Unveiling process insights from refactoring practices

João Caldeira [iD] [a], Fernando Brito e Abreu [iD] [a], Jorge Cardoso [iD] [b,c], José
Pereira dos Reis [iD] [a]

[a]*Iscte - Instituto Universitário de Lisboa, ISTAR-Iscte, Lisboa, Portugal*
[b]*CISUC, Dept. of Informatics Engineering*
University of Coimbra, Portugal
[c]*Huawei Munich Research Center, Germany*

Abstract

Context: Software comprehension and maintenance activities, such as refactoring, are said to be negatively impacted by software complexity. The methods used to measure software product and processes complexity have been thoroughly debated in the literature. However, the discernment about the possible links between these two dimensions, particularly on the benefits of using the process perspective, has a long journey ahead.

Objective: To improve the understanding of the liaison of developers' activities and software complexity within a refactoring task, namely by evaluating if process metrics gathered from the IDE, using process mining methods and tools, are suitable to accurately classify different refactoring practices and the resulting software complexity.

Method: We mined source code metrics from a software product after a quality improvement task was given in parallel to **(117)** software developers, organized in **(71)** teams. Simultaneously, we collected events from their IDE work sessions **(320)** and used process mining to model their processes and extract the correspondent metrics.

Results: Most teams using a plugin for refactoring ***(JDeodorant)*** reduced software complexity more effectively and with simpler processes than the ones that performed refactoring using only ***Eclipse*** native features. We were able to find moderate correlations **(≈43%)** between software cyclomatic complexity and process cyclomatic complexity. Using only process driven metrics, we computed ≈30,000 models aiming to predict the type of refactoring method

Email addresses: `jcppc@iscte-iul.pt` (João Caldeira [iD]), `fba@iscte-iul.pt`
(Fernando Brito e Abreu [iD]), `jcardoso@dei.uc.pt` (Jorge Cardoso [iD]),
`jvprs@iscte-iul.pt` (José Pereira dos Reis [iD])

arXiv:2010.15692v1 [cs.SE] 29 Oct 2020

(automatic or manual) teams had used and the expected level of software cyclomatic complexity reduction after their work sessions. The best models found for the refactoring method and cyclomatic complexity level predictions, had an accuracy of **92.95%** and **94.36%**, respectively.

Conclusions: We have demonstrated the feasibility of an approach that allows building cross-cutting analytical models in software projects, such as the one we used for detecting manual or automatic refactoring practices. Events from the development tools and support activities can be collected, transformed, aggregated, and analyzed with fewer privacy concerns or technical constraints than source code-driven metrics. This makes our approach agnostic to programming languages, geographic location, or development practices, making it suitable for challenging contexts such as in modern global software development projects. Initial findings are encouraging, and lead us to suggest practitioners may use our method in other development tasks, such as, defect analysis and unit or integration tests.

Keywords: Software Complexity, Software Process Complexity, Software Development Process Mining, Refactoring Practices

---

1. Introduction

> *"...All things - from the tiniest virus to the greatest galaxy - are, in reality, not things at all, but processes..."*[1]
>
> —Alvin Toffler(1928-2016)[2]

A process[3] is *"a series of actions taken in order to achieve a result"*. In many business areas, either on delivering products and/or services, the quality of the outcome is very often related with the process followed to build it [1, 2, 3]. This is expected to be no different in the software development domain. Therefore, to fully comprehend how software quality and improved maintainability are achieved, one should look carefully to the process perspective to complement any code related analysis [4].

Software development is intrinsically a process and, accordingly, it is a blend of activities performed by developers, often working from different

---

[1]In *"Future Shock"*, Penguin Random House, New York, 1970.

[2]American writer, futurist, and businessman known for his works discussing modern technologies, including the digital and the communication revolutions, with emphasis on their effects on cultures worldwide.

[3]Adapted from https://dictionary.cambridge.org/dictionary/english/process

locations and using a multitude of languages, tools and methodologies in order to create a new product or maintain an existing one [4]. Since the early days of software development, it was understood that programming is an inherently complex and error-prone process, and to fully understand it, we should mine, in a timely and proper manner, all facets of that process [5]. Any relevant insights one may obtain should therefore originate from the activities and/or artifacts recorded in software repositories during the development life cycle.

Studies on estimating the effort to develop a certain artifact, the identification of software defects, the prediction of time to solve bugs or on software comprehension, and the detection of refactoring opportunities, are amongst the most common use cases for those repositories [6, 7, 8, 9, 10, 11, 12].

Refactoring on its own is still a very challenging activity. The identification of components to refactor and the forecast of which methods to embrace continue to be relevant topics for research [13, 14, 15, 16]. These challenges emerge partially due to the significant functionality limitations software repositories contain and the type of data they use [17].

Some authors confirm that developers perform refactoring tasks manually more frequently than automatically [12]. Furthermore, it has been observed, in a real-life scenario, that refactoring can be harmful when done manually, using only IDE native features or simply driven by developers' skills, as it may introduce non-expected defects in the code [18].

On trying to comprehend software development processes, including refactoring practices, many data sources, methods, and tools have been used with validated outcomes, but some others are yet to be fully exploited [19]. For example, since Version Control Systems (VCS) are widely used by developers, researchers get easy access to historical data of many projects and use file-based VCSs as the primary source of code evolution data [20]. Although it is often convenient to use such repositories, research-based on VCS data is imprecise and incomplete [17].

As such, answering questions that correlate code changes with other activities (e.g., test runs, refactoring) is often unfeasible. Several reasons may contribute to it, as for instance:

- developers may not commit all their tests and/or refactorings;

- there are many ways to refactor one version of the code, therefore it is important to determine the refactoring activities sequences and frequencies;

- often we cannot distinguish if a refactoring was done manually or through a tool, just by comparing source code snapshots [21].

3

1.1. Code vs. Process Analysis

Most published work on software quality-related issues is based on source code metrics, especially on Java systems [22, 23, 24]. Tools for collecting those metrics upon other frequently used languages, such as JavaScript or Python, are often not available, which expose well the difficulties to reproduce the same research on projects having diverse languages. In case those metric collection tools exist, they often require to share the source code with third-party organizations [25], particularly on cloud-based platforms. Such scenarios raise privacy and ownership issues on sensitive data. Source code obfuscation does not mitigate this problem because developers need to keep code semantics for interpreting the metrics in context.

Instead, mining the developers' activities and behaviors, the same is to say, to mine their development process fragments, may be a more feasible approach since it is not specific to any programming language, geographic location or development methodology followed.

Event data can be obfuscated without losing the process structure and coherence, therefore, whoever is responsible to analyze the logs can apply algorithms to discover process models in very similar ways as if the logs were not obfuscated [26]. In other words, events from the development tools and support activities can be collected, transformed and aggregated with fewer privacy concerns and technical hurdles. As such, it has been pointed out that software development event logs can be used to complement, or even replace, source code data in software development analytics-related tasks [27].

1.2. Contributions

It is frequent to find software prediction models using source code and ownership metrics [16]. However, periodically this data is not easily accessible or has imprecisions. Nowadays, development teams use a diversity of languages, methodologies and tools, therefore, the collection and aggregation of data from software projects remains a challenge. Additionally, process metrics have been found to be good predictors for modeling software development tasks [28].

Thus, we proposed earlier [29] and are now evaluating deeper the use of process metrics gathered from the IDE (Integrated Development Environment), as a way to enhance existing models or eventually, build new ones.

Software product and process metrics have long been proposed, as well as techniques for their collection [30, 31, 32, 19, 33, 34, 35]. However, the association between product and process dimensions is only marginally discussed in the literature [36]. In order to improve our understanding on the liaison between the type of development activities executed and the resulting software product characteristics, namely to ascertain if developers' behavior has

4

an impact on software product quality, we collected data during a software quality improvement task (application of refactoring operations) given to 71 development teams. Regarding developers' behavior, we recorded all events corresponding to the activities/tasks/operations team members performed within their IDE and used those events to mine the underlying process and extract their metrics. Regarding software quality, we collected complexity metrics before and after the refactoring actions took place. The main objectives for this work are, therefore:

- to assess the use of software process metrics to facilitate and improve the analysis and predictions on refactoring tasks and/or other generic software activities;

- to evaluate a possible association between the complexity of the produced code and developers' practices in different refactoring tasks;

- to build classification models for refactoring practices using only process metrics and assess the prediction accuracy of such approach.

The rest of this paper is organized as follows: section 2 provides background related to the research area and emphasizes the need for the followed approach; subsequent section 3 outlines the related work; the research methodology and the study setup are presented in section 4; the results, the corresponding analysis and implications can be found in section 5 and threats to validity are discussed in section 6; the concluding comments and the outline for future work are produced in section 7.

2. Background

Empirical software engineering and software analytics are now mature research areas with substantial contributions to the software development best practices [37]. The knowledge base created to support those achievements took a great advantage from the experience gathered on analyzing past software projects. Based on the maturity obtained, it was possible to derive several models to measure software complexity, effort and relationships.

2.1. Early models

Lines of Code(LOC). The identification and quantification of software size/defect relationship did not happen overnight. The first known "size" law, saying the number of defects D was a function of the number of LOC; specifically, D = 4.86 + 0.018 * i, was the result of decades of experience and was presented by Fumio Akiyama [38].

Cyclomatic Complexity. One of the most relevant propositions to assess the difficulty to maintain software was introduced by Thomas McCabe when he stated that the complexity of the code was more important than the number of *LOC*. He argued that when his "cyclomatic complexity" metric was over 10, the code is more likely to be defective [39]. This metric, underpinned by graph theory, went through thorough validation scrutiny and then became the first software metric recognized by a standardization body, the NIST [31], what makes it even more relevant in the context of this journal.

Halstead Complexity. On trying to establish an empirical science of software development, Maurice Howard Halstead, introduced the Halstead complexity measures [40]. These metrics, which are computed statically from the code, assume that software measurement should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. Halstead's metrics were used, among other things, to assess programmers' performance in software maintenance activities (measured by the time to locate and successfully correct the bug) [41].

Effort Estimators. Later, Barry Boehm proposed an estimator for development effort that was exponential on program size: $effort = a * KLOC^b * EffortMultipliers$, where $2.4 \leq a \leq 3$ and $1.05 \leq b \leq 1.2$ [42].

Henry and Kafura Metrics. These two authors defined and validated a set of software metrics based on the measurement of information flow between system components. Specific metrics are defined for procedure complexity, software modules complexity, and module coupling [43].

The above models were the foundation knowledge for what is nowadays often categorized as Software Development Analytics [44]. However, current development methods, tools and data repositories are very different from the past. Back in those years, software developers were mainly using a text editor and a compiler. Software projects were essentially built employing a single programming language, following a fairly simple development methodology and the developers were rarely located in different geographies or across multiple time zones. These workspace conditions have changed.

2.2. Modern Days

In 2019, JetBrains[4] polled almost 7000 developers about their development ecosystem. Results show that more than 30 different programming languages are being used and confirmed that web back-end, web front-end and mobile applications are the type of applications mostly developed, with figures of 60%, 46% and 23%, respectively. It was unanimous the adherence of cross-platform development frameworks and 80% said they use any type of source code collaboration tool, 75% use a standalone IDE and 71% use a lightweight desktop editor. Almost 50% said they use continuous integration/delivery (CI/CD) and issue tracking tools. Less than 15% responded that they use any sort of static analysis, code review and in-cloud IDE tools. Table 1 presents the key takeaways from the mentioned survey.

In summary, currently, a software development ecosystem has to deal with at least the following facets:

- **Multi-Language Ecosystem.** According to a recent work about multi-language software development [45], the authors present evidences that non-trivial enterprise software systems are written in at least 7 programming languages and, a previous work showed that in the open source world alone, the average is 5 languages per project. Among these, one may find general-purpose languages(GPL) such as Java or C# and also domain-specific languages(DSL) like SQL and HTML, and cross-language links are also quite common, meaning some code artifacts are shared between languages. As a result, developers confirm they find more problems in activities such as implementing new requirements (78%) and in refactoring (71%).

- **IDE Evolution.** A substantial change was carried in the Integrated Development Environments (IDEs). Software development moved away from the early days of the code editor. As confirmed by the Jetbrains poll, developers now use powerful platforms and frameworks which allow them to be more productive on their jobs. This results from the combination of different software development life cycle activities, such as: requirements elicitation, producing analysis and design models, programming, testing, configuration management, dependencies management or continuous integration into one single tool such as `Eclipse`, `IntelliJ IDEA`, `Netbeans` or `Visual Studio Code`. These tools support the needs of different stakeholders, as they embed a myriad of

---

[4]https://www.jetbrains.com/lp/devecosystem-2019/

Table 1: Survey Key Takeaways*

Findings

Programming Languages Overall Results	
Java	The most popular primary programming language
JavaScript	The most used overall programming language
Go	The most promising language as 13% said they will adopt it
Python	Most studied language as 27% said they used it in the last 12 months

Languages used in last 12 months

JavaScript(69%), HTML/CSS(61%), SQL(56%), Java(50%), Python(49%)

Shell Scripting(40%), PHP(29%), TypeScript(25%), C#(24%), C++(20%)

Development Environments(Operating Systems)

Windows(57%), macOS(49%), Unix/Linux(48%), Other(1%)

Type of Application Development

Web Back-End(60%), Web Front-End(46%), Mobile(23%), Libraries(14%)

Desktop(12%), Other Back-End(16%), Data Analysis(13%), Machine Learning(7%)

Type of Tests Used

Unitary(71%), Integration(47%), End-to-End(32%), Other(2%), Don't Test(16%)

Targeted Mobile Operating Systems & Frameworks Used

Android(83%), iOS(59%), Other(3%)

React Native(42%), Flutter(30%), Cordova(29%), Ionic(28%), Xamarin(26%)

Regularly Used Tools

Source Code Collaboration Tool(80%), Standalone IDE(75%)

Lightweight Desktop Editor(71%), CI/CD Tool(45%), Issue Tracker(44%)

Static Analysis Tool(13%), Code Review Tool(10%)

*All values(%) represent the percentage of affirmative respondents

plugins available in their marketplaces. These plugins are not just available, they are properly customized for specific users/purposes, such as for modellers, programmers, testers, integrators or language engineers.

- **Low Code and No Code Paradigms.** Modern software development practices make consistent use of both approaches. They enable faster development cycles requiring little to no coding in order to build and deliver applications and processes. Low-code development platforms are seen as advanced IDEs which employ drag-and-drop software components and visual interfaces to replace extensive coding. With

high-level visual modeling languages, they provide higher levels of abstraction that allow a major reduction in hand-coding to develop an application [46]. In the extreme case we have no-code development where, by definition, textual programming is banned, giving rise to the so-called *citizen developers*. The most notable examples are online application generators (OAGs) that automate mobile and web app development, distribution, and maintenance, but this approach is claimed to be pledged with security vulnerabilities [47]. This paradigm shift in software development may also require a change in the way we assess critical properties of a software project, such as, quality, maintainability, and evolvability.

