electronics

MDPI

Article

Understanding Coding Behavior: An Incremental Process Mining Approach

Pasquale Ardimento [1,*,†], Mario Luca Bernardi [2,†], Marta Cimitile [3,†], Domenico Redavid [1,†] and Stefano Ferilli [1,*,†]

[1] Department of Computer Science, University of Bari Aldo Moro, 70125 Bari, Italy; domenico.redavid1@uniba.it
[2] Department of Engineering, University of Sannio, 82100 Benevento, Italy; bernardi@unisannio.it
[3] Department of Economy and Law, Unitelma Sapienza University of Rome, 00185 Rome, Italy; marta.cimitile@unitelmasapienza.it
* Correspondence: pasquale.ardimento@uniba.it (P.A.); stefano.ferilli@uniba.it (S.F.)
† These authors contributed equally to this work.

Abstract: Capturing and analyzing interaction data in real-time from development environments can help in understanding how programmers handle coding activities. We propose the use of process mining to learn coding behavior from event logs captured from a customized Integrated Development Environment, concerning interactions with both such an environment and a Version Control System. In particular, by using an incremental approach, the discovered model can be refined after every single development session, which avoids the need to for the model to learn from scratch from previous sessions. It would also allow one to provide the programmer timely suggestions to improve their performance. In this paper, we applied off-line incremental behavior, so as to be able to analyze it at several levels of depth and at different moments. As a preliminary evaluation of our approach, we investigated the coding activities of six novice students of a Java academic programming course working on a programming case study. The results provide some useful information about the initial difficulties in coding activities faced by programmers and show that their coding behavior could be considered as "formed" after a development task requiring approximately 4000 rows of code.

Keywords: process mining; source code; process discovery; coding behavior

Citation: Ardimento, P.; Bernardi, M.L.; Cimitile, M.; Redavid, D.; Ferilli, S. Understanding Coding Behavior: An Incremental Process Mining Approach. *Electronics* **2022**, *11*, 389. https://doi.org/10.3390/electronics11030389

Academic Editors: Abrar Ullah, Ryad Soobhany, Sajid Anwar and Imran Razzak

Received: 23 December 2021
Accepted: 26 January 2022
Published: 27 January 2022

1. Introduction

Improving the quality of programmers' code is one of the mandatory objectives pursued by all software organizations. Code quality can be evaluated by two different, but no alternative, perspectives. The former concerns the outcome of the coding activity, that is the source code. Source code evaluation can be made in many ways, such as inspection reviews, to detect defects through a manual examination of source code written by different programmers, or evaluating the quality of code by means of source code quality metrics. The latter perspective, in contrast, concerns the process used by programmers to write the source code. In fact, novice programmers usually approach coding problems in a way that is different from expert programmers. For example, novice programmers need to remember language syntax, and for this reason, quickly learning the usage of contextual help facilities is important for guidance in completing partially entered code. In addition, they can face hurdles in debugging activities because they usually lack a thorough understanding of what should happen. For this reason, knowing both the hurdles faced by novice programmers, and how they face them, could be useful in providing them suggestions about how to improve their performance. As a consequence, capturing and mining the interactions of programmers with the Integrated Development Environments (or IDEs for short) also helps one to understand the behavior of the software being written [1–4]. Therefore, software coding is a process that is expected to comply with some model, expressing the appropriate

combinations of activities that can be carried out, and/or that should be carried out by programmers to support the creation of a quality product. Since such a model may depend on the development context, it is often not formally specified, because it would be complex and costly to produce. For these reasons, we propose a process mining approach to extract information about programmers' behavior in developing a stand-alone application. In particular, we focus on the coding carried out by novice programmers, and specifically students working on a programming case study, where understanding their underlying process model may help in their evaluation, and having a 'correct' model available may allow one to provide them helpful indications when they take wrong or unsuitable actions.

The core of our approach is capturing and mining the interactions of programmers with Integrated Development Environments (IDEs) and with a Version Control System (VCS) so as to monitor programmers' behavior when they are involved in performing programming tasks [5,6]. The analysis of such models should give a great contribution to understanding and evaluating coding activities and drive the manager in identifying pitfalls and technically sound practices. Thus, this study also proposes a framework to capture the interactions of a programmer involved in a development task with both the Eclipse IDE and the Version Control System (VCS) and record all of them in an event log. The logs are then mined using an incremental process mining technique [7–9] to discover the coding workflow carried out by each programmer and support the managers in their evaluation and understanding of the behavior of single programmers.

