associated with program comprehension rather than program edition. The Clean-Build category is not strongly associated with program comprehension, but it is however associated with debugging activities.

Finally, the Text-Nav and Control categories are less common in both positive and negative interruptions than in the average. We were somewhat surprised that Text-Nav was less common in positive interruptions than the average, as we expected it to be associated with program edition behavior. We note that it is however more frequent in positive interruptions rather than negative interruptions. Additional exploration is necessary to understand the behavior of the Control category.

Summing up, we can conclude that edition-related activities are indeed more frequent after a positive interruption, while comprehension-related activities are more frequent after a negative interruption. These categories of events agree with our previous hypothesis that positive interruptions are related to information seeking behavior: programmers would interrupt their IDE activities when

encountering a roadblock, find the information elsewhere (from the Internet, an expert, etc), and return to the IDE with an increased productivity. This also agrees with the literature [21, 14]. The categories common after negative interruptions also agree with our hypothesis that negative interruptions are related to actual task switches where context needs to be rebuilt afterwards, an activity that involves program comprehension [16, 22].

Of course, this exploratory study does not allow us to confirm these hypotheses. However, the evidence we have discovered so far does not disprove them, and rather goes in the direction of our hypotheses, which means that our confidence in these hypotheses is higher as a consequence.

4.2. Activities and working patterns

There are numerous ways to identify the kind of activities and working patterns performed by programmers, like through observational studies [9], surveys [21], usage data [19] and activity logs, being the first the one that provides richer information. Usage and activity data has been widely used to analyze activity patterns from programmers as part of the Mining Software Repositories research area, but the kind of activities that can be assessed depends on the available data. In this case, our approach to identifying activities and working patterns is by analyzing the execution frequency of certain events.

4.2.1. Identifying activities

In the literature we can find evidence about the kind of activities performed by programmers, mostly from observational studies and usage data [14, 9, 16, 17]. We could have defined a set of activities based on the conclusions from these resources, but how can we assure that the same activities are present in our data? Do we have enough data to identify those activities? Is the same kind of activities identifiable in both datasets? These are some of the problems in establishing a set of activities based on related work. To avoid that we used clustering techniques to find them. This way we can find a set of common activities according to the data and the thresholds to differentiate between them.

During the Transformation phase we obtained a set of sessions and decomposed each into chunks of smaller size. Moreover, every one of these chunks has the proportion of execution of events by detailed type. So, the observations that were clustered are chunks and the attributes are the proportions. This means we have two datasets with 43,769 (UDC) and 23,624 (ABB) observations with 11 numerical attributes each.

As we do not know the number of activities we need an algorithm that chooses the number of clusters based on the data provided, like Mean Shift, Affinity Propagation and DBSCAN. We need an implementation that allows us to explore the values for each of the axis (attributes) of the found clusters, so we only considered Mean Shift and Affinity Propagation.

Mean Shift [4] discovers centers based on the density of samples of a set of regions whose size relies on the parameter *bandwidth*. It updates possible centroids so they are mean of the samples within a given region. Afterwards, the centers that are near-duplicate are summarized and then it presents the final set of centers. After some initial experiments, the results with this algorithm tend to contain a small number of clusters; it finds very common activities but ignores some rare that are absorbed by more general clusters. This is reflected in low average values of the distance within clusters, which is translated into bad Silhouette Coefficients, a metric described later.

Affinity Propagation [8] creates clusters by sending messages between all the data points; this message contains a value indicating how suitable is a point to be an exemplar, and the accumulated values of the messages are used to make a decision. This is performed until high-quality exemplars are found and corresponds to a cluster. We observed after some experiments that this algorithm tends to create a high number of clusters, providing all kinds of activities but some of them at many levels; this means that several clusters cover one activity (e.g. Programming) but at different levels of intensity (e.g. high usage of Programming and low usage of Programming). This can be seen as the contrary effect of the Mean Shift results and also with bad Silhouette averages. The main drawback of this algorithm is the high time-complexity of the order $O(n^2t)$, where n is the number of samples and t the number of iterations.

The Silhouette Coefficient [23] is a value per data point where high values are

obtained from well defined clusters. It is composed of two scores: a representing the mean distance between a sample and all other points of the same cluster, and b the mean distance between a sample and all other points in the next nearest cluster. So, the Silhouette Coefficient s of a point is obtained as:

$$s = \frac{b - a}{\max(a, b)}$$

This coefficient evaluates the compactness (how close are objects within the same cluster) and separation (how well-separated is a cluster from other clusters). It is a value between -1 and 1; a value of 1 indicates that the point is correctly clustered and a -1 indicates that it is probably contained in the wrong cluster. Values around 0 means the observation lies between two clusters. We use the average Silhouette Coefficient (also known as the Silhouette width) to compare between models and to conclude if there is an actual structure in the data.