- **Global Software Development.** The aforementioned IDE platforms facilitated collaboration and the adoption of Global Software Development (GSD). Nowadays, a single software project often has developers, testers and managers located in different time zones and distinct world regions or countries [48].

- **CI/CD and DevOps.** Continuous Integration and Continuous Deployment (CI/CD) have seen an incremental usage in the last few years. However, efficient CI/CD pipelines are rare, particularly in the mobile apps world where developers seem to prefer the execution of *ad hoc* tasks [49]. Whilst CI/CD focuses more on the automation of tools along a defined software life cycle, DevOps has major concerns with the responsiveness, responsibilities and processes within the development, the deployment and the operational phases of software projects. Keeping these intertwined processes compliant with organizational rules is therefore a persistent requirement.

- **Resource Coordination.** It is still one of the fundamental problems in software engineering [50] and it can be characterized as a socio-technical phenomenon. Understanding the dependencies between development tasks and discover teams' behaviours continues to be a challenge in resource allocation and coordination of modern software projects.

Software product repositories have many limitations in terms of the process data they handle. For example, these repositories usually deal only with source code and do not track the developers' geographic location, their workflows within the IDE nor the developers' environment characteristics. A complete repository of process related data with the communications, activities, decisions and actions taken by developers, testers and project managers,

are, most of the time, if not always, neglected when the goal is to study a development process. Usually, even if the authors claim they are studying a process, they are often doing it using only artifact related data [36].

With the existing diversity of languages, methodologies, tools and the fact that resources are now distributed across the world and originate from multiple cultures with different skills, it is somewhat an anachronism to keep using old methods to assess, for example, complexity or build cross-cutting analytical models in current software projects. New approaches, supporting multi-languages, being multi-process aware, and keeping geography diversity transparent are called for, such as our pioneering approach for mining of software development processes based on the IDE event logs. That approach, dubbed Software Development Process Mining [29], allows reversing engineer a complete software development process, just a process fragment or simply *ad hoc* activities performed by developers, by mining event logs taken from real software development activities.

3. Related Work

To address the incompleteness of data sources related with software repositories, we strongly believe that Software Development Process Mining based at least on the IDE(but not limited to) can play that role and Process Mining tools and methods can be the vehicles to achieve that goal. Many authors have followed similar paths, bringing not only evidences for its usefulness but also valid contributions to improve established methods.

A decade ago, [51] mined software repositories to extract knowledge about the underlying software processes, and [52, 53] have learned about user behavior from software at operations. [54] was able to extract events from `Eclipse` and have discovered, using a process mining tool, basic developers' workflows. Some statistics were computed based on the activities executed and artifacts edited.

[55] presented an application of mining three software repositories: team wiki (used during requirement engineering), version control system (development and maintenance) and issue tracking system (corrective and adaptive maintenance) in the context of an undergraduate Software Engineering course. Experimentation revealed that not only product but process quality varies significantly between student teams and mining process aspects can help the instructor in giving directed and specific feedback. However, in this case, IDE usage mining was not contemplated.

The working habits and challenges of mobile software developers with respect to testing were investigated by [49]. A key finding of this exhaustive study, using 1000 Android apps, demonstrates that mobile apps are still

tested in a very ad hoc way, if tested at all. A another relevant finding of this study is that Continuous Integration and Continuous Deployment (CI/CD) pipelines are rare in the mobile apps world (only 26% of the apps are developed in projects employing CI/CD) - authors argue that one of the main reasons is due to the lack of exhaustive and automatic testing. Therefore, distinguishing during development sessions the type of tests being done can contribute to the overall software quality.

[56] explored if one can characterize and identify which commits will be reverted. An identification model (e.g., random forest) was built and evaluated on an empirical study on ten open source projects including a total of 125,241 commits. The findings show that the 'developer' is the most determinant dimension of features for the identification of reverted commits. This suggests that assessing developers behaviors can lead to better understand software products quality.

[57] studied the dialogue between users and developers of free apps in the Google Play Store. Evidences found, showed that it can be worthwhile for app owners to respond to reviews, as responding may lead to an increase in the given rating and that studying the dialogue between user and developer can provide valuable insights which may lead to improvements in the app store and the user support process. We believe the same rationale may be applied to comprehend the workflows and dialogues between developers and project owners, and how that may impact software products.

Development activities were extracted by [58] from non-instrumented applications and used machine learning algorithms to infer a set of basic development tasks. However, in this case, no process mining techniques were used to discover any pattern of application usage. The extraction of usage smells was the focus of [59], where a semi-automatic approach was adopted to analyze a large dataset of IDE interactions using cluster analysis. Again, process mining techniques were not used. Process mining was indeed used by [60] to gain knowledge on software under operation (not under development) by analyzing the hierarchical events produced by application calls(eg: execution of methods within classes) at runtime.

[61] collected events from the IDE to measure program comprehension and evaluated the correlation between developers' activities and the time they spent on them. Despite the fact that a process was being studied, no evidence of using process mining methods was provided.

A few authors have also followed the route we suggested earlier and resumed in [62]. As such, we are witnessing more evidences that it is indeed a valid approach, therefore, [63] used process mining to evaluate developers' coding behavior in software development processes. Process models were discovered and used to classify the developers as low-performing and high-

performing profiles. With a similar goal, in [64], a different miner algorithm was assessed to obtain complementary results and in [65], developers' profiling was achieved by mining event logs from a web-based cloud IDE.

Finally, [16] highlights the importance of having more fine-grained process metrics in prediction models and evaluated several machine learning algorithms in predicting software refactoring opportunities. This work focuses on deciding when, what and why to refactor, however, it does not address which refactor practice was indeed applied.

The studies mentioned above used a multitude of process mining techniques, statistics and machine learning methods. Different data source types have been used to extract the information needed to support them. However, to the best of our knowledge, none of these works combine process and product metrics with the aim of assessing potential correlations and/or impacts between the process and the product. Moreover, none uses only process metrics to discover work patterns or to predict development behaviors, particularly, refactoring practices.

4. Study Setup

We setup an environment where the same well-defined tasks on software quality assurance was performed independently by several teams.

Our research guaranteed that all teams had similar backgrounds and performed the same task upon the same software system. This approach was used to block additional confounding factors in our analysis. The task targeted a complex open-source Java system named `Jasml (Java Assembling Language)`[5].

To understand the work developed by each team in each task, we collected the corresponding IDE events for mining the underlying process. At the end of each task, we also collected the modified `Jasml` project code for each team and obtained the corresponding product metrics.

4.1. Subject Selection

Our subjects were the finalists (3rd year) of a B.Sc. degree on computer science at the ISCTE-IUL university, attending a compulsory software engineering course. They had similar backgrounds as they have been trained across the same set of courses along their academic path. Teams were assembled with up to 4 members each and were requested to complete a code smells detection assignment, aiming at identifying refactoring opportunities and then to apply them.

---

[5]http://jasml.sourceforge.net/

4.2. Data Collection

The participants were requested to perform the refactoring tasks in two different ways: **Automatically** and **Manually**.

The refactoring tasks had the following requirements:

- **Automatic Refactoring(AR).** This task was executed from March 1st to March 20th, using `JDeodorant`[6]. This tool suggests refactoring opportunities by detecting, among others, the following four well-known code smells: `Long Method, God Class, Feature Envy` and `Type Checking` [66]. Once participants have detected the occurrences of those code smells, they were required to apply `JDeodorant`'s fully automated refactoring features to fix the critical ones.

- **Manual Refactoring(MR).** This task was pursued from March 21st to 28th and differed from the previous one because `JDeodorant` automatic refactoring capabilities were banned. Instead, subjects could use Eclipse's native interactive refactoring features or perform the refactorings manually.

The `Eclipse` IDE has an internal event bus accessed by the interface `IEventBroker`[7] which is instantiated once the application starts. It contains a publishing service to put data in the bus, whilst the subscriber service reads what's in that bus. Using this feature we developed an `Eclipse` plugin[8] capable of listening to the actions developers were executing. Before the experiment, the plugin was installed on each subject's IDE, and later, all subjects received an unique *username/key* pair as credentials.

---

[6]https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[7]https://wiki.eclipse.org/Eclipse4/RCP/Event_Model
[8]https://github.com/jcaldeir/iscte-analytics-plugins-repository

A sample event instance collected with our plugin is presented in listing 1. The field tags are self explanatory.

Listing 1: Sample Eclipse Event Instance in `JSON` format

```
{
"team" : "Team−10",
"session" : "dkoep74−ajodje5−63j3k2",
"timestamp_begin" : "2019−05−03 16:53:52.144",
"timestamp_end" : "2019−05−03 16:54:04.468",
"fullname" : "John User",
"username" : "john",
"workspacename" : "Workspace1",
"projectname" : "/jasml_0.10",
"filename" : "/jasml_0.10/src/jasml.java",
"extension" : "java",
"categoryName": "Eclipse Editor",
"commandName": "File Editing",
"categoryID": "org.eclipse.ui.internal.EditorReference",
"commandID": "iscte.plugin.eclipse.commands.file.edit",
"platform_branch": "Eclipse Oxygen",
"platform_version": "4.7.3.M20180330−0640",
"java": "1.8.0_171−b11",
"continent": "Europe",
"country": "Portugal",
"city": "Lisbon",
....
"hash": "00b7c0ef94e02eb5138d33daf38054e3" //To detect event tampering
}
```

4.2.1. Data Storage

Collected data was stored locally on each subject's computer in a CSV file. Whenever Internet connection was available, the same data was stored in real-time in the cloud[9]. This storage replication mechanism allowed for offline and online collection[10]. The final dataset, combining the two different sources, was then loaded into a MySQL database table where the username and event timestamps that formed the table's unique key were used to detect and avoid duplicated data insertions. Figure 1 presents the complete schema for the data collection workflow. We use the BPMN standard process definition language for that purpose [67].

4.2.2. Data Preparation

When the software quality task ended, we collected from each team their projects' code together with the events files containing the actions performed

---

[9]https://azure.microsoft.com/en-us/services/event-hubs/

[10]The plugin currently supports the collection of events locally in CSV and JSON files; stream events to Azure Event Hub and Kafka remotely; and uses an integration with Trello to extract project activities which can be triggered as manual events by the developers. Kafka and Trello integrations were not used in this experiment.
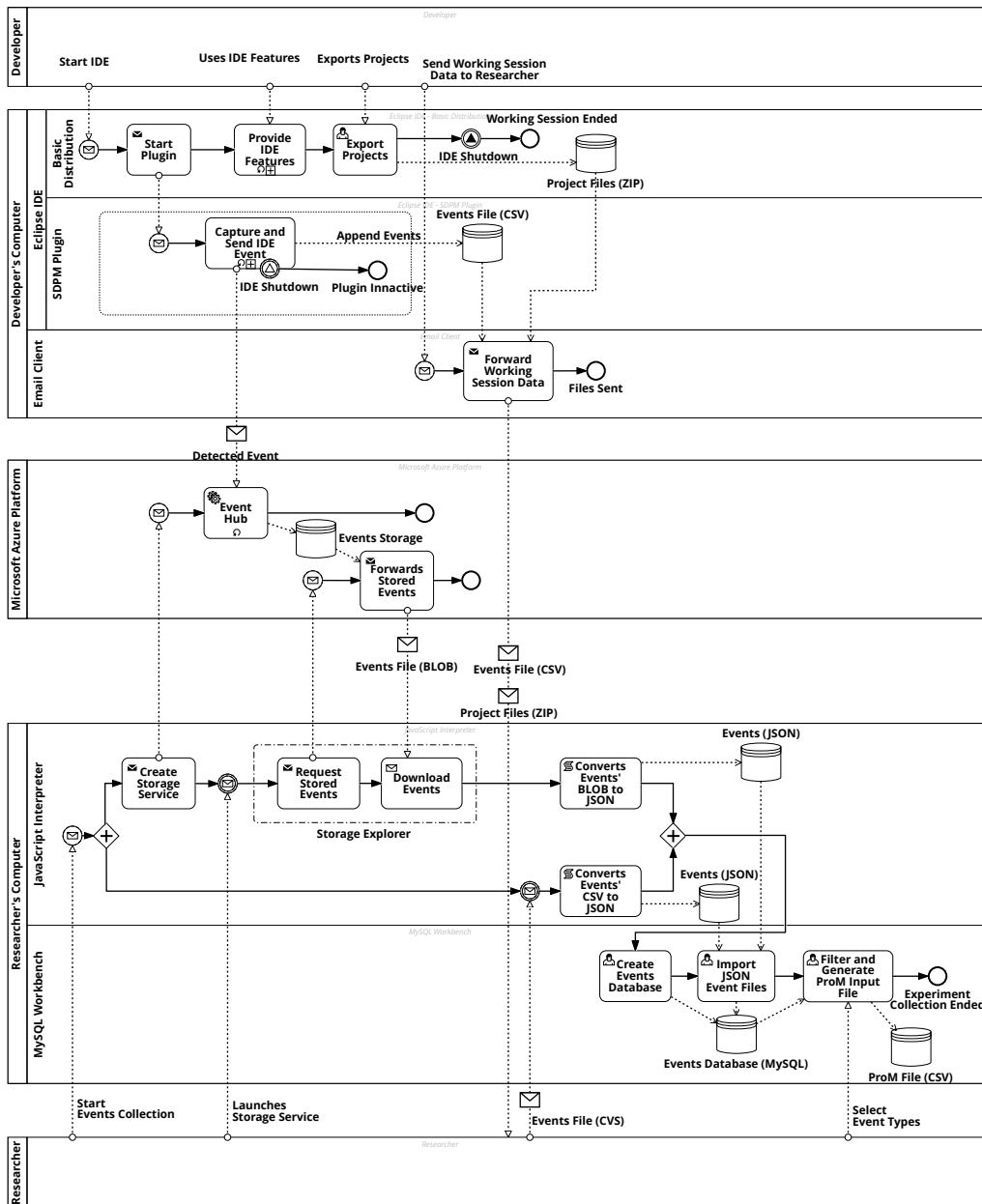
14

Figure 1: Experiment Data Collection Workflow

during the aforementioned activities. As such, each team produced and delivered two new Jasml projects, one for the automatic and another for the manual refactoring. The events files would map events for the two different tasks, as they were done in different time frames.

All events stored in the database were imported into the ProM process

mining tool[11] and converted to the IEEE eXtensible Event Stream (XES) standard format [68]. The following event properties were mapped when converting to XES format:

- *team* and *session* were used as **CaseID** since we were interested to look into process instances of teams and their multiple development sessions, not of individual programmers.

- Properties *filename*, *categoryName* and *commandName* forming a hierarchical structure were used as the **Activity** in the process.

- The *timestamp_begin* and *timestamp_end* were both used as activity **Timestamps**.