While the process mining approach to modeling coding activities of students and programmers that interact with IDEs was already adopted in previous works [6,10,11], the novelty introduced in this paper consists in the application, for the first time, of an *incremental* process mining approach to extract the student development workflows from both the IDE and VCS. The incremental approach allows one to quickly understand how novice programmers interact with the IDE and, in case of need, provide timely feedback to them. To provide a preliminary evaluation of the proposed approach, we collected event logs captured in the development sessions of a case study assigned to six students attending a second-year course for programmers learning the Object-Oriented Paradigm and the Java language, held at the University of Bari Aldo Moro. The focus of this work is more on checking whether our approach can provide relevant and useful information to better understand the behavior of specific programmers/students, than on obtaining a general model that captures the most widespread (and possibly correct) behavior, or that allows one to draw reliable statistics on the most common behaviors. To this purpose, we consider six students to be a sufficient number to allow the analysis of different cases while still providing an insight into the behavior of each case.

In summary, this paper proposes the use of process mining to understand if the interaction behavior of programmers with the IDE interface while coding software could be improved to enhance the efficiency and quality of the source code they write. Its main contributions are:

- Presenting the application of incremental process mining techniques that can prospectively be used online to provide software programmers feedback to immediately understand wrong behaviors and suggest corrective actions to take;
- Evaluating a specific incremental process mining system to assess its performance and scalability, and to understand whether it is a suitable candidate for supporting the previous goal;
- Studying specific cases of application of the proposed approach and system.

The rest of the paper is structured as follows. Section 2 presents and discusses the main related work on existing approaches to capture and model the interactions of software programmers. Section 3 reports first on background information on process mining in general, and then on the framework used in this paper. Then, Section 4 shows how the proposed framework was applied for the specific purposes of this paper, while in Section 5 a case study is presented and discussed. Section 6 discusses the threats to the validity of our study and finally, Section 7 draws conclusions.

2. Related Work

This paper proposes the use of process mining techniques to support software programmers in their work and improve the usability of IDE platforms by learning models of behavior and checking new executions against these models. There are several studies on programmers' behavior based on tools installed in the workstation of programmers to collect data automatically from the software development environment. These studies generally focus on the IDEs to understand what tool and source of information programmers need in their maintenance activities. Many of these works have been carried out by relying on the Eclipse IDE interaction data collected by the Mylyn tool. Eclipse Mylyn [12] is a tool that allows a programmer to record their activities in a task, such as bug fixing or developing a new functionality, while they are working on it. Each Mylyn task has a context that captures the involved classes, methods, and the cursor position in the opened Java or text editor. One of the first studies [13] based on the use of Mylyn, previously called Mylar, investigates how programmers use the Eclipse IDE. The main results of this study are: (i) The three most used views are the package explorer, the console, and the search one, while the Declaration view is never used; (ii) the three most used perspectives are Java, Debug, and Team synchronization, while the Java Type Hierarchy and Java Browsing perspectives are never used; (iii) the most used commands in the IDE are 'delete', 'save', and 'copy', and (iv) all but three used the six debugging views, the less-used views were the Expressions view and Display view. In [14], instead, the authors used Mylyn to analyze the time spent by programmers to perform a task. The authors discovered that on average only 38% of files opened are significantly related to the final implementation of the task and that programmers who opened a larger number of files than those required for the solution to a task take a longer time to complete the task.

Bavota et al. [15] performed a controlled experiment, with 33 undergraduate students, to investigate how they navigate different sources of documentation (Javadoc, java source files, sequence diagrams, and use cases) available in the IDE while performing a development task that requires to identify classes and class elements to be changed. The study discovered that students usually do not use a waterfall approach, a sequential development approach with each phase completely wrapping up before the next phase begins. They usually start from the source code and then browse back and forth between the source code and design documents (class and sequence diagrams). Ying and Robillard [16] studied whether a relationship existed between the nature of a task and when a programmer edits code during a programming session. In this regard, they used the Mylyn tool to analyze the interaction history of over 4000 programming sessions that are attachments of bug reports. They found that different types of tasks, such as enhancement tasks, and minor and major bug fixes, were associated with different editing styles. For example, "knowing a programming session being EDIT-FIRST, the development environment could show more of the editing-related features, rather than the navigation-related features". They conclude by stating that the IDE knowing the editing style of a task should dynamically self-configure to show only the most relevant parts to the particular editing style. The coding behavior of refactoring tasks was studied in [17], where the authors analyzed eight different datasets. The most significant findings are: Refactorings performed using a tool that occurs in batches in about 40 percent of cases; only 10 percent of configuration defaults in refactoring tools are changed when programmers use the tools; commit messages in almost all cases do not report the presence of refactoring in the commit; and refactorings performed manually differ from the kind performed with a tool.