The issues of implementing only one clustering technique were solved with the following approach:

1. First, cluster the observations with K-means into k clusters. The parameter k should be considerably big.
2. Then cluster the resulting centroids with Mean Shift selecting a bandwidth b .
3. Finally, label the observations according to the centroids of the second model.

By clustering first with K-means we approximate the number of clusters with an algorithm with lower time-space complexity. The value for k should be many times greater than the actual number of centers we expect to see. With this we try to cover all the probable activities, common and rare. The resulting clusters will be separated enough (and without much elements in between) so that they will not be summarized into one or two when applying the second clustering algorithm.

We still do not know what kind of activities are present in the data, so we need an algorithm that helps us discover them. For that, we cluster the centroids

Table 4.5: Activities found via clustering for UDC (left) and ABB (right).

<i>Activity</i>	<i>% of chunks</i>		<i>Activity</i>	<i>% of chunks</i>
Programming	43.41 %		Debugging	44.45 %
Navigation	38.05%		Programming	34.67 %
Debugging	6.69 %		Navigation	17.20%
Tool-usage	4.27%		Version	2.77 %
File-mgmt	3.80 %		Tool-usage	0.54 %
Version	2.88 %		Testing	0.37 %
Search	1.16 %			
Refactoring	0.47 %			

obtained from K-means with Mean Shift choosing a bandwidth b . To set the values for k and b we selected those that maximize the average Silhouette width. For ABB the parameters that maximize the metric are $k = 3000$ and $b = 0.41$. And for UDC the parameters should be $k = 2500$ and $b = 0.33$. With this approach we not only obtained acceptable clusters in terms of the activities they represent, but also got models with the best Silhouette width.

After we have the centers from Mean Shift, the next step is to add a label to each of them considering the values of the attributes. We took advantage that a high value for one of the attributes indicates that the activities of that center have heavy use of that event. Every center has a group of attributes with higher values than the rest; for example, a center labeled as Debugging has a value of 0.8 for the Debug attribute, and a value close to 0 for the rest, and a center labeled as Programming has higher values for the Edit-Text and Text-Nav attributes. The task involves manual work and the interpretation from the authors of what activity a cluster is modeling, so this could be a threat to the validity of the results.

The assigned label represents the activity that a center models and several centers could represent the same activity. The Table 4.5 shows the resulting activities for ABB and UDC.

4.2.2. Comparing activities between datasets

Five activities (Programming, Navigation, Versioning, Debugging and Tool-usage) are present in both datasets and the rest are unique. We can see more variety of activities in the UDC dataset, which we attribute to the different types of programmers and to the sort of events that can be captured from the IDEs.

There are some interesting differences. First, the Programming activity is common in both cases with a high percentage of activities (43.41% in UDC and 34.67% in ABB), and surprisingly Debugging is only common in ABB (44.45%), falling to the third place in UDC with only 6.69%. Both datasets contain a very similar amount of events classified as Debug, that is 68 for UDC and 76 for ABB. The common events of control like start debugging, step over and step into are present in both datasets, but the problem is in the 8 extra events available in ABB.

The ABB programmers, which are Visual Studio users, make heavy use of the Locals Window of Visual Studio that allows them to explore the state of the objects when debugging and change around processes and threads. Eclipse has a similar functionality in the Debug Perspective and captures an event when the programmer use it, but it does not support exploring different instances in the same fashion as Visual Studio. On top of that, in the ABB data the most common events of the type Debug are in fact those that allow to change the current process or thread. This could be the main cause for the differences of Debugging activities between datasets.

Another difference is that Navigation and Tool-usage activities have more incidence in the UDC data. First, in respect to Navigation, UDC users seems to perform more navigation around tabs and search for classes from the Package Explorer of Eclipse. In the other hand, ABB users commonly perform navigation around the hierarchy of classes by calling to definitions. The kind of navigation performed by UDC users is translated into much more navigation events of high level.

The amount of Tool-usage activities is related to the variety of programmers in the UDC data in contrast with ABB. In the former there are a group of users that are Web developers and execute events involving PHP, JavaServer Faces,

HTML and CSS. To a lesser extent, there are also SQL and J2EE (Java Enterprise Edition) developers. In ABB the small amount of Tool-usage activities is mostly related to the design of classes and architecture of the system in development, and also to connections to SQL databases. From this, we can infer that the programmers in ABB have mostly *back-end* roles and probably maintain an existing system that is supported by a multithreading paradigm, judging by our observations on the Debugging activities. However, we are unable to ascertain these assumptions.