- Other properties were not used in the process discovery phase, however, they were later used for metrics aggregation and context analysis.

4.3. Data Analysis

4.3.1. Context

All teams started with the same version of **Jasml 0.10**, therefore, we had two relevant moments to get measures from:

1. The initial moment *(t0)*, when we extracted the metrics for the initial product version. However, we didn't know how it was built, therefore, we were missing[12] the process metrics.

2. The end of the task *(t1)*, when we extracted again the product metrics for the changed **Jasml 0.10** project of each team as they stand after the refactoring sessions. In addition, we had also IDE usage events which provide evidences on how the product was changed.

Following data extraction, we computed, for each product metric defined in Table A.7, their relative variance as shown by Equation 1. The relative variance variables were the ones we used in all **RQs**.

$$\Delta product\ metrics_{(t1\text{-}t0)} = \frac{product\ metrics_{(t1)} - product\ metrics_{(t0)}}{product\ metrics_{(t0)}} * 100 \quad (1)$$

---

[11]Version 6.8, available at http://www.promtools.org

[12]In reality we may consider all of them to be zero

16

The relative variance was used in order to generalize our approach, thus, making it applicable in scenarios where different teams work on distinct software projects.

Process metrics described in Table A.8 were derived from the events dataset captured between moments *(t0)* and *(t1)*, either by summing the events or using the method described in 4.3.3. These metrics may be seen as a representation of the effort done by each team during the refactoring practices.

The complete workflow followed in data pre-processing, aggregation and analysis is presented in Figure 2.

4.3.2. Product and Process Metrics

To extract software metrics we used the plugin built by Sauer[13]. Although having more than a decade of age, it is still one of the more proven and popular options regarding open source metrics plugins for Eclipse.

The plugin itself offers a simple interface and reporting capabilities with which users can define optimal ranges and issue warnings for certain metrics, as well as being able to export calculated metrics to XML files. The set of metrics obtained by this plugin are presented in Table A.7 on Appendix A.1.

4.3.3. Process Discovery

Several well known algorithms exist to discover process models, such as, the α-algorithm, the heuristics, genetic and the fuzzy miner amongst others [69, 70]. Our need to discover and visualize the processes in multiple ways lead us to choose the `ProM's StateChart Workbench` plugin [60]. This plugin, besides supporting process model discovery using multiple hierarchies and classifiers, also allows to visualize the model as a Sequence Diagram and use notations such as Petri Nets and Process Trees. This plugin is particularly suitable for mining software logs, where an event structure is supposed to exist, but it also supports the mining of other so-called generic logs.

Events collected from software in operation (e.g. Java programs) reveals the presence of a hierarchical structure, where methods reside within classes, and classes within packages [71]. The same applies to IDE usage actions where identified menu options and executed commands belong to a specific category of command options built-in the `Eclipse` framework. Supported by this evidence, we used the Software log Hierarchical discovery method with a Structured Names heuristic to discover the processes based on the fact that the events were using a *filename|category|command* structure (e.g. `/jasml0.10/src/jasml.java|Eclipse Editor|File Open`).

---

[13]http://metrics.sourceforge.net

17
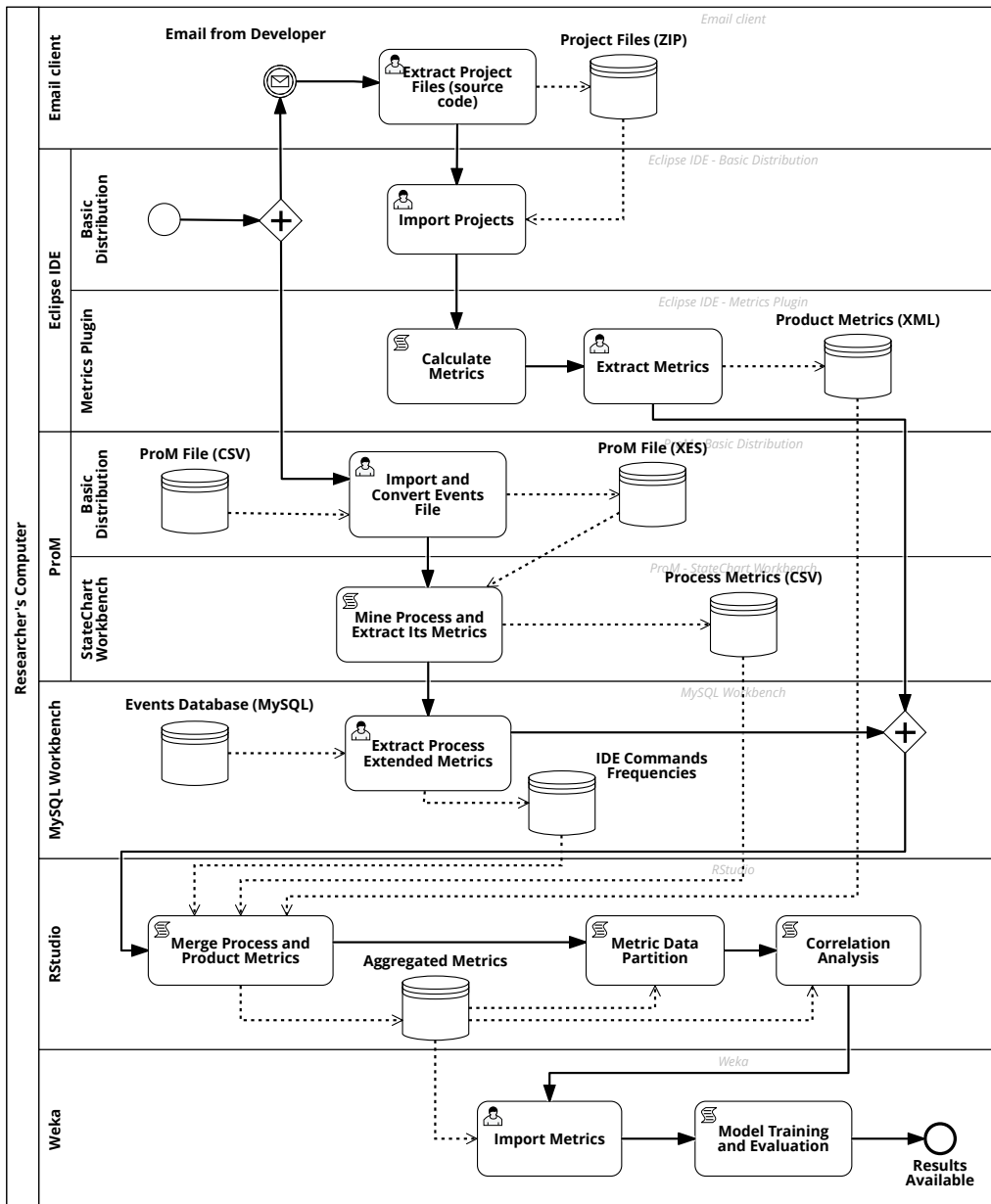
Figure 2: Study Computation and Analysis Process

Several perspectives can be used to discover and analyze a business process. The commonly used are: `Control-Flow`, `Organizational`, `Social` and `Performance`. We have focused on the `Control-Flow` perspective in this paper. It defines an approach that consists in analyzing how each task/activity follows each other in an event log, and infer a possible model for the behavior captured in the observed process.

Process metrics, shown in Tables A.8 and A.9 on Appendix A.2, were obtained using the discovery method described in 4.3.3, and by running queries into the events database as presented in Figure 2.

4.3.4. Data Partitioning

We used the **k-means** clustering algorithm to compute new variables based on the partition of the teams across different levels (clusters) of Process Cyclomatic Complexity *(PCC)* and McCabe Cyclomatic Complexity variance *(ΔVG)* . The decision of how many clusters to use **(k)** was supported by a detailed analysis of the **Elbow** and **Silhouette** methods:

- **Elbow Method.** It is frequently used to optimize the number of clusters in a data set. This heuristic, consists of rendering the explained variation as a function of the number of clusters, and picking the elbow of the curve as the optimal number of clusters to use. In cluster analysis, the elbow method runs **k-means** clustering on the dataset for a range of values for **k** (say from 2-10), and then, for each value of **k** computes an average score for all clusters. The distortion score is computed as the sum of square distances from each point to its assigned center [72].

- **Silhouette Method.** It is a commonly used approach of interpretation and validation of consistency within clusters of data. Provides a concise graphical representation of how well each object has been classified within the corresponding cluster. The Silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette can be calculated with any distance metric, such as the Euclidean distance or the Manhattan distance, and ranges from -1 to +1. A high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. The clustering configuration is appropriate if most objects have a high value. If many objects have a low or negative value, then the clustering configuration may have too many or too few clusters and, as such, requires further research before a decision on the optimal number of **k** clusters is made [73].

4.3.5. Model Selection with Hyperparameter Optimization

To build, tune model parameters as recommended [74, 75], train, evaluate and select the best-fit classification models presented in Tables 5 and 6, we

used **Weka** and the **Auto-Weka** plugin. **Weka** (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software written in Java. It's workbench contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to this functionality [76]. **Auto-Weka** is a plugin that installs as a **Weka** package and uses Bayesian optimization to automatically instantiate a highly optimized parametric machine learning framework with minimum user intervention [77].

4.3.6. Model Evaluation

Several evaluation metrics can be used to assess model quality in terms of false positives/negatives (FP/FN), and true classifications (TP/TN). However, commonly used measures, such as **Accuracy, Precision, Recall and F-Measure**, do not perform very well in case of an imbalanced dataset or they require the use of a minimum probability threshold to provide a definitive answer for predictions. For these reasons, we used the **ROC**[14], which is a threshold invariant measurement. Nevertheless, for general convenience, we kept present in Tables 5 and 6 all the evaluation metrics.

ROC gives us a 2-D curve, which passes through (0, 0) and (1, 1). The best possible model would have the curve close to y = 1, with and area under the curve (**AUC**) close to 1.0. **AUC** always yields an area of 0.5 under random-guessing. This enables comparing a given model against random prediction, without worrying about arbitrary thresholds, or the proportion of subjects on each class to predict [28].

4.4. Research Questions

The research questions for this work are:

- **RQ1:** How different refactoring methods perform when the goal is to reduce complexity, future testing and maintainability efforts?.
  Methods Used. Process Mining Model Discovery, Descriptive statistics and Cluster Analysis.

- **RQ2:** Is there any association between software complexity and the underlying development activities in refactoring practices?
  Methods Used. Process Mining Model Discovery, Correlation Analysis using the Spearman's rank correlation.

---

[14]Receiver operating characteristic (**ROC**) is a curve that plots the true positive rates against the false positive rates for all possible thresholds between 0 and 1.

- **RQ3:** Using only process metrics, are we able to predict with high accuracy different refactoring methods?
  Methods Used. Supervised and Unsupervised Learning Algorithms with Hyperparameter Optimization.

- **RQ4:** Using only process metrics, are we able to model accurately the expected level of complexity variance after a refactoring task?
  Methods Used. Supervised and Unsupervised Learning Algorithms with Hyperparameter Optimization.

5. Study Results

In this section, we present the experiment results with respect to our research questions.

5.1. *RQ1. How different refactoring methods perform when the goal is to reduce complexity, future testing and maintainability efforts?*

In this **RQ**, we used as product metrics, the ones identified in section 4.3.2. Since IDE usage is a sequence of actions (it can be seen as a process, or at least, as a process fragment), we used as process metrics the ones identified in 4.3.3. Notice that both, product and process metrics, have been computed to obtain the Δ between **t1 and t0**.

Table 2: Teams' Statistics

Task Mode	Teams	Dev.	Ses.	Evts.	ΔVG	PCC
Automatic Refactoring	32	65	150	10443	7.81%	166.5
Manual Refactoring	39	52	170	22676	2.69%	300.3
Total	71	117	320	33119		

Dev - Developers, **Ses** - Sessions, **Evts** - Events,
ΔVG - McCabe Cyclomatic Complexity Reduction %(mean),
PCC - Process Cyclomatic Complexity(mean)

Table 3: Teams' Refactoring Results

Metric Name	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Automatic Refactoring						
ΔVG	2.68%	5.87%	6.95%	7.81%	8.84%	16.77%
PCC	24.0	77.0	168.5	166.5	218.2	407.0
Manual Refactoring						
ΔVG	0.32%	0.62%	0.94%	2.69%	3.92%	13.98%
PCC	53.0	152.0	275.0	300.3	407.0	738.0
Data Partition						
VG_LEVEL	**LOW** = [0%, 4%]; **MEDIUM** = [4.1%, 9%]; **HIGH** = [>9%]					
PCC_LEVEL	**LOW** = [0, 285]; **HIGH** = [>285]					

ΔVG - McCabe Cyclomatic Complexity Reduction %,
PCC - Process Cyclomatic Complexity

We had 32 teams performing automatic refactoring using the *JDeodorant* plugin, and 39 doing manual refactoring supported only by the Eclipse native features and/or driven by the developers experience and skills. Table 2 shows the total number of developers and their activities, here referred as development sessions. In Table 3 we show measures of central tendency and measures of variability regarding the distribution of ΔVG and PCC, together with how both were partitioned.