In [18], the authors conducted an observational study in an industrial context, capturing programmers' interaction with the IDE and other applications such as web-browsers and web processors, and related them with activities performed on source code files. The main findings of their study suggest that: (i) Programmers rarely use online help, showing only 6% of total activities executed; (ii) programmers often execute the system under development after working on code, likely to verify the effect of applied changes on the system's behavior; and (iii) programmers tend to execute the system when they work on files highly

coupled or complex. Finally, the study of programmers' coding behavior based on process mining techniques was evaluated in [6,10]. Here the authors use a declarative approach to mine students' coding activities, beginning with the information extracted by the IDE.

With these contributions, the approach proposed in this study presents the following main novelties. The first novelty consists in the use of the data extracted from both the IDE and VCS. The second novelty regards the mining algorithm used. It is declarative but is based on full incrementality. It can begin learning from an empty model and from the first development session, without the need for a number of cases to draw significant statistics before learning starts, unlike batch techniques that need large training sets to learn their models. Additionally, it can refine an existing programmers' coding model, adding new cases from other development sessions. This would allow novice programmers, such as students, to quickly receive timely information about their coding behaviors. With proper guidance, provided by teachers, students could receive frequent feedback about their development sessions. Finally, none of the studies known in the literature has been conducted for a time comparable to the study presented in this paper.

3. Background

This section provides the essential notions needed to follow the incremental process mining approach on which our research is based.

3.1. Process Mining

Research on process mining has been carried out in two directions: The formalisms for representing processes, and the algorithms to learn and handle process models. Starting in the 90s, research on process model discovery has progressively refined the setting and approaches for the task. After initial attempts to reuse existing research on Finite State Automata and Hidden Markov Models, which cannot fully capture the complexity of processes (e.g., concurrency), specific techniques have been developed. Here we will mention a few with the aim of providing a landscape of different approaches, rather than discussing them in-depth. A more extensive overview of the field can be found in [19,20].

The formalism most widely used in the literature for representing process models is Petri Nets. Often, their restriction Workflow Nets [21], purposely defined for workflow models, is exploited. It allows for expressing alternative and concurrent executions. Whilst in Petri or Workflow Nets a process model is described as a graph, where tasks are associated to one kind of vertexes and the remaining vertexes and edges express possible flows of execution among tasks, 'declarative' process mining approaches [22] adopt a de-structured representation in which the overall model is described as a set of constraints, typically expressed as formal logics formulas. This approach may improve flexibility, since there is no monolithic model (it is simply required that the constraints are satisfied by process executions), and support interpretability (since each constraint should be easily understandable by humans).

Concerning the process mining algorithms, various solutions have been proposed for the Petri/Workflow Net formalism. The traditional approach used statistics on the order of execution of pairs of tasks to piece-wise infer the structure of a Workflow Net [21,23]. This proved inaccurate when the process involves many parallel tasks and/or nested loops [7]. Other approaches were more theory-driven, defining classes of models of limited complexity but guaranteeing that all models in those classes could be learned by given algorithms [24–26]. These solutions significantly reduced the allowed power of the models. Yet other approaches resorted to traditional machine learning techniques, such as genetic algorithms [27,28], but again with limitations on the kinds of schemes that could be learned, and additionally introducing serious efficiency issues. The ProM suite (https://promtools.org/doku.php, accessed on 6 December 2021) provides ready-to-use implementations of many algorithms in the literature, most of which learn models expressed as Petri/Workflow Nets.

As to the declarative setting, the Declarative Process Model Learner (DPML) [29] is worth mentioning. It needs also negative examples, which are not standard in process mining [28]. Moreover, specifying examples, background knowledge, and language bias might be difficult for the user. WoMan [7] proposed a quite peculiar declarative formalism. Declare is currently the most famous declarative solution for process mining [30]. The importance of efficient and declarative approaches was also identified [31].