4.2.3. Identifying patterns of sessions

During a working session, a programmer could go throughout several states or perform multiple activities. It might be possible to understand better this activities by using the information of the clusters we got in the previous subsection.

In this section we look for patterns of sets of activities (sessions) that might be common among programmers. Due to the nature of the data we expect to see more common patterns in the ABB data and more variety in the UDC data, for more kind of programmers are present in the latter.

Once we had the chunks clustered and labeled, we divided the chunks of each session into three groups of equal size representing the beginning, middle and end. We selected 4 activities that are present in both datasets (Programming, Navigation, Debugging and Version), to facilitate the comparison, and calculated the proportion of each at every phase. After that, every observation (session) had 12 attributes and proceeded to cluster them using the Affinity Propagation technique; this time we want to see all the possible clusters without limiting to a low number like when clustering with chunks.

We found 47 clusters in ABB and 98 in UDC, but we only work with the 5 more populated. The Figure 4.4 show the results for ABB and UDC. The Types (A to E) are ordered from the most to the less populated. This selection covers 35% of the sessions in ABB and 39% in UDC.

4.2.4. Comparison of working sessions patterns

For the sessions in ABB we have the following observations:

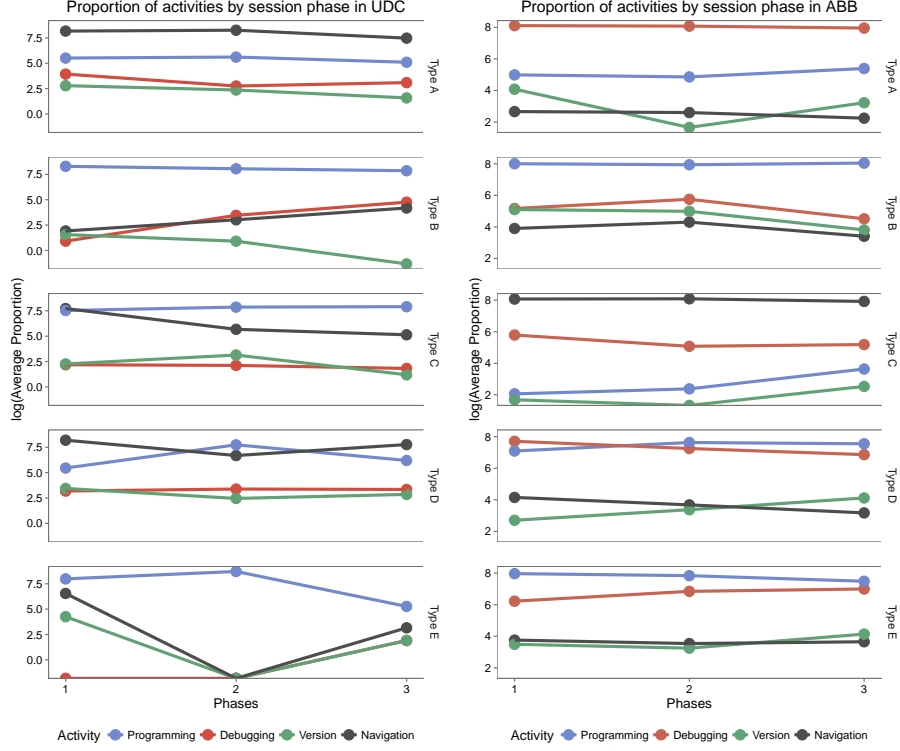
CHAPTER 4: RESULTS

- The most common pattern (Type A) involves mostly Debugging activities throughout the session and its followed by Type B, where the most common activity is programming throughout the session as well; it could be an effect of the two more frequent activities.
- The pattern described by the Type C is mostly composed of Navigation activities followed by Debugging. This can be related to program comprehension sessions [16] or bug fixing due to the low amount of Programming.
- Types D and E have a fair amount for all the activities throughout the sessions and do not show notorious changes. Also, both Debugging and Programming are executed frequently, whence we relate these two patterns to multitasking sessions.
- Versioning tasks are common during all the session when Programming is also a common. However it is only common at the begging or ending of the session when the Programming tasks are low. It could be that during Programming activities the user reaches more checkpoints that should be committed.