Figure 3 provides evidence for selecting the optimal number of clusters to partition the data according to **LOW** or **HIGH** levels of process cyclomatic complexity used in Figure 4. The same clustering method was used to partition the different levels of software cyclomatic complexity as **LOW**, **MEDIUM** or **HIGH**.
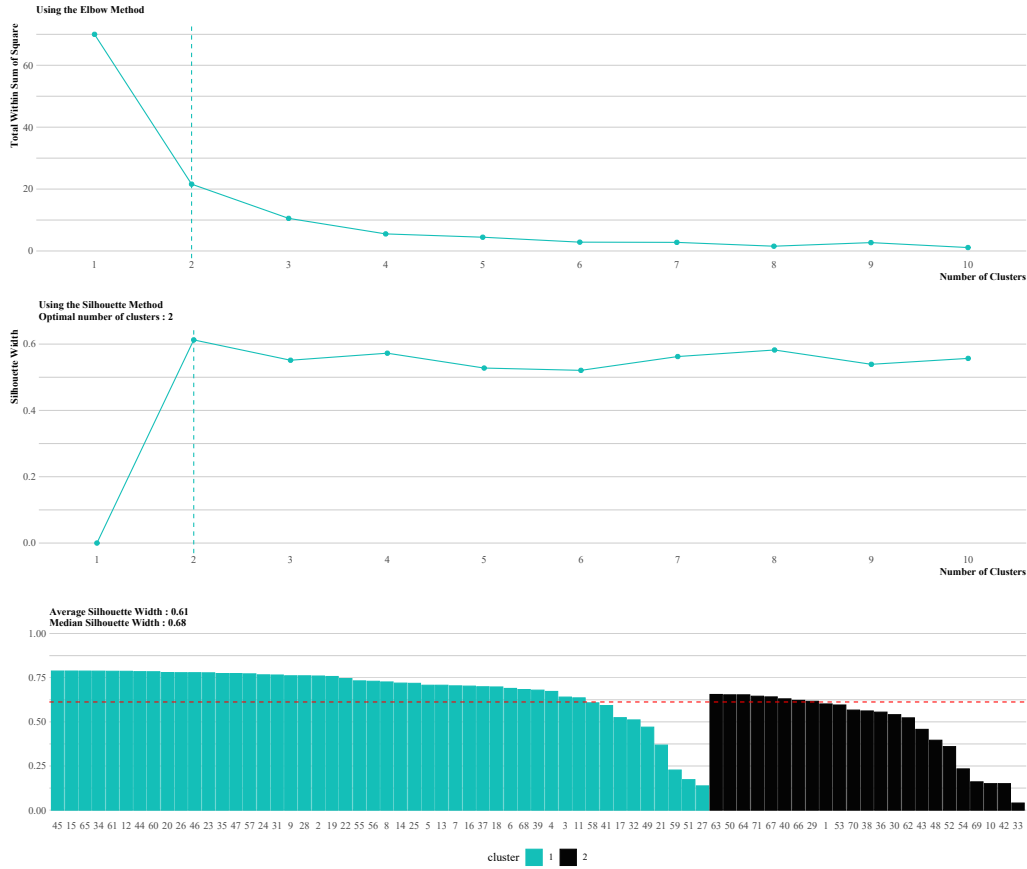
Figure 3: Detecting optimal partitions of PCC

Observation 1: Automatic Refactoring achieves higher levels of McCabe Cyclomatic complexity reduction. Consider relevant in Table 2, how the mean of code cyclomatic complexity reduction *(ΔVG)* for automatic refactoring is almost three times the reduction when doing manual refactoring. It is also relevant to mention, by looking at Figure 4, that only four teams had high complexity levels in their work sessions when doing refactoring using *JDeodorant*. Furthermore, from those, one team had the major software complexity reduction(16.77%), whilst other had near the lowest value of reduction(2.68%) within the automatic refactoring practice. The observation of such different results raised the doubt about the comprehension, focus and behaviour of those two teams in the given task. This demanded further investigation on their efficiency, for which, we provide some evidences later using Figures 6 and 7.

Observation 2: Manual refactoring practices have higher process
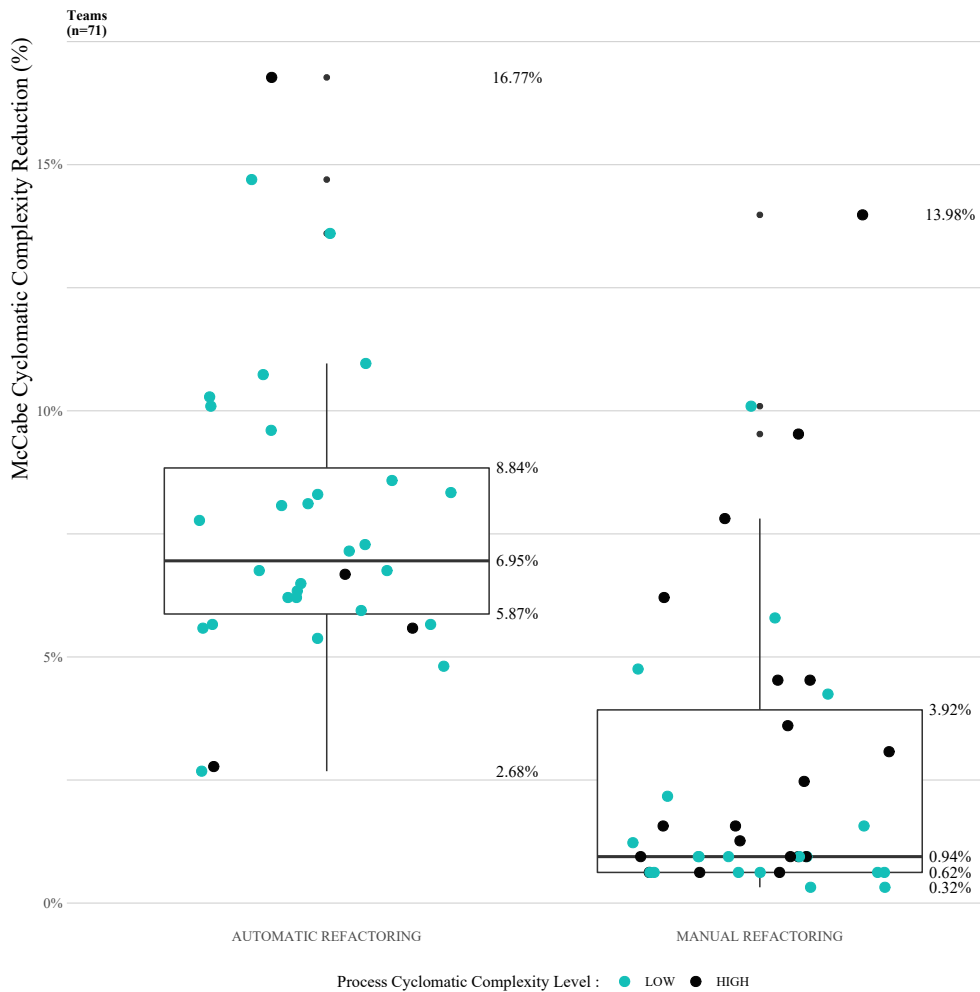
Figure 4: Refactoring Practices Comparison

cyclomatic complexity. We observe that teams doing manual refactoring almost double the mean of process cyclomatic complexity *(PCC)*, when compared with the ones using the automatic features of *JDeodorant*. Being deprived of the code smell detection plugin, these teams had to do more manual work to potentially achieve the same results as the ones doing automatic refactoring. This suggest that the refactoring plugin was working as expected, thus reducing software complexity with less effort simply because several code snippets may have been introduced automatically.

On the contrary, teams doing the task manually needed to do more code, and therefore, more actions within the IDE to detect and correct the code smells. As shown earlier in section 1, manual refactoring tasks can introduce

24
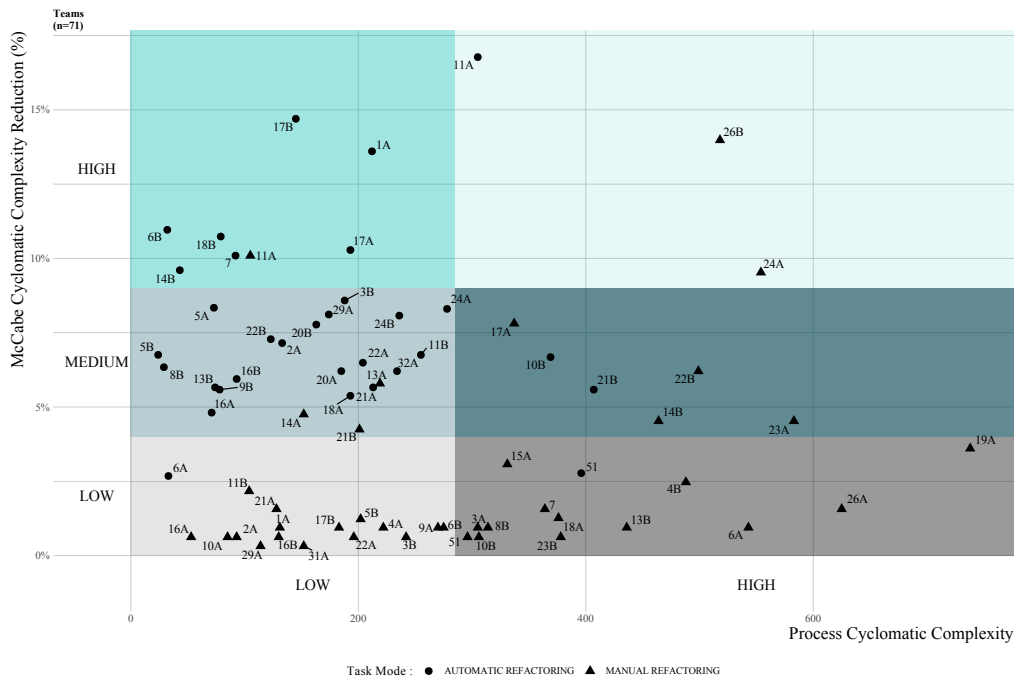
Figure 5: Plotting teams according to levels of software and process cyclomatic complexity

non expected defects in the code and is seen as a practice to avoid.

Figure 4 plot the percentage of McCabe Cyclomatic Complexity per method reduction obtained after both refactoring sessions. The different colors plot the different levels of process cyclomatic complexity as discovered from mining each team events log.
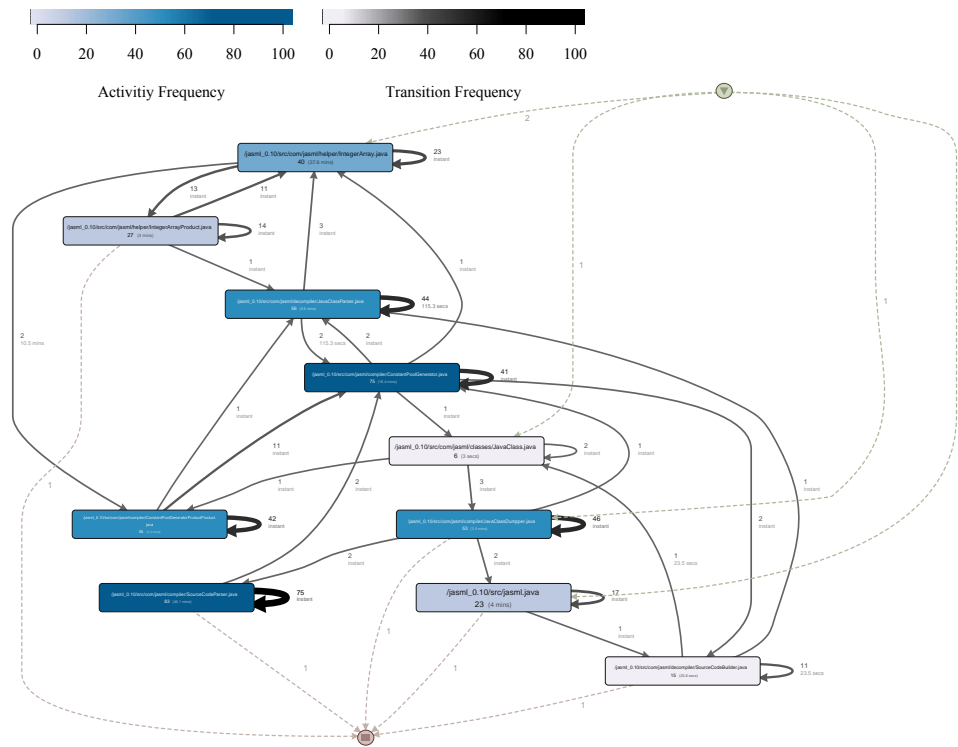
Figure 6: Team 11A : High PCC and High VG reduction (20% activities/files, 80% paths)
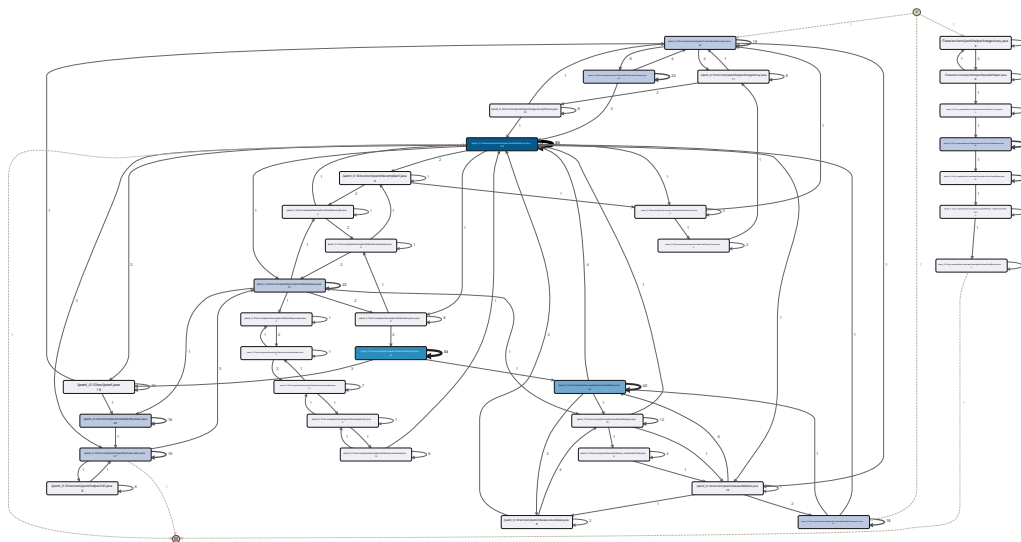
Figure 7: Team 51 : High PCC but Low VG reduction (20% activities/files, 80% paths)

Observation 3: Even using JDeodorant, similar work efforts does not mean the same level of gains in software complexity reduction. If it is apparent that, when using *JDeodorant*, the processes tend

to have lower levels of complexity and obtained globally more gains in product complexity reductions, the same cannot be said for teams doing manual refactoring. These teams have a more heterogeneous process behavior since they were free to apply refactoring functionalities without any guidelines in detection and correction from a dedicated plugin. Figure 5 identifies all teams and distributes them according to their levels of software and process complexity.

From Figure 4, we can also observe that the team (11A) with the highest reduction in code complexity ($\approx 16.77\%$), had also a high level of process complexity even if they were using the JDeodorant plugin. We can also identify a team(51) doing automatic refactoring with high levels of process complexity but having instead, very low gains in code cyclomatic complexity reduction ($\approx 2.68\%$). As such we investigated the activities of both teams in order to identify potential reasons for this substantial variation.

Figures 6 and 7, represent the process flow views of both individual teams regarding the files browsed and/or changed during the refactoring practice[15]. Based on the same values for the activities and paths, we can clearly identify that the team with high gains in VG reduction worked in less files (number of nodes) and was focused evenly on all of them (dark blue nodes means more actions on those files).

On the contrary, the team with low gains in VG, visited more files but worked frequently on only 3 of them. This fuzzy behavior suggests lack of focus and/or knowledge about the task to accomplish, and present a good way to measure efficiency on development teams or individual developers. That can be confirmed by comparing both teams statistics in Figure 8, where we present product metrics, process metrics and extended process metrics scaled to represent their position to the mean value of each action for both teams.