As in the more general machine learning landscape, the vast majority of the proposed solutions works in batch mode, meaning that the whole set of training cases must be available to the system before it starts its operation; the final model is returned in a single learning session and cannot be subsequently adjusted or refined if new data become available in time. This setting simplifies the operation of the system, since having all the information available since the beginning, and assuming nothing else is missing, allows it to draw statistics and check hypotheses before committing to specific modeling choices. On the other hand, in most real-world situations, data become available in time and batch approaches should learn from scratch a new model from old and new data. Not only would this cause significant inefficiency, it would also prevent immediate model responsiveness to changes in the environment and would prevent their online ability to continuously track process executions and possibly notify conformance failures as well as suggest possible corrections or even driving the agents in performing correct processes.

Exceptions to this landscape exist. Here we mention the incremental version of DPML [32] and, again, WoMan [7]. The former suffers from the same issues as DPML. The latter proved able to learn very complex process models, both on toy problems and in many real-world domains (including Ambient Intelligence, Recommendation Systems, and even Chess Playing), also in cases in which other state-of-the-art systems could not. It also provides many additional features (including noise handling, context-based pre-and post-condition handling, and activity and process prediction). Last but not least, it carried out its tasks quite efficiently.

More recently, the focus of the research community has drifted from the discovery of the structure of process models to the extension of the features to be considered in the model, so as to be able to handle more complex domains. In addition to the traditional problem of noise handling, which affects all real-world endeavors, the investigations concerned guards (i.e., conditions for carrying out the tasks), the use of contextual information, the data, and agents that are involved in the process, etc. In addition, other kinds of tasks have been considered, such as model analysis, formalisms refinement and extension, and real-world applications. In this paper, we will focus only on the process model discovery task and on the flow of activities in process execution, and thus will not delve further into these issues.

Given this landscape, we decided to base this research on the WoMan framework, primarily because it is incremental, efficient, and able to learn complex models in very variable real-world domains (as we expect for the coding activities).

3.2. WoMan Framework

WoMan is a framework, also implemented in a system, that covers a wide range of tasks involved in Process Management:

Conformance Checking to check whether a new process execution is compliant to a given process model;
Workflow Discovery to learn a process model starting from descriptions of specific executions;
Activity Prediction to guess what will be the next activities that are likely to happen at a given point of a process execution, according to a given process model;
Process Prediction to guess the process that is likely to be enacted in a given (even partial) execution, among those specified in a given set of models;
Simulation to generate possible cases that are compliant with a given model, also considering the likelihood of the different tasks and of their allowed combinations.

WoMan is *incremental*, meaning that conformance checking, activity prediction, and process prediction can be carried out at any moment in time during a case execution and

that workflow discovery can be carried out after every single case is available, refining an existing model instead of having to learn it from scratch from all training cases each time. Even more so, WoMan is *inherently* incremental; even if only complete cases and/or entire training sets are provided, it still processes them event by event and case by case.

WoMan is a declarative framework, in which process models are expressed as sets of first-order logic constraints, rather than as overall graphs of activities with a fixed structure (e.g., Petri Nets). The core concepts in WoMan's models are so-called *transitions*, of the form:

$$(\{I_1, \ldots, I_n\}, \{O_1, \ldots, O_m\})$$

meaning that after ending the concurrent execution of activities $I = \{I_1, \ldots, I_n\}$, the concurrent execution of activities $O = \{O_1, \ldots, O_m\}$ can be started, without any other activity being completely executed in the middle. I and O are multisets, and thus the specific order of execution of the activities they specify is irrelevant, and they may include many occurrences of the same tasks (to be intended as different concurrent executions of the same task). For all tasks and transitions, several kinds of information and statistics are maintained by WoMan, concerning their frequency, mutual connections, period of applicability (intended as time or a number of other activities carried out from the start of process execution), the maximum number of executions, conditions of applicability, and outcomes.

An incremental approach to process management requires fine-grained information about the relevant events to be provided to the system. The WoMan framework adopted an input formalism in which log elements are 7-tuples of the form:

$$\langle T, E, P, C, A, N, R \rangle$$

where:

T is the timestamp of the event; timestamps must be unique for each log entry, and in principle should be directly proportional to the time passed from the start of the process execution, so as to allow the system to draw statistics about the timing and duration of the activities.