For the sessions in UDC we have the following observations:

- We relate Types A, C and D to multitasking sessions. The activities do not suffer sudden changes on any of the three phases.
- Type B keeps a steady amount of Programming throughout the session and there is a gradual increase of Debugging and Navigation until reaching a peak at the end. This is an interesting pattern that could be related to an increase of testing and proof of concept towards the end of the session.
- The sessions with a pattern of the Type E lacks completely of Debugging activities, while Programming is very common. Navigation and Version are only common at the beginning and end. This kind of sessions could be related to novice Programmers that instead of performing Debugging activities usually print to screen certain values to inspect the behavior of the program, also called tracing [18, 1].
- Although Debugging is a rather uncommon activity, it is frequently executed (to some extent) during the most common working session patterns.

Figure 4.4: Cluster of sessions in UDC (left) and ABB (right).



4.2.5. Validation of the results

To validate the results from the clustering of chunks and sessions we measure the Silhouette Width of the observations. The Silhouette Width is a number between 1 and -1; a value of 1 means that the observation is very well clustered and the contrary for a value of -1. If the value is close to 0 it means that the observation is between two different clusters.

In ABB the average Silhouette Average of the clusters of chunks is 0.55 and for UDC 0.38. These numbers represent a reasonable structural definition of the clusters. We can see in the Figure 4.5 the graphical representation of the silhouette of each cluster of chunks for ABB (right) and UDC (left). In average, the clusters in ABB are better defined but in both cases we can spot some observations wrongly clustered (negative values) that can also be outliers.

In the case of the clustering of sessions, we have an average silhouette of 0.35 for ABB and 0.40 for UDC, indicating a somewhat weak structure but still identifiable. This can be observed in the Figure 4.6. In UDC a big number of the observations of the cluster Type C are outliers and the rest of the clusters are better defined. In ABB clusters Type A and C show good shape but the rest of the clusters have smaller averages.

Overall, we can see a clear pattern in the clustering of chunks and sessions. We believe the patterns are strong enough to support our observations and the fact there are patterns in the data. The data from ABB is stronger in this sense, probably due to the fact that it all comes from developers. In the other hand, the UDC data contains more variety of users and provokes a variety of activities and practices that are harder to cluster.

Figure 4.5: Silhouette analysis of the clustering of chunks for UDC (top) and ABB (bottom).

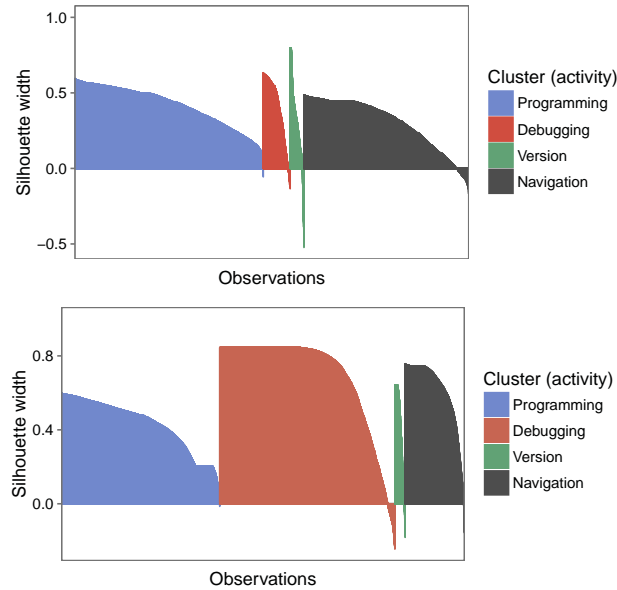
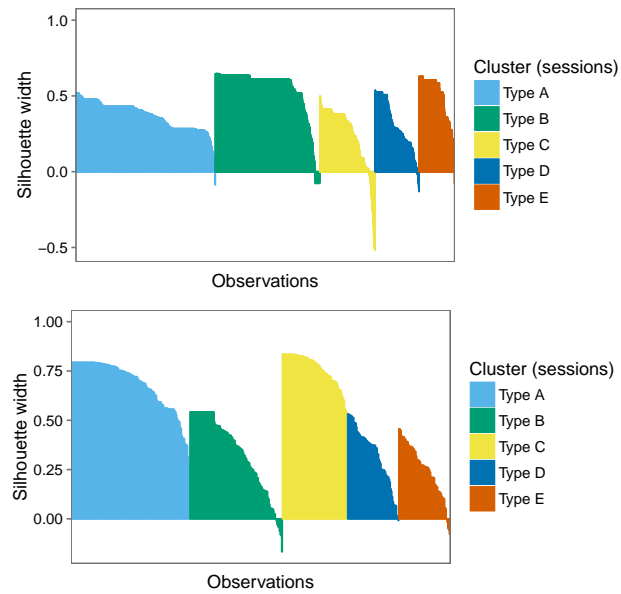


Figure 4.6: Silhouette analysis of the clustering of sessions for UDC (top) and ABB (bottom).