We highlight in the extended process metrics the fact that the team with bigger VG reduction was the one with less frequencies in commands such as : Undo, Cut, File Open, File Close plus other navigational and less productive actions. This team had also bigger frequencies in commands to detect and fix code smells, such as: God Class, Duplicated Code and Type Checking. However, the gains in the VG reduction were achieved at the cost of increasing 28% the number of classes(NOC) and the lack of cohesion of methods($LCOM$) by $\approx 72\%$. On the process side, despite the fact that this team had more work sessions(7), they touched less files, meaning their activities

---

[15]We acknowledge that the labels in these two diagrams, produced by the Disco tool, are illegible in a printing version. However, since the figures are in vectorial format, they can be *zoomed in* easily if this paper is read in its electronic version (pdf), the most probable access medium.
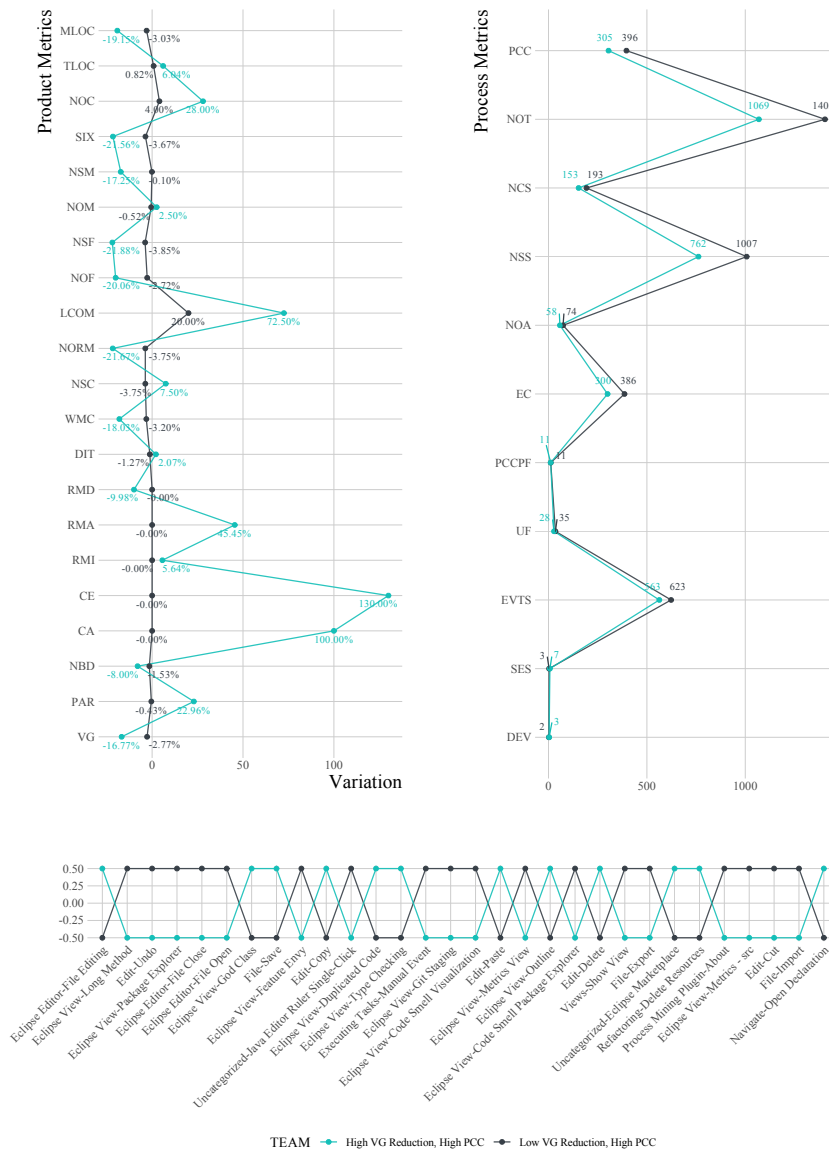
Figure 8: Teams(11A vs. 51) with distinct VG variance positioning but similar PCC levels

were less complex, and that is confirmed by the *UF*, *NOA* and *PCC* metrics.

5.2. **RQ2. Is there any association between software complexity and the underlying development activities in refactoring practices?**

With the evidences shown in **RQ1** for the two distinct refactoring methods, one may question if the product complexity reduction gains are monotonically correlated with the development activities which originated them.

We used the Spearman correlation coefficient to measure the strength of correlation between metrics of these two dimensions, product and process complexities. This coefficient ranges from -1 to 1, where -1 and 1 correspond to perfect negative and positive relationships respectively, and 0 means that the variables are independent of each other.

To validate our results, we performed a significance test to decide whether based upon this sample there is any or no evidence to suggest that linear correlation is present in the population. As such, we tested the null hypothesis, H_0, and the alternative hypothesis, H_1, to gather indication of which of these opposing hypotheses was most likely to be true.

Let p_s be the Spearmans' population correlation coefficient both for automatic and manual refactoring, then we can thus express this test as:

H_0: $p_s = 0$: No monotonic correlation is present in the practice.

H_1: $p_s \neq 0$: A monotonic correlation is present in the practice.

Automatic Refactoring. After computing the Spearman correlation coefficient on the subset of teams doing automatic refactoring, and despite the fact that some correlations were slightly negative as we expected, we got no significant statistics on the correlation of ΔVG and PCC or any other pair of metrics, as shown by Spearmans' *rho* and *p-value* in Table 4.

Observation 4: No significant correlation was found between product and process metrics on automatic refactoring practices. Hence, we can say that we cannot reject the null hypothesis, H_0, meaning that a monotonic correlation cannot said to be found between code cyclomatic complexity and process cyclomatic complexity or any other process metric.

Table 4: Spearmans' Correlation - Refactoring Tasks

Factors	Automatic Refactoring ΔVG		Manual Refactoring ΔVG	
	Spearmans' rho	*p-value*	*Spearmans' rho*	*p-value*
PCC	-0.02	0.9707	**0.43**	**0.0432***
UF	0.01	0.5218	0.32	0.3427
SES	0.15	0.7489	0.24	0.2814
DEV	-0.05	0.7342	0.03	0.8193
NPER	-0.19	0.4976	**0.32**	**0.0197***
NISP	-0.10	0.6875	**0.35**	**0.0120***
PCCPF	-0.01	0.7787	**0.45**	**0.0059***
NCAT	-0.11	0.6309	**0.39**	**0.0096***
NCOM	-0.05	0.6240	0.42	0.0712

*Statistically significant if p-value < 0.05

Manual Refactoring. When analyzing the dataset with manual refactoring activities, we found that product complexity reduction was moderately correlated with the process cyclomatic complexity and several other metrics process related. Table 4 presents Spearmans' *rho* and *p-value*, highlighting the significant correlations[16].

Observation 5: A moderate correlation was found between product metrics and process metrics on manual refactoring tasks. It is relevant to highlight the presence of a moderate positive correlation between the product cyclomatic complexity reduction *(ΔVG)* and the overall process cyclomatic complexity *(PCC)* and per unique file touched *(PCCPF)*. This means that the more actions the teams have done within the IDE the bigger the gains obtained in complexity reduction.

---

[16]Other product and process metrics were omitted due to the absence of significant correlations
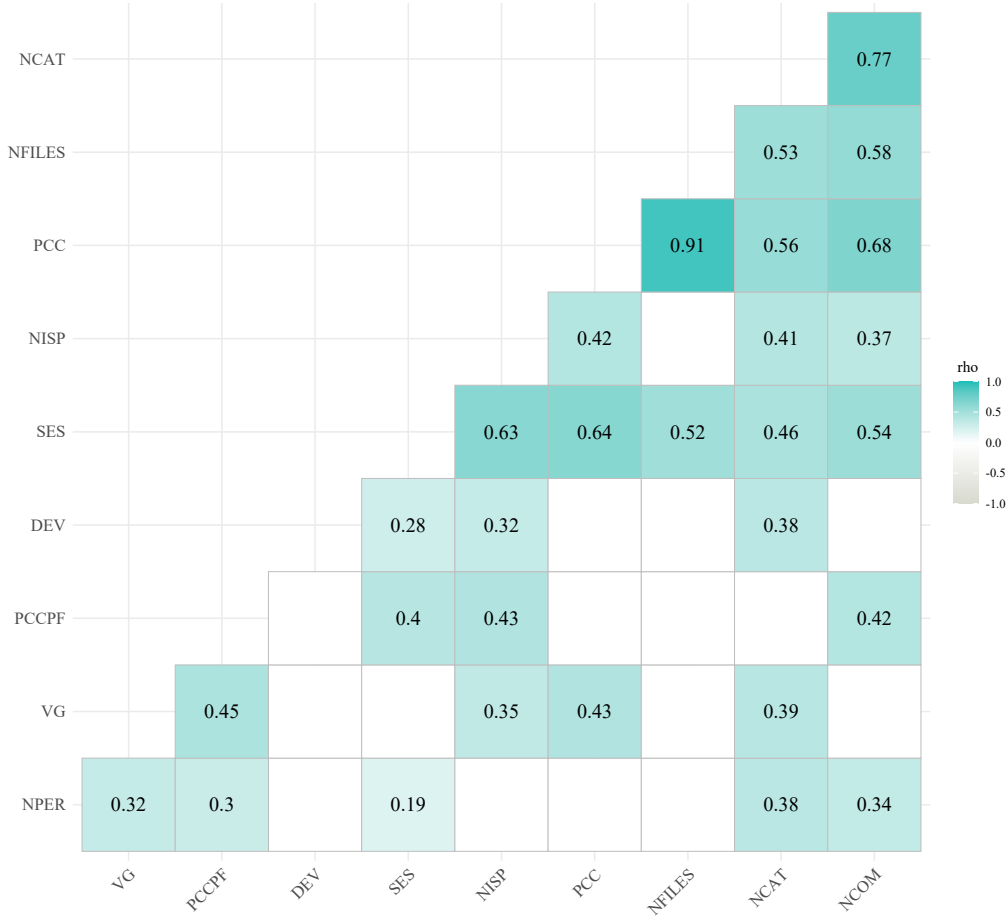
Figure 9: Manual Refactoring correlation results

Observation 6: Weak to moderate correlations were found between product complexity reduction and IDE command categories. Weak to moderate correlations emerge when we pair the product complexity reduction with the number of the IDE command categories *(NCAT)*, IDE perspectives activated *(NPER)* and the number of distinct physical locations from where the task was performed *(NISP)*. Based on the significance tests, we can reject H_0, and accept H_1, meaning that a monotonic correlation exists between code cyclomatic complexity and process cyclomatic complexity as well as with the other highlighted metrics.

Observation 7: No significant correlations were found between any process metrics and product metrics, except for ΔVG. All product and process metrics collected are shown in Tables A.7 and A.8.

Figure 9 plot only the significant correlations[17] among all those we studied. As expected, process metrics show strong correlations between themselves, however, we find this result obvious and not relevant withing the context of this study.

5.3. *RQ3. Using only process metrics, are we able to predict with high accuracy different refactoring methods?.*

Process metrics have been confirmed as suitable predictors for many software development prediction models. They were found not only suitable, they performed significantly better than code metrics across all learning techniques in several studies [78, 28].

Our goal was to use the process metrics described in Table A.8, to predict if a refactoring task executed by a group of teams had been done automatically, using the *JDeodorant* features, or manually, using only the Eclipse native functionalities or driven by developers skills. Each subject in our dataset has the class to predict labelled as **AR** and **MR** for automatic and manual refactoring, respectively. In this case, we did not use metrics from Table A.9 because that would introduce bias in our models since the process extended metrics can easily be used to understand if developers used or not IDE built in features or their own skills during a refactoring practice.

Table 5 present the results for the 5 best models we got out of the ≈30,000 we evaluated on our research. In this context, the machine learning models used were built by assembling and testing supervised or unsupervised algorithms adjusted with feature selection and hyperparameter optimization. From the models built, the ones with higher ROC were chosen. A brief explanation of each algorithm can be found in Appendix A.3, as well as the code obtained from training Model 1.

Observation 8: Random Forest confirms its accuracy. Random Forest models were found to be the ones with higher accuracy in predicting refactoring opportunities in previous studies [16]. We observe the same behaviour. Random Forest shows twice in the top 5 of our best models, with a **ROC** value of 0.983 and 0.939 for Model 1 and 2, respectively. In both cases, the models were computed by a meta learner which builds an ensemble of randomizable base classifiers, the Random Committee.

---

[17]Blank squares means non significant values

Table 5: Detailed Model Evaluation

Model	TP	FP	Pre.	Rec.	F-M.	MCC	ROC	PRC
Model 1, RandomCommittee/RandomForest, Accuracy = 92.95%								
AR	0.906	0.051	0.935	0.906	0.921	0.858	0.983	0.980
MR	0.949	0.094	0.925	0.949	0.937	0.858	0.983	0.987
W. Avg.	0.930	0.075	0.930	0.930	0.929	0.858	0.983	0.984
Model 2, RandomCommittee/RandomForest, Accuracy = 90.14%								
AR	0.875	0.077	0.903	0.875	0.889	0.801	0.939	0.923
MR	0.923	0.125	0.900	0.923	0.911	0.801	0.939	0.948
W. Avg.	0.901	0.103	0.901	0.901	0.901	0.801	0.939	0.937
Model 3, Logistic Model Trees, Accuracy = 90.14%								
AR	0.906	0.103	0.879	0.906	0.892	0.802	0.945	0.938
MR	0.897	0.094	0.921	0.897	0.909	0.802	0.945	0.951
W. Avg.	0.901	0.098	0.902	0.901	0.902	0.802	0.945	0.945
Model 4, RandomSubSpace/REPTree, Accuracy = 88.73%								
AR	0.844	0.077	0.900	0.844	0.871	0.772	0.929	0.907
MR	0.923	0.156	0.878	0.923	0.900	0.772	0.929	0.935
W. Avg.	0.887	0.120	0.888	0.887	0.887	0.772	0.929	0.922
Model 5, Logistic Regression, Accuracy = 83.09%								
AR	0.750	0.103	0.857	0.750	0.800	0.659	0.939	0.940
MR	0.897	0.250	0.814	0.897	0.854	0.659	0.939	0.950
W. Avg.	0.831	0.184	0.833	0.831	0.829	0.659	0.939	0.945

TP-True Positive, **FP**-False Positive, **Pre**-Precision, **Rec**-Recall,
F-M-F-Measure, **MCC**-Matthews Correlation Coefficient,
ROC-Receiver Operating Characteristic, **PRC**-Precision-Recall Curve,
AR-Automatic Refactoring, **MR**-Manual Refactoring,
W. Avg-Weighted Average