E is the type of event; 6 types of events are handled by the system:

- **begin_process** involves specifying that a process execution has been started;
- **begin_activity** involves specifying that a task execution (i.e., an 'activity') has been started;
- **end_activity** involves specifying that an activity has been completed;
- **atomic_activity** involves specifying that an activity whose time span is not significant has been executed;
- **context_description** involves specifying that contextual information is being provided, other than activity execution;
- **end_process** involves specifying that a process execution has been completed.

P is the identifier of the process the event refers to; each different process handled by the system must have a different identifier that will be used by the system to name the corresponding model.

C is the identifier of the process execution (the 'case') the event refers to; each different case for a given process handled by the system must have a different identifier that will be used by the system to denote the corresponding trace and information.

A If $E \in \{$ **begin_activity**, **end_activity** $\}$, it is the name of a task relevant to process P. If $E =$ **context_description**, it is a set of first-order logic atoms built on domain-specific predicates that describe relevant contextual information associated with the timestamp, to be interpreted as the logical conjunction (AND) of the atoms.

N If $E \in \{$ **begin_activity**, **end_activity** $\}$, it is a progressive integer distinguishing the activities in the same case associated with the same task; this is relevant in the case of

many concurrent executions of the same task, to properly match their **begin_activity** and **end_activity** events. If $E =$ **begin_process**, this parameter can be used to specify a noise tolerance threshold in $[0, 1]$, such that the process model components with frequency less than N are considered as noise and ignored during compliance checks.

R is the role of the agent that is carrying out the activity or an identifier of the specific agent (the 'resource'); the system can be provided with the associations of resources to roles and with a taxonomy of roles, so as to allow generalizations over them. If agents are not relevant, or unknown, for the process model, this information can be dropped, and thus log items are 6-tuples rather than 7-tuples.

More specifically, each entry is expressed as a first-order logic fact of the form:

$$\texttt{entry}(T, E, P, C, A, N, R)\,.$$

In addition to providing the system with an entire log file including entries for different cases, possibly interleaved to one another, each entry can be provided to the system by itself, and the system will collect and suitably organize all the entries for each case. This is the true incremental setting. For each entry, the system compares the corresponding event information to the model considering the current status of the case, updates the current status, and returns a compliance outcome among the following options:

ok the event is compliant with the process model;

warning the event is not compliant with the process model, for one of several reasons:

- *task*: The event has started the execution of a task that is not accounted for in the current model;
- *transition*: The event has started an activity for a task that is recognized by the model, but not applicable in the current status of the case;
- *loop*: A task or transition was run more times than expected;
- *time*: The event started or terminated a task, or a transition, outside the allowed period (with some tolerance) specified by the model;
- *step*: The event started or terminated a task, or a transition, outside the allowed step interval required by the model (each activity in the case increments the number of steps carried out thus far);
- *agent*: The activity was carried out by an agent whose role is not allowed by the current model for that task in general, or for that task in the specific transition that occurred;
- *pre-condition* or *post-condition*: Various kinds of pre or post conditions (learned by the system from the structure or context of past process executions) for carrying out a task, or a transition, or a task in a specific transition, are not satisfied.

Each warning carries a different degree of severity, expressed as a numeric weight. In particular, some warnings encompass others (e.g., a new task also implies a new transition), and thus have a greater severity degree than the others.

error the event cannot be processed, due to one of the following reasons:

- Incorrect syntax of the entry;
- Unknown process or case;
- Termination of an activity that was never started;
- Completion of a case while activities are still running.

Between events, users may ask the system for a prediction (that can be used also as a suggestion) of the next activities that are likely to be started, or of the kind of process that is being executed among a set of candidate processes whose models are available to the system. The latter can be used when the current execution might belong to different processes and the aim is classifying it. After all the events of a case have been processed (i.e., after its **end_process** event), the system may be asked to use the case for refining the model. In such a case, the compliance of the next case for that process will be checked against the updated model, as required by a truly incremental setting.

4. Proposed Approach

This section describes the overall architecture of the process mining framework proposed. Figure 1 provides an overview of the architecture based on IDE instrumentation and the WoMan tool. The main components are an Eclipse plugin called Log Processor, a Log Exporter subsystem, and the WoMan miner tool. The Log Processor is the component that captures the Human-Computer Interaction (HCI) events and stores the collected logs in a development sessions logs repository. HCI events are the stream of activities performed by the programmers when they interact with the development environment. All the collected logs need to be refined in order to apply process mining. Specifically, the following steps are executed: (1) Interactions that are unrelated with development activities in the programmer workspace are removed; (2) inconsistencies in the data are detected and corrected; (3) HCI events are integrated with commits events; are (4) the event logs are converted into the WoMan format. The WoMan logs represents the coding behavior of the programmers. The traces coming from new developers can be analyzed using the Workflow Discovery functionality of the WoMan tool to update the coding process model executed by these programmers.