Bibliography

- [1] Ahmadzadeh, M., D. Elliman, and C. Higgins (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin* 37(3), 84–88.
- [2] Aral, S., E. Brynjolfsson, and M. Van Alstyne (2012). Information, technology, and information worker productivity. *Information Systems Research* 23(3-part-2), 849–867.
- [3] Cohen, J. (1994). The earth is round ($p < 0.5$). *American Psychologist* 49(12), 997–1003.
- [4] Comaniciu, D. and P. Meer (2002). Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence* 24(5), 603–619.
- [5] Corley, C. S., F. Lois, and S. Quezada (2015). Web usage patterns of developers. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 381–390. IEEE.
- [6] Czerwinski, M., E. Horvitz, and S. Wilhite (2004). A diary study of task switching and interruptions. In *Proceedings of CHI 2004*, pp. 175–182.
- [7] Fayyad, U., G. Piatetsky-Shapiro, and P. Smyth (1996). From data mining to knowledge discovery in databases. *AI magazine* 17(3), 37.
- [8] Frey, B. J. and D. Dueck (2007). Clustering by passing messages between data points. *science* 315(5814), 972–976.

BIBLIOGRAPHY

- [9] González, V. M. and G. Mark (2004). "constant, constant, multi-tasking craziness": managing multiple working spheres. In *Proceedings of CHI 2004*, pp. 113–120.
- [10] Iqbal, S. T. and E. Horvitz (2007). Disruption and recovery of computing tasks: field study, analysis, and directions. In *Proceedings of CHI 2007*, pp. 677–686.
- [11] Kaptelinin, V. and B. A. Nardi (2007). Acting with technology: Activity theory and interaction design. *First Monday* 12(4).
- [12] Keogh, E. J. (2005). Exact indexing of dynamic time warping. *Knowledge and Information Systems* 7, 358–386.
- [13] Kersten, M. and G. C. Murphy (2006). Using task context to improve programmer productivity. In *Proceedings of FSE 2006*, pp. 1–11.
- [14] LaToza, T. D., G. Venolia, and R. DeLine (2006). Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pp. 492–501. ACM.
- [15] Mark, G., D. Gudith, and U. Klocke (2008). The cost of interrupted work: more speed and stress. In *Proceedings of CHI 2008*, pp. 107–110.
- [16] Minelli, R., A. Mocci, M. Lanza, and T. Kobayashi (2014). Quantifying program comprehension with interaction data. In *2014 14th International Conference on Quality Software*, pp. 276–285. IEEE.
- [17] Murphy, G. C., M. Kersten, and L. Findlater (2006). How are java software developers using the eclipse ide? *IEEE software* 23(4), 76–83.
- [18] Murphy, L., G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander (2008). Debugging: the good, the bad, and the quirky—a qualitative analysis of novices’ strategies. In *ACM SIGCSE Bulletin*, Volume 40, pp. 163–167. ACM.
- [19] Murphy-Hill, E., C. Parnin, and A. P. Black (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38(1), 5–18.
- [20] Parnin, C. and R. DeLine (2010). Evaluating cues for resuming interrupted programming tasks. In *Proceedings of CHI 2010*, pp. 93–102.

BIBLIOGRAPHY

- [21] Parnin, C. and S. Rugaber (2011). Resumption strategies for interrupted programming tasks. *Software Quality Journal* 19(1), 5–34.
- [22] Parnin, C. and S. Rugaber (2012). Programmer information needs after memory failure. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 123–132. IEEE.
- [23] Rousseeuw, P. J. (1987a). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20, 53–65.
- [24] Rousseeuw, P. J. (1987b). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20, 53–65.
- [25] Sanchez, H., R. Robbes, and V. M. Gonzalez (2015). An empirical study of work fragmentation in software evolution tasks. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 251–260. IEEE.
- [26] Snipes, W., E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. R. Nair, and D. Shepherd (2015). *Analyzing Software Data*, Chapter A Practical Guide to Analyzing IDE Usage Data. Morgan Kaufmann.
- [27] Struyf, A., M. Hubert, and P. Rousseeuw (1997, 2). Clustering in an object-oriented environment. *Journal of Statistical Software* 1(4), 1–30.
- [28] Van Solingen, R., E. Berghout, and F. Van Latum (1998). Interrupts: just a minute never is. *IEEE software* (5), 97–103.