Our dataset is not imbalanced, thus, we have almost the same number of subjects for each class, meaning we may use also the **Accuracy** metric to complement our analysis. Model 1 and 2 had respectively, an accuracy of 92.5% and 90.14%.
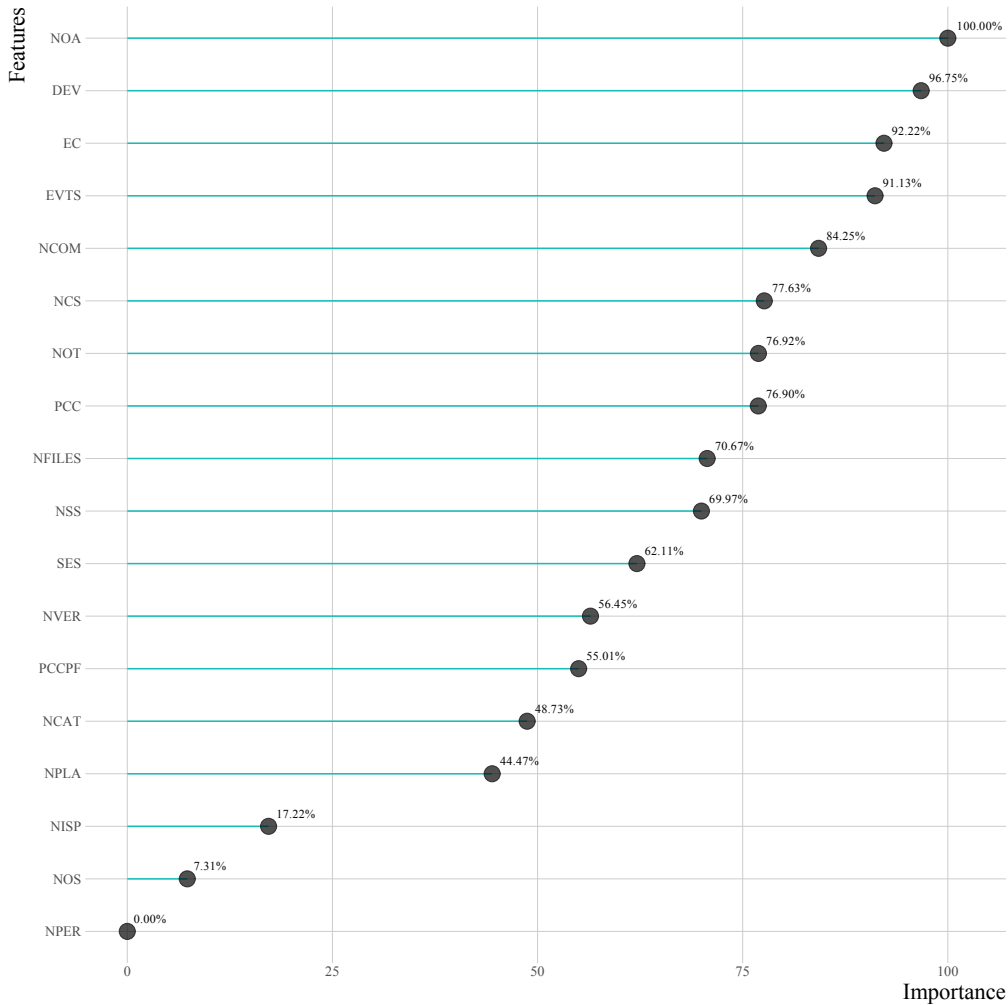
Figure 10: Feature importance for models on Table 5

During models computation phase, we also assessed which of the features were more or less important to predict the refactoring practices: automatic(**AR**) or manual(**MR**). Figure 10 shows their average importance.

Observation 9: Number of Activities, Developers and Commands are the most relevant model features. These features show among the ones with highest importance in the models we computed. We recall that the number of activities (NOA) is a composite metric obtained by the process mining extraction plugin using a hierarchical structure composed of the filename, command category and commands issued during the coding phase. Having a mid level importance we find the process cyclomatic complexity and the number of development sessions.

Observation 10: Distinct IDE Perspectives and Operating Sys-

34

tems have almost irrelevant importance. In our models, the different types operating system used by the developers, the different number of IDE perspectives and number of development locations (*NISP*) are irrelevant predictors in modeling the type of refactoring performed. We argue that, particularly the number of different locations from where the developers performed their work require additional research in order to get any generalized conclusions about this insight.

5.4. *RQ4: Using only process metrics, are we able to model accurately the expected level of complexity variance after a refactoring task?*

To answer this **RQ**, we used not only the metrics from Table A.8, but also the ones from Table A.9. During our analysis, it was clear that process extended metrics, representing the commands issued by each developer/team, added significant predictive power to the models computed. Therefore, to predict the expected software cyclomatic complexity we needed to include individual commands frequencies in addition to the process metrics used in previous **RQ**. By doing this we were able to achieve models with higher accuracy and good ROC values. However, in general, these models have lower accuracy than the ones in **RQ3**.

Table 6 shows the top five models computed to predict the complexity level gains obtained after a refactoring session, either using a dedicated plugin or simply by using Eclipse features.

Observation 11: Locally Weighted Learning combined with a Decision Table outperforms Random Forrest. Contrary to the previous RQ, in this case the best model is not based on a Random Forrest algorithm. However, the latter show as the second best model in terms of accuracy. The Locally Weighted Learning method uses an instance-based algorithm to assign instance weights which are then used by a specified weighted instances handler. It uses a stack of methods, initially a cluster like mechanism such as the LinearNNSearch and then a Decision Table to classify the outcome. This shows up at no surprise since Decision Tables use the simplest hypothesis spaces possible and usually outperform state-of-the-art classification algorithms.

Table 6: Detailed Model Evaluation

Model	TP	FP	Pre.	Rec.	F-M.	MCC	ROC	PRC
Model 1, LWL/LinearNNSearch/DecisionTable, Accuracy = 94.36%								
LOW	0.968	0.000	1.000	0.968	0.984	0.972	0.991	0.992
MEDIUM	1.000	0.095	0.879	1.000	0.935	0.892	0.994	0.992
HIGH	0.727	0.000	1.000	0.727	0.842	0.832	0.992	0.967
Weighted Avg.	0.944	0.039	0.950	0.944	0.942	0.917	0.993	0.988
Model 2, Bagging/RandomForest, Accuracy = 83.09%								
LOW	0.839	0.075	0.897	0.839	0.867	0.771	0.938	0.94
MEDIUM	0.828	0.095	0.857	0.828	0.842	0.737	0.971	0.945
HIGH	0.818	0.083	0.643	0.818	0.720	0.668	0.971	0.827
Weighted Avg.	0.831	0.085	0.841	0.831	0.834	0.741	0.957	0.926
Model 3, KStar, Accuracy = 78.87%								
LOW	0.935	0.225	0.763	0.935	0.841	0.707	0.948	0.951
MEDIUM	0.862	0.143	0.806	0.862	0.833	0.713	0.945	0.915
HIGH	0.182	0.000	1.000	0.182	0.308	0.398	0.982	0.904
Weighted Avg.	0.789	0.157	0.818	0.789	0.755	0.661	0.952	0.929
Model 4, RandomCommittee/REPTree, Accuracy = 74.64%								
LOW	0.903	0.300	0.700	0.903	0.789	0.603	0.895	0.873
MEDIUM	0.759	0.143	0.786	0.759	0.772	0.619	0.886	0.847
HIGH	0.273	0.000	1.000	0.273	0.429	0.491	0.932	0.738
Weighted Avg.	0.746	0.189	0.781	0.746	0.726	0.592	0.897	0.842
Model 5, LWL/LinearNNSearch/DecisionTable, Accuracy = 71.83%								
LOW	0.871	0.300	0.692	0.871	0.771	0.569	0.843	0.803
MEDIUM	0.759	0.190	0.733	0.759	0.746	0.565	0.800	0.729
HIGH	0.182	0.000	1.000	0.182	0.308	0.398	0.823	0.541
Weighted Avg.	0.718	0.209	0.757	0.718	0.689	0.541	0.822	0.732

TP-True Positive, **FP**-False Positive, **Pre**-Precision, **Rec**-Recall,
F-M-F-Measure, **MCC**-Matthews Correlation Coefficient,
ROC-Receiver Operating Characteristic, **PRC**-Precision-Recall Curve,
LOW-Low level of Cyclomatic Complexity,
MEDIUM-Medium level of Cyclomatic Complexity,
HIGH-High level of Cyclomatic Complexity,
W. Avg-Weighted Average

Observation 12: Teams with LOW level of software complexity gains are frequently spotted with higher F-Measure and ROC values. Our models perform better in detecting subjects achieving low levels of complexity reduction. These are the most critical cases, as such, a software development project manager can quickly detect the teams or individuals responsible for those outcomes and implement actions to bring the project under acceptable quality parameters.

Observation 13: Process extended metrics have in general higher importance than process standard metrics. From Figure 11 we can understand that 18 out of 30 metrics are related with the commands issued by the developers. In general, these metrics have also higher importance in the models. It is not surprising to find methods and class extraction commands in the top of the list, with ≈86% and ≈56% importance, respectively. It was however unexpected to find project export actions being so relevant (≈70%).
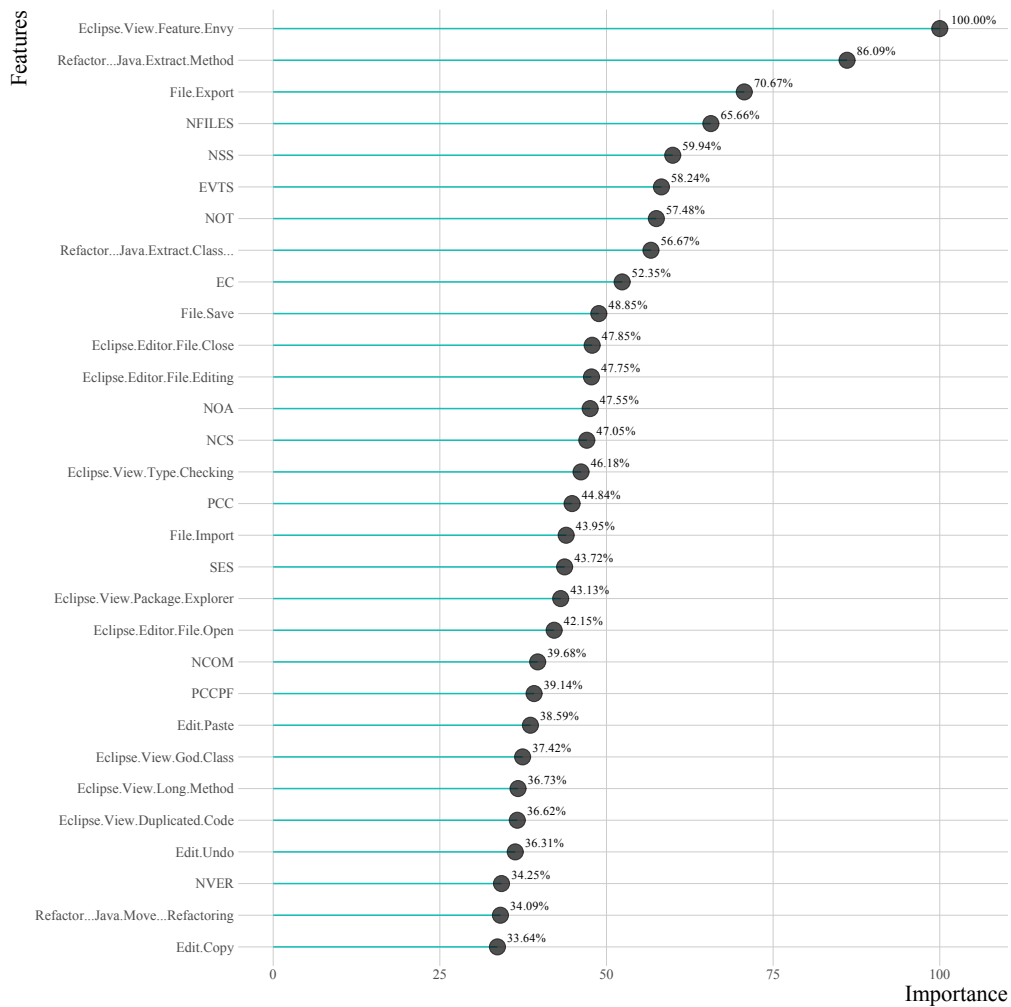

Figure 11: Feature importance for models on Table 6 (Top 30 only)

6. Threats to Validity

The following types of validity issues were considered when interpreting the results from this article.

6.1. Construct Validity

We acknowledge that this work was supported by an academic environment and using subjects with different maturity and skills which we did not assessed deeply upfront. Additionally, although some work has been done in this domain, we are still just scratching the surface in mining developers' activities using process mining tools. Some of these tools are not ready yet to automate the complete flow of: collecting data, discover processes, compute metrics and export results. As a consequence, the referred tasks were mostly done manually, thus, introducing margin for errors in the data metrics used in the experiment. To soften this, and to reduced the risk of having incoherent data, we implemented the validation of metric values from multiple perspectives. Another possible threat is related with the event data pre-processing tasks before using the Process Mining tools to discover the processes and associated metrics. Events were stored initially in a database, and from there, queries were issued to filter, aggregate and select some process related metrics. We used all the best practices in filtering and querying the data. However, there is always a small chance for the existence of an imprecise query which may have produced incorrect results and therefore, impacted our data analysis.

6.2. Internal Validity

We used a cluster analysis technique supported by the Elbow and Silhouette methods. This was used to partition the subjects according to different software and process cyclomatic complexity levels. Even if this is a valid approach, other strategies could have been followed, thus, results could vary depending on the alternative methods used, since the models computed to address RQ3 and RQ4 make use of this data partition approach. As mentioned earlier, our population was not very large and we had to use it for training and test purposes. As such, our prediction models were all trained using k-fold cross validation and using feature selection methods.

6.3. External Validity

We understood from the beginning there was a real possibility that events collected and stored in CSV/JSON files on developers' devices could be manually changed. We tried to mitigate this threat of having data tampering by using a hash function on each event at the moment of their creation. As such,

each event contain not only information about the IDE activities, but also a hash code introduced as a new property in the event for later comparison with original event data. For additional precautions regarding data losses, we implemented also real-time event streaming from the IDE to a cloud event hub.

Our initial dataset contains events collected from a group of teams when performing an academic exercise. Each user was provided with a *username* and *password* to enter in the Eclipse Plugin. With this approach, we can easily know which user was working on each part of the software and their role in the whole development process. However, we cannot guarantee that each developer used indeed their own *username*. This does not cause any invalid results in the number of activities for example, but may introduce some bias in the number of developers per team[18].

6.4. Conclusion Validity

We performed an experiment using data from 71 software teams executing well defined refactoring tasks. This involved 320 sessions of work from 117 developers. Since this is a moderate population size for this type of analysis, we acknowledge this may be a threat to generalize conclusions or make bold assertions. The Spearmmans' correlation, a nonparametric measure (therefore having less statistical power) of the strength and direction of association that exists between two variables, was done on 32 and 39 teams for automatic and manual refactoring tasks respectively. These figures, although valid, are close to the minimum admissible number of subjects for this type of analysis. Nevertheless, the insights we unveil in this study should be able to trigger additional research in order to confirm or invalidate our initial findings.