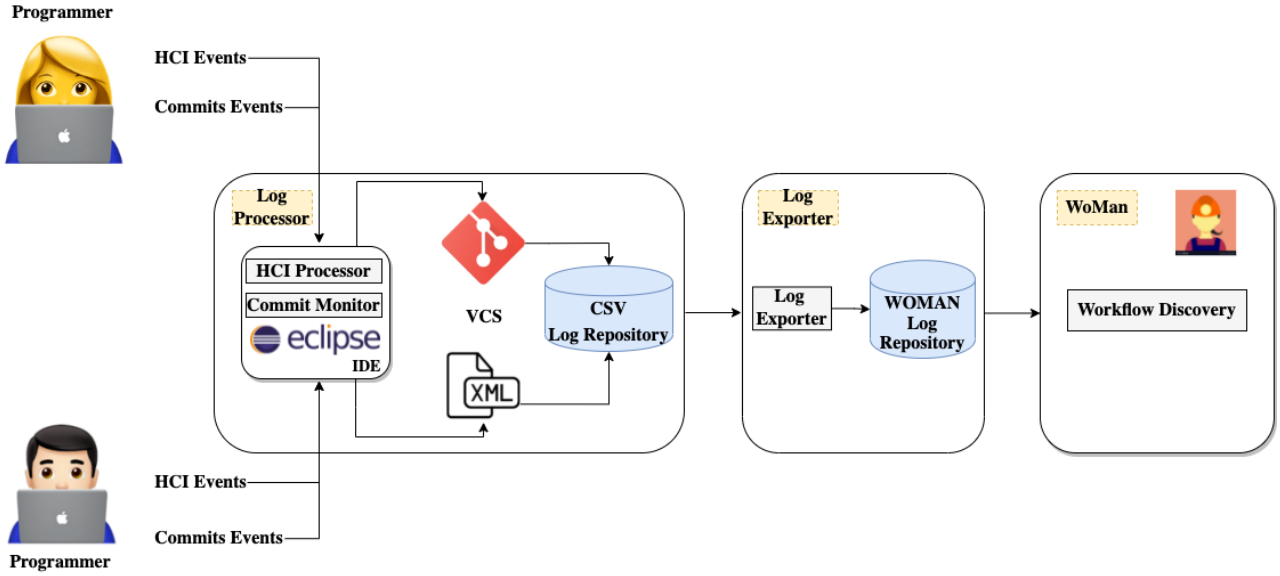

Figure 1. The environment architecture.

4.1. Log Processor

In [6], the authors described the "Log processor", an Eclipse plugin built as an extension of the Fluorite plugin developed in the School of Computer Science at Carnegie Mellon University [33]. Fluorite is capable of recording, in log files in XML format, all of the low-level events when using the Eclipse code editor without interrupting the coding activities. The Log Processor extends Fluorite by: (i) Adding contextual information to low-level events such a case ID to group all events executed by a programmer during a single development session; (ii) capturing also high-level events (such as create or open a file, close a project, open a view, reset a perspective); and (iii) capturing any interaction event with VCS (e.g., commits and pull events). A Log Processor keeps track of all these events that occur in the code editor and saves the log files in the CSV format. The Commit events are also stored in a remote repository, hosted either on the GitHub platform or BitBucket, and managed by a VCS for collaborative software development. Log Processor retrieves the commits events from the remote repositories and merges them with the events captured in the IDE. The merge between HCI events and Commit events is based on two fields: The case ID and programmer ID. Then, all the captured interactions are cleaned and filtered out. The cleaning activity consists of removing the programmers' interactions that are unrelated to the project's activity and fixing inconsistencies. Finally, the interactions are

stored in a CSV format, one for each programmer to become event logs. Further details on high and low events captured by the Log Process are provided in [6].

4.2. Log Exporter

All the log events are then processed by the Log Exporter, a module able to export in different formats the logs captured [6,10]. In this work, an extension of the Log Exporter is proposed to export logs in a customized WoMan format so that the WoMan miner can discover the process from the logs captured. To process the log events, the following input formalism is adopted:

$$\langle T, E, P, C, A, N \rangle$$

where:

T is the timestamp of the development session.