7. Conclusion

7.1. Main conclusions

Software maintenance activities, such as refactoring, are said to be very impacted by software complexity. Researchers are measuring software product and processes complexities for a long time and the methods used are frequently debated in the software development realm. However, the comprehension on the links between these two dimensions has a long journey ahead.

In this work, we tried to understand deeper the liaison of process and software complexity. Moreover, we assessed if process driven metrics and

---

[18]A metric used on almost all **RQs** and identified as having a high importance

IDE issued commands are suitable to build valid models to predict different refactoring methods and/or the expected levels of software cyclomatic complexity variance on development sessions.

We mined the software metrics from a software product after a software quality improvement task was given to a group of developers organized in teams. At the same time we collected events from those developers during the change activities they have performed within the IDE.

To the best of our knowledge, this is the first study where, using proven process mining methods, process metrics were gathered and combined with product metrics in order to understand deeper the liaison of product and process dimensions, particularly the cyclomatic complexities. Furthermore, it brings to the attention of researchers the possibility to adopt process metrics extracted from the IDE usage as a way to complement or even replace product metrics in modeling the development process.

We can't compare our study to any previous works, however, with a small set of features, we were able to unveil important correlations between product and process dimensions and obtain good models in terms of accuracy and ROC when predicting the type of refactoring done or the expected level of cyclomatic complexity variance after multiple sessions of development. We used a refactoring task as our main use case, however, by taking a snapshot of product and process metrics in different moments in time, one can measure other development practices the same way.

7.2. Relevance for practitioners

This approach can be particularly relevant in cases where product metrics are not available or are difficult to obtain. It can be also a valid approach to measure and monitor productivity within and between software teams. As we showed by analyzing the sessions complexity and the software cyclomatic complexity variance, non efficient teams can easily be detected. Our method easily support real-time data collection from individuals located in different geographic zones and with a multitude of development environments. Because the data collection is not dependent on code repositories and is decoupled from check-ins and/or commits, process and code analysis can be performed before repositories are updated. Development organizations can leverage this approach to apply conformance checking methods to verify the adherence of developers' practices with internally prescribed development processes. This facilitates mainly the detection of low performance practices and may trigger quick correction actions from project managers.

7.3. Limitations

We are aware that in this work we used only events from the IDE usage. This limits the generalization of the current method. However, our approach, although valid on it's own, may be used to complement project management analysis based on other repositories. Events from tools containing information about the documentation, project management decisions, communication between developers and managers, Q & A services, test suites and bug tracking systems, together with our method and metrics can build more robust models to comprehend development practices and the relation between software and processes followed to produce it.

7.4. Future Work

In the short term, we plan to apply a similar research approach, but in another context. Instead of refactoring an existing product, we will collect both product and process data for software development from scratch in the context of a programming contest. We intend to assess how the adopted process, both in terms of complexity and efficiency, influences effectiveness, as measured by an automatic judge.

In a previous paper [62] we described how we found that even for a well-defined software development task, there may be a great deal of process variability, due to the human factor. Less focused teams produced more complex process models, due to the spurious/non-essential actions that were carried out and therefore were less efficient. Following this path and using clustering techniques, we expect to derive a catalog of process smells and/or fingerprints to characterize development behaviors. Then, we will use that taxonomy in a personal software process dashboard, that through self-awareness is expected to foster improvement on process efficiency (e.g. less wasted effort) and effectiveness (e.g. yield better deliverables).

We also consider that the following aspects deserve further research efforts:

- **Software Repository Diversity.** Traditional software repositories have limitations and imprecisions. To expand the analytics coverage on the mining of software development processes, we should explore non trivially used repositories, such as the IDE. This is particularly interesting to drive studies aiming to combine development perspectives: i) product quality and ii) the underlying development process.

- **Software Development Process Mining Pipeline.** Many process mining tools are not ready for non-human intervention. Due to

41

this reality, many metrics in this article had to be extracted semi-automatically, using a tool but not dispensing user interaction. This is a strong limitation in advancing research based on event data and current process mining methods. A microservices-based architecture seems to be a good alternative for building a coherent pipeline for software development process mining.

- **Data Sharing.** Research combining software product and process data is scarce and experiments in this area are difficult to design and execute. To mitigate this problem, we expect an increment in shared datasets containing this hybrid data, providing that privacy and/or anonymity on sensitive information is guaranteed.

References

[1] W. E. Deming, Out of the Crisis: Quality, Productivity and Competitive Position, Massachusetts Institute of Technology, Center for advanced engineering study, 1986.

[2] K. Ishikawa, What is total quality control? The Japanese way, Prentice-Hall, 1985.

[3] G. Taguchi, Asian Productivity Organization., Introduction to quality engineering : designing quality into products and processes, The Organization, 1986.

[4] A. Fuggetta, E. Di Nitto, Software process, in: Proceedings of the on Future of Software Engineering - FOSE 2014, ACM Press, New York, New York, USA, 2014, pp. 1–12. `doi:10.1145/2593882.2593883`.

[5] W. Van Der Aalst, Process Mining: Data Science in Action, 2nd Edition, Springer-Verlag Berlin Heidelberg, 2016. `doi:10.1007/978-3-662-49851-4`.

[6] N. Moha, Y. G. Guéhéneuc, P. Leduc, Automatic generation of detection algorithms for design defects, in: 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006, 2006, pp. 297–300. `doi:10.1109/ASE.2006.22`.

[7] R. Malhotra, A. Sharma, Analyzing machine learning techniques for fault prediction using web applications, Journal of Information Processing Systems 14 (3) (2018) 751–770. `doi:10.3745/JIPS.04.0077`.

[8] H. Karna, L. Vicković, S. Gotovac, Application of data mining methods for effort estimation of software projects, in: Software - Practice and Experience, Vol. 49, John Wiley and Sons Ltd, 2019, pp. 171–191. `doi:10.1002/spe.2651`.

[9] F. A. Batarseh, A. J. Gonzalez, Predicting failures in agile software development through data analytics, Software Quality Journal 26 (1) (2018) 49–66. `doi:10.1007/s11219-015-9285-3`.

[10] M. Kersten, G. C. Murphy, Using Task Context to Improve Programmer Productivity, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, Portland, Oregon, USA, 2006, pp. 1–11.

[11] G. C. Murphy, P. Viriyakattiyaporn, D. Shepherd, Using activity traces to characterize programming behaviour beyond the lab, IEEE International Conference on Program Comprehension (2009) 90–94`doi:10.1109/ICPC.2009.5090031`.

[12] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, A Comparative Study of Manual and Automated Refactorings, in: Proceedings of the 27th European conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 2013, pp. 552–576. `doi:10.1007/978-3-642-39038-8_23`.

[13] M. Vakilian, N. Chen, S. Negara, B. Ambresh, R. Roshanak, Z. Moghaddam, R. E. Johnson, The Need for Richer Refactoring Usage Data, in: Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, Association for Computing Machinery, 2011, pp. 31–38.

[14] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, R. E. Johnson, Use, disuse, and misuse of automated refactorings, in: Proceedings - International Conference on Software Engineering, 2012, pp. 233–243. `doi:10.1109/ICSE.2012.6227190`.

[15] J. Ratzinger, T. Sigmund, P. Vorburger, H. Gall, Mining software evolution to predict refactoring, in: Proceedings - 1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, IEEE, 2007, pp. 354–363. `doi:10.1109/ESEM.2007.9`.

[16] M. Aniche, E. Maziero, R. Durelli, V. H. S. Durelli, The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring, Tech. rep., Delft University of Technology (2020).

[17] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, D. Dig, Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, in: European Conference on Object-Oriented Programming, ECOOP 2012: ECOOP 2012 – Object-Oriented Programming, Springer, Berlin, Heidelberg, 2012, pp. 79–103. `doi:10.1007/978-3-642-31057-7_5`.

[18] M. Kim, T. Zimmermann, N. Nagappan, An Empirical Study of Refactoring Challenges and Benefits at Microsoft, Transactions on Software Engineering (2014) 633–649.

[19] T. Menzies, T. Zimmermann, Software analytics: So what?, IEEE Software 30 (4) (2013) 31–37. `doi:10.1109/MS.2013.86`.

[20] M. Kim, D. Cai, S. Kim, An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution, in: 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 2011, pp. 151–160.

[21] E. Murphy-Hill, C. Parnin, A. P. Black, How We Refactor, and How We Know It, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, p. 287–297.

[22] J. Finlay, R. Pears, A. M. A. Connor, Data stream mining for predicting software build outcomes using source code metrics, Information and Software Technology 56 (2) (2014) 183–198. `doi:10.1016/j.infsof.2013.09.001`.

[23] P. Lerthathairat, N. Prompoon, An Approach for Source Code Classification Using Software Metrics and Fuzzy Logic to Improve Code Quality with Refactoring Techniques, in: International Conference on Software Engineering and Computer Systems (ICSECS 2011), 2011, p. 830.

[24] T. H. Chen, S. W. Thomas, A. E. Hassan, A survey on the use of topic models when mining software repositories, Empirical Software Engineering 21 (5) (2016) 1843–1919. `doi:10.1007/s10664-015-9402-8`.

[25] F. Peters, On Privacy and Utility while Improving Software Quality, in: International Conference on Current Trends in Theory and Practice of Computer Science, Vol. 75, SOFSEM SRF, 2017, pp. –.

[26] S. A. Fahrenkrog-Petersen, W. M. van der Aa Hanand, PRIPEL: Privacy-Preserving Event Log Publishing Including Contextual Information, in: Business Process Management, Springer International Publishing, Cham, 2020, pp. 111–128.

[27] G. Casale, C. Chesta, P. Deussen, E. Di Nitto, P. Gouvas, S. Koussouris, V. Stankovski, A. Symeonidis, V. Vlassiou, A. Zafeiropoulos, Z. Zhao, Current and Future Challenges of Software Engineering for Services and Applications, in: Procedia Computer Science, Vol. 97, Elsevier B.V., 2016, pp. 34–42. `doi:10.1016/j.procs.2016.08.278`.

[28] F. Rahman, P. Devanbu, How, and Why, Process Metrics Are Better, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, San Francisco, CA, USA, 2013, p. 432–441.

[29] J. Caldeira, F. Brito e Abreu, Software development process mining: Discovery, conformance checking and enhancement, in: Proceedings -

2016 10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016, IEEE, 2016, pp. 254–259. `doi:10.1109/QUATIC.2016.061`.

[30] F. Brito e Abreu, R. Esteves, M. Goulão, The Design of Eiffel Programs: Qualitative Evaluation using the MOOD Metrics, in: R. Ege (Ed.), Proc. of 20th International Conference on Technology of Object Oriented Languages and Systems (TOOLS'96 USA), Zenodo, Santa Barbara, USA, 1996, pp. —. `doi:10.5281/zenodo.1216932`.

[31] A. H. Watson, T. J. Mccabe, D. R. Wallace, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, Tech. rep., NIST (1996).

[32] F. Brito e Abreu, Using OCL to formalize object oriented metrics definitions, Tech. Rep. ES007/2001, INESC (5 2001).

[33] G. Botterweck, C. Werner (Eds.), Mastering Scale and Complexity in Software Reuse, Vol. 10221 LNCS, Springer Verlag, 2017. `doi:10.1007/978-3-319-56856-0`.

[34] J. Cardoso, J. Mendling, G. Neumann, H. A. Reijers, A discourse on complexity of process models, in: International Conference on Business Process Management, Springer, 2006, pp. 117–128.

[35] I. Vanderfeesten, J. Cardoso, J. Mendling, H. A. Reijers, W. Van Der Aalst, Quality Metrics for Business Process Models, Tech. rep., Technische Universiteit Eindhoven (2007).

[36] T. Menzies, T. Zimmermann, Software Analytics: What's Next?, IEEE Software Engineering (2018) 64–70.

[37] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, M. Research Asia, T. Xie, Software Analytics in Practice, IEEE Software (2013) 30–37.

[38] F. Akiyama, An Example of Software System Debugging, in: C. V. Freiman, J. E. Griffith, J. L. Rosenfeld (Eds.), IFIP Congress (1), North-Holland, 1971, pp. 353–359.

[39] T. J. Mccabe, A Complexity Measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320. `doi:10.1109/TSE.1976.233837`.

[40] T. Hariprasad, G. Vidhyagaran, K. Seenu, C. Thirumalai, Software complexity analysis using halstead metrics, in: Proceedings - International Conference on Trends in Electronics and Informatics, ICEI 2017,

Vol. 2018-January, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 1109–1113. `doi:10.1109/ICOEI.2017.8300883`.

[41] B. Curtis, S. B. Sheppard, P. Milliman, Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics, in: Proceedings of the 4th International Conference on Software Engineering, ICSE '79, IEEE Press, 1979, pp. 356–360.

[42] B. W. Boehm, Software Engineering Economics, 1st Edition, Prentice Hall PTR, Upper Saddle River, NJ, United States, 1981.

[43] S. Henry, D. Kafura, Software Structure Metrics Based on Information Flow, IEEE Trans. Softw. Eng. 7 (5) (1981) 510–518. `doi:10.1109/TSE.1981.231113`.

[44] R. P. L. Buse, T. Zimmermann, Analytics for Software Development, Tech. rep., Microsoft Research (2010).

[45] P. Mayer, M. Kirsch, M. A. Le, On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers, Journal of Software Engineering Research and Development 5 (2017) 1. `doi:10.1186/s40411-017-0035-z`.

[46] H. Henriques, H. Lourenço, V. Amaral, M. Goulão, Improving the developer experience with a low-code process modelling language, in: Proceedings - 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Association for Computing Machinery, Inc, 2018, pp. 200–210. `doi:10.1145/3239372.3239387`.

[47] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, M. Backes, The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators, in: 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 634–647. `doi:10.1109/SP.2018.00005`.

[48] M. Niazi, S. Mahmood, M. Alshayeb, M. R. Riaz, K. Faisal, N. Cerpa, S. U. Khan, I. Richardson, Challenges of project management in global software development: A client-vendor analysis, Information and Software Technology 80 (C) (2016) 1–19. `doi:10.1016/j.infsof.2016.08.002`.

[49] L. Cruz, R. Abreu, D. Lo, To the attention of mobile software developers: guess what, test your app!, Empirical Software Engineering 24 (2019) 2438–2468. `doi:10.1007/s10664-019-09701-0`.

[50] J. Herbsleb, Building a Socio-Technical Theory of Coordination: Why and How (Outstanding Research Award), in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 2–10. `doi:10.1145/2950290.2994160`.

[51] W. Poncin, A. Serebrenik, M. V. D. Brand, Process Mining Software Repositories, 2011 15th European Conference on Software Maintenance and Reengineering (2011) 5–14`doi:10.1109/CSMR.2011.5`.