E is the type of event; 5 types of events are handled by the Log Exporter:

- **begin_process** involves specifying that a development session has been started;
- **begin_activity** involves specifying that a command execution (i.e., an 'activity') has been started;
- **end_activity** involves specifying that an activity has been completed;
- **atomic_activity** involves specifying that an activity whose time span is not significant has been executed;
- **end_process** involves specifying that a development session has been completed.

P is the identifier of the programmer the event refers to; each different programmer handled by the system must have a different identifier that will be used by the system to name the corresponding model.

C is the identifier of the development session (the 'case') the event refers to; each different case for a given programmer handled by the system must have a different identifier, and is composed by the identifier of the programmer with the initial timestamp session.

A is the name of a command relevant to model P.

N is a progressive integer distinguishing the commands in the same development session associated with the same main command; this is relevant in the case of many concurrent executions of the same command, to properly match their **begin_activity** and **end_activity** events.

The Log Exporter generates a unique log for each programmer containing all the development sessions executed to realize the program. For each programmer, the WoMan logs are collected in the WoMan log repository.

4.3. WoMan Miner Tool

Finally, the logs converted into the WoMan format are provided as input to the WoMan tool. For the aim of this work, we used only the Woman's WorkFlow Discovery subsystem. This subsystem allows obtaining the declarative process model describing the coding process executed by every single programmer, trace by trace. These process models are then used to evaluate each student's coding behavior.

5. Case Study and Lesson Learned

In this section, we show an application of the proposed approach and discuss the obtained results. The context description, the lesson learned, and the obtained results are reported in the following.

5.1. Evaluation Setting

The preliminary evaluation of our approach was performed on six students from a programming course held at the University of Bari Aldo Moro (Italy). The students selected were only those enrolled as a second-year students and had no previous knowledge of the

Java programming language. In particular, we used the proposed approach to investigate the following aspects: (i) Coding behavior, (ii) programmers' productivity, and (iii) the more frequent misconceptions and mistakes. Finally, we used the proposed approach to build the student's learning curve to better understand the novice programmers' learning process.

The project work assigned to the students consists of implementing a simple software system by using the Java programming language. Each student received a personal copy of the Eclipse IDE for Java Developers (version: 2019-09 R 4.13.0) equipped with the Log Processor plugin, and personal access to both the software development platform and the GitHub platform. Each student created a personal private repository on GitHub where they can upload and commit any file and change of the project. Each student had three months to complete their project and those who did not respect this deadline were discarded in the case study analysis.

After project completion, whereby the teachers were officially notified, the students uploaded the final version of the application, including the metadata collected by the Log Processor, and access to the private repository was granted to the teaching staff. Then, the HCIs and students' commits were processed by the Log Exporter and transformed into logs. Following the cleaning step, a total of 427 logs were collected for all the students involved in the case study. Subsequently, the students' logs were mined using the WoMan miner to discover their underlying process models. Finally, each student project was evaluated by the teacher at the final exam. Experiments were run on a Lenovo Thinkpad T580 laptop computer at the Department of Computer Science of the University of Bari, endowed with an Intel Core i7-8550U @ 1.80 GHz 64-bit processor with 16 GB of RAM, running YAP Prolog 6.2.2 under Kubuntu Linux 18.04.

5.2. Discussion about the Coding Behavior

Table 1 reports statistics about the students' behavior and the learned model. Each row corresponds to one user, identified by a character code. Each user was associated to a logfile, reporting the trace of all sessions on the Eclipse environment to fulfill their assignment, where each session is considered as a case of process execution. Column #*cases* reports the number of cases/sessions, and column #*activities* reports the overall number of activities (i.e., instances of task execution) in those cases. Column #*activ./case* reports the average number of activities per case. Then, the *runtime* of the system for processing all cases is reported, followed by the average runtime to process every single activity in milliseconds (column *ms/activ.*). The last columns report the number of different tasks carried out by the user (column #*tasks*) and the number of transitions (i.e., partial task combinations) identified by the system (column #*transit.*), indicating the complexity of the model.

Table 2 reports, instead, some metrics related to programmers' productivity, which is useful in granting another perspective to the models discovered by WoMan. Each row corresponds to one user, who is always identified by a character code. The column #*exam score* reports the exam scores obtained by the user classified by three ranges: Low, medium, and high. Low is when the score of the final exam is less the 23. Medium is when the exam score is between 23 and 27. Finally, high is when the exam score is greater than 27. Universities in Italy use a 30-point scale.