[52] V. A. Rubin, I. Lomazova, W. M. P. v. d. Aalst, Agile development with software process mining, Proceedings of the 2014 International Conference on Software and System Process - ICSSP 2014 (2014) 70–74`doi:10.1145/2600821.2600842`.

[53] V. A. Rubin, A. A. Mitsyuk, I. A. Lomazova, W. M. P. van der Aalst, Process mining can be applied to software too!, Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14 (2014) 1–8`doi:10.1145/2652524.2652583`.

[54] C. Ioannou, A. Burattin, B. Weber, Mining Developers' Workflows from IDE Usage, in: Lecture Notes in Business Information Processing, Vol. 316, Springer, 2018, pp. 167–179. `doi:10.1007/978-3-319-92898-2_14`.

[55] M. Mittal, A. Sureka, MIMANSA : Process Mining Software Repositories from Student Projects in an Undergraduate Software Engineering Course Categories and Subject Descriptors, Software Engineering Education and Training — ICSE 2014 (2014) 344–353.

[56] M. Yan, X. Xia, D. Lo, A. E. Hassan, S. Li, Characterizing and identifying reverted commits, Empirical Software Engineering 24 (4) (2019) 2171–2208. `doi:10.1007/s10664-019-09688-8`.

[57] S. Hassan, C. Tantithamthavorn, C. P. Bezemer, A. E. Hassan, Studying the dialogue between users and developers of free apps in the Google Play Store, Empirical Software Engineering 23 (3) (2018) 1275–1312. `doi:10.1007/s10664-017-9538-9`.

[58] L. Bao, Z. Xing, X. Xia, D. Lo, A. E. Hassan, Inference of development activities from interaction with uninstrumented applications, Empirical Software Engineering 23 (3) (2018) 1313–1351. `doi:10.1007/s10664-017-9547-8`.

[59] K. Damevski, D. C. Shepherd, J. Schneider, L. Pollock, Mining Sequences of Developer Interactions in Visual Studio for Usage Smells, IEEE Transactions on Software Engineering 43 (4) (2017) 359–371. `doi:10.1109/TSE.2016.2592905`.

[60] M. Leemans, W. M. P. van der Aalst, M. G. J. van den Brand, The Statechart Workbench: Enabling scalable software event log analysis using process mining, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 502–506. `doi:10.1109/SANER.2018.8330248`.

[61] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li, Measuring Program Comprehension: A Large-Scale Field Study with Professionals, IEEE Transactions on Software Engineering 44 (10) (2018) 951–976. `doi:10.1109/TSE.2017.2734091`.

[62] J. Caldeira, F. Brito e Abreu, J. Reis, J. Cardoso, Assessing Software Development Teams' Efficiency using Process Mining, in: 2019 International Conference on Process Mining (ICPM), Institute of Electrical and Electronics Engineers (IEEE), 2019, pp. 65–72. `doi:10.1109/icpm.2019.00020`.

[63] P. Ardimento, M. L. Bernardi, M. Cimitile, F. M. Maggi, Evaluating Coding Behavior in Software Development Processes: A Process Mining Approach, in: 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP), IEEE, 2019, pp. 84–93. `doi:10.1109/ICSSP.2019.00020`.

[64] P. Ardimento, M. L. Bernardi, M. Cimitile, G. D. Ruvo, Learning analytics to improve coding abilities: A fuzzy-based process mining approach, in: IEEE International Conference on Fuzzy Systems, Vol. 2019-June, Institute of Electrical and Electronics Engineers Inc., 2019, pp. 1–7. `doi:10.1109/FUZZ-IEEE.2019.8859009`.

[65] P. Ardimento, M. L. Bernardi, M. Cimitile, G. De Ruvo, Mining Developer's Behavior from Web-Based IDE Logs, in: 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for

Collaborative Enterprises (WETICE), IEEE, 2019, pp. 277–282. `doi:`
`10.1109/WETICE.2019.00065`.

[66] K. Beck, M. Fowler, J. Brant, W. Opdyke, D. Roberts, Bad Smells in
Code, in: Improving the design of existing code, O'Reilly, 1999, Ch. 3,
pp. –.

[67] M. Chinosi, A. Trombetta, BPMN: An introduction to the standard,
Computer Standards & Interfaces 34 (1) (2012) 124–134.

[68] C. Günther, E. Verbeek, XES standard definition, Tech. rep., BPMcen-
ter. org (2014).

[69] A. R. C. Maita, L. C. Martins, C. R. López Paz, L. Rafferty, P. C. K.
Hung, S. M. Peres, M. Fantinato, A systematic mapping study of process
mining, Enterprise Information Systems 12 (5) (2018) 505–549. `doi:`
`10.1080/17517575.2017.1402371`.

[70] C. d. S. Garcia, A. Meincheim, E. R. Faria Junior, M. R. Dalla-
gassa, D. M. V. Sato, D. R. Carvalho, E. A. P. Santos, E. E. Scal-
abrin, Process mining techniques and applications – A systematic map-
ping study, Expert Systems with Applications 133 (2019) 260–295.
`doi:10.1016/j.eswa.2019.05.003`.

[71] M. Leemans, W. M. P. van der Aalst, M. G. J. van den Brand, Re-
cursion aware modeling and discovery for hierarchical software event
log analysis, in: 2018 IEEE 25th International Conference on Software
Analysis, Evolution and Reengineering (SANER), 2018, pp. 185–196.
`doi:10.1109/SANER.2018.8330208`.

[72] B. Purnima, K. Arvind, EBK-Means: A Clustering Technique based on
Elbow Method and K-Means in WSN, International Journal of Com-
puter Applications 105 (9) (2014) 17–24.

[73] N. Kaoungku, K. Suksut, R. Chanklan, K. Kerdprasop, N. Kerdprasop,
The silhouette width criterion for clustering and association mining to
select image features, International Journal of Machine Learning and
Computing 8 (1) (2018) 69–73. `doi:10.18178/ijmlc.2018.8.1.665`.

[74] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen, T. Menzies, Hyperpa-
rameter Optimization for Effort Estimation, Tech. rep., North Carolina
State University (4 2018).

[75] L. L. Minku, S. Hou, Clustering dycom an online cross-company software effort estimation study, in: ACM International Conference Proceeding Series, Association for Computing Machinery, 2017, pp. 12–21. `doi: 10.1145/3127005.3127007`.

[76] S. B. Jagtap, B. G. Kodge, Census Data Mining and Data Analysis using WEKA, in: International Conference in Emerging Trends in Science, Technology and Management-2013, Singapore, 2013, pp. –.

[77] C. Thornton, H. H. Hoos, K. Leyton-Brown, Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA, Journal of Machine Learning Research 1 (2017) 429.

[78] L. Madeyski, M. Jureczko, M. Jureczko, Which process metrics can significantly improve defect prediction models? An empirical study, Software Quality Journal 23 (2015) 393–422. `doi:10.1007/ s11219-014-9241-7`.

Appendix A. Appendix

Appendix A.1. Product Metrics

Table A.7: Product Metrics Description

Name	Description	Scale
VG	McCabe Cyclomatic Complexity (Avg. per Method)	Numeric
PAR	Number of Parameters (Avg. per Method)	Numeric
NBD	Nested Block Depth (Avg. per Method)	Numeric
CA	Afferent Coupling (Avg. per Package Fragment)	Numeric
CE	Efferent Coupling (Avg. per Package Fragment)	Numeric
RMI	Instability (Avg. per Package Fragment)	Numeric
RMA	Abstractness (Avg. per Package Fragment)	Numeric
RMD	Normalized Distance (Avg. per Package Fragment)	Numeric
DIT	Depth of Inheritance Tree (Avg. per Type)	Numeric
WMC	Weighted methods per Class (Avg. per Type)	Numeric
NSC	Number of Children (Avg. per Type)	Numeric
NORM	Number of Overridden Methods (Avg. per Type)	Numeric
LCOM	Lack of Cohesion of Methods (Avg. per Type)	Numeric
NOF	Number of Attributes (Avg. per Type)	Numeric
NSF	Number of Static Attributes (Avg. per Type)	Numeric
SIX	Specialization Index (Avg. per Type)	Numeric
NOP	Number of Packages	Numeric
NOC	Number of Classes (Avg. per Package Fragment)	Numeric
NOI	Number of Interfaces (Avg. per Package Fragment)	Numeric
NOM	Number of Methods (Avg. per Type)	Numeric
NSM	Number of Static Methods (Avg. per Type)	Numeric
MLOC	Method Lines of Code (Avg. per Method)	Numeric
TLOC	Total Lines of Code	Numeric
TLOC	Total Lines of Code	Numeric
VG_LEVEL	Different levels of **ΔVG** (**LOW, MEDIUM, HIGH**)	Categorical

Appendix A.2. Process Metrics

Table A.8: Process Metrics Description

Name	Description	Scale
DEV	Number of Developers	Numeric
SES	Number of User/Development Sessions	Numeric
EVTS	Number of Events Collected	Numeric
NFILES	Number of Unique Files Touched	Numeric
NCOM	Number of Unique Commands Issued in IDE	Numeric
PCCPF	Process Cyclomatic Complexity per File Touched	Numeric
EC	Number of Event Classes	Numeric
NOA	Number of Activities	Numeric
NSS	Number of Simple States	Numeric
NCS	Number of Composite States	Numeric
NOT	Number of Transitions	Numeric
PCC	Process Cyclomatic Complexity	Numeric
NVER	Number of Unique IDE Versions	Numeric
NCAT	Number of Unique Command Categories	Numeric
NPLA	Number of Unique IDE Platforms	Numeric
NISP	Number of Unique Geographic Locations	Numeric
NOS	Number of Unique Operating Systems	Numeric
NPER	Number of Unique Perspectives used in the IDE	Numeric
PCC_LEVEL	Different levels of **PCC** (**LOW, HIGH**)	Categorical

Table A.9: Process-Extended Metrics Description

Category	Name	Scale
Refactor	Java-Extract Method	Numeric
	Java-Move - Refactoring	Numeric
	Java-Extract Class...	Numeric
	Java-Rename - Refactoring	Numeric
	Delete Resources	Numeric
	Java-Encapsulate Field	Numeric
	Java-Change Method Signature	Numeric
	Java-Move Type to New File	Numeric
Eclipse Editor	File Open	Numeric
	File Editing	Numeric
	File Close	Numeric
Eclipse View	Project Explorer	Numeric
	Package Explorer	Numeric
	Long Method	Numeric
	God Class	Numeric
	Code Smell Visualization	Numeric
	Type Checking	Numeric
	Feature Envy	Numeric
	Duplicated Code	Numeric
Edit	Find and Replace	Numeric
	Copy	Numeric
	Paste	Numeric
	Cut	Numeric
	Delete	Numeric
	Undo	Numeric
	Redo	Numeric
File	Import	Numeric
	Refresh	Numeric
	Save	Numeric
	Save All	Numeric
Source	Generate Getters and Setters	Numeric
Compare	Select Next Change	Numeric
....	//List is truncated on purpose	
....	//List size is ≈ 250	
Text Editing	Delete Previous Word	Numeric

Appendix A.3. Algorithms shown in Model Evaluations

   RandomCommittee. Method for building an ensemble of randomizable base classifiers. Each base classifier is built using a different random seed number (but based one the same data). The final prediction is a straight average of the predictions generated by the individual base classifiers.

RandomSubSpace. This method constructs a decision tree based classifier that maintains highest accuracy on training data and improves on generalization accuracy as it grows in complexity. The classifier consists of multiple trees constructed systematically by pseudo-randomly selecting subsets of components of the feature vector, that is, trees constructed in randomly chosen sub-spaces.

RandomForest. Method for constructing a forest of random trees. It consists of a learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

RepTree. Fast decision tree learner. Builds a decision/regression tree using information gain/variance and prunes it using reduced-error pruning (with back-fitting). Only sorts values for numeric attributes once. Missing values are dealt with by splitting the corresponding instances into pieces.

LMT. 'Logistic Model Trees' are classification trees with logistic regression functions at the leaves. The algorithm can deal with binary and multi-class target variables, numeric and nominal attributes and missing values.

Logistic Regression. Method for building and using a multinomial logistic regression model with a ridge estimator. Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although more complex extensions exist.

LWL. The Locally Weighted Learning method uses an instance-based algorithm to assign instance weights which are then used by a specified WeightedInstancesHandler. Can do classification (e.g. using naive Bayes) or regression (e.g. using linear regression).

LinearNNSearch. This method implements the brute force search algorithm for nearest neighbour search.

DecisionTable. Builds and uses a simple decision table majority classifier.

Bagging. Method for bagging a classifier to reduce variance. Can do classification and regression depending on the base learner.

KStar. Is an instance-based classifier, that is, the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function.

Appendix A.4. Best-Fit Models - Source Code

Listing 2: Best-Fit Model Code for Refactoring Practice Detection

```java
/** Java code to implement the best model found. */

/** Attribute Search **/
AttributeSelection as = new AttributeSelection();
ASSearch asSearch = ASSearch.forName("weka.attributeSelection.GreedyStepwise
    ", new String[]{"-C", "-R"});
as.setSearch(asSearch);

/** Attribute Evaluation and Selection **/
ASEvaluation asEval = ASEvaluation.forName("weka.attributeSelection.
    CfsSubsetEval", new String[]{"-M", "-L"});
as.setEvaluator(asEval);
as.SelectAttributes(instances);

/** Reduce Dimensions **/
instances = as.reduceDimensionality(instances);

/** Build Classifier **/
Classifier classifier = AbstractClassifier.forName("weka.classifiers.meta.
    RandomCommittee", new String[]{"-I", "64", "-S", "1", "-W", "weka.
    classifiers.trees.RandomForest", "--", "-I", "29", "-K", "13", "-
    depth", "3"});
classifier.buildClassifier(instances);
```

Listing 3: Best-Fit Model Code for expected Cyclomatic Complexity level detection

```java
/** Java code to implement the best model found. */

/** Attribute Search **/
AttributeSelection as = new AttributeSelection();
ASSearch asSearch = ASSearch.forName("weka.attributeSelection.GreedyStepwise
    ", new String[]{"-C", "-R"});
as.setSearch(asSearch);

/** Attribute Evaluation and Selection **/
ASEvaluation asEval = ASEvaluation.forName("weka.attributeSelection.
    CfsSubsetEval", new String[]{"-L"});
as.setEvaluator(asEval);
as.SelectAttributes(instances);

/** Reduce Dimensions **/
instances = as.reduceDimensionality(instances);

/** Build Classifier **/
Classifier classifier = AbstractClassifier.forName("weka.classifiers.lazy.
    LWL", new String[]{"-K", "60", "-A", "weka.core.neighboursearch.
    LinearNNSearch", "-W", "weka.classifiers.rules.DecisionTable", "--",
    "-E", "auc", "-S", "weka.attributeSelection.GreedyStepwise", "-X", "2
    "});
classifier.buildClassifier(instances);
```