The total number of insertions and deletions are respectively reported by columns #*insertions* and #*deletions*. Finally, column #*insertions-deletions* reports the effective number of rows written by the user obtained as the difference between the total number of insertions and the total number of deletions.

Observing Tables 1 and 2, it is apparent that the measures for user A, concerning both the activities executed per case and the actual number of rows written, differ significantly from those of the other users. In fact, while for all other students the average number of activities per case ranges between 537 and 620, for A it is just 291. Moreover, user A wrote 1631 lines only, unlike other students who wrote between 3239 and 3986 lines. This seems

to suggest that the average number of activities per case could be considered as a further warning about the programmer's productivity.

Table 1. Statistics on the training data and learning outcomes.

User	*#cases*	*#activities*	*#activ./case*	*runtime*	*ms/activ.*	*#tasks*	*#transit.*
A	14	4078	291.29	51 s	13	41	206
B	207	128,389	620.24	2 h 16 min 25 s	64	71	580
C	113	60,725	537.39	44 min	43	85	521
D	44	30,413	691.20	47 min 5 s	93	33	254
E	32	20,881	652.53	24 min 58 s	72	68	466
F	17	10,977	645.71	4 min 39 s	25	49	299
Average	71.17	42,577.17	573.06	43 min	52	57.83	387.67
Total	427	255,463	-	4 h 17 min 58 s	-	-	-
Overall	427	255,463	598.27	4 h 21 min 41 s	61	132	1078

Table 2. Statistics on students' productivity and their exam scores.

User	*#exam score*	*#insertions*	*#deletions*	*#insertions-deletions*
A	low	1761	130	1631
B	high	8014	4340	3674
C	high	4536	1131	3405
D	medium	4089	850	3239
E	high	5434	1448	3986
F	medium	4380	407	3973

Tukey's method was used to check whether user A was actually an outlier. This method states that the outliers are values more than 1.5 times the Interquartile Range (IQR) from the quartiles, i.e., they are either below $Q1 - 1.5 \cdot IQR$ (lower bound) or above $Q3 + 1.5 \cdot IQR$ (upper bound), where Q1 and Q3 represent, respectively, the first and the third Quartile. Table 3 reports the lower (column *lower*) and the upper (column *upper*) bounds of Tukey's method, along with all the values necessary to calculate them. Comparing the lower and upper bounds reported in Table 3 to the values reported in Tables 1 and 2, we see that only the values of user A are lower than the lower bound, for both the measures. This proves that user A represents an outlier.

Table 3. Tukey's method to detect outliers.

Measure	*Min*	*Q1*	*Average*	*Q3*	*Max*	*IQR*	*Lower*	*Upper*
#activ./case	291.29	558.10	573.06	650.82	691.2	92.72	419.018	789.90
#insertions-deletions	1631	3280.5	3318	3898.25	3986	617.75	2353.87	4824.87

The last three rows in Table 1 report, respectively, the average and total values for the six user models, and the statistics for an additional experiment in which we learned a single model from the data of all students (*Overall*). We note that the overall time is almost the same as the sum of the times for the single models, meaning that there is no significant overhead in adding more data to the training set. In fact, the average runtime per activity in the overall model (microaverage) is very close to the average runtime of the single student (macroaverage). On the other hand, the number of tasks and transitions in the overall model is quite higher than the average number of tasks and transitions for the single models, suggesting that each user has a portion on quite 'personal' behavior. Still, these numbers are much less than the sum of the number of tasks and transitions in the single models, suggesting that a relevant portion of common behavior is present.

The main findings are: (i) Students execute the system only in 70% of development sessions. This suggests that they have not always checked for the changes made in the

source code and (ii) students toggle breakpoint in the source code only in 1% of development sessions. This suggests that students, even if they have the knowledge and the meaning of debugging tasks, are not prone to debug the source code to find and fix bugs and (iii) students did not commit their changes on the repository in 20% of development sessions. This suggests that, for the development sessions without any commit made, they had some significant troubles and, for this reason, feedback or a dialogue with the teacher or manager is necessary before continuing their coding task.

5.3. Discussion about Student's Misconception

Looking at Table 1 we can add further considerations about students' misconceptions. For two students, B and C, the number of development sessions is larger than the number of days available to complete the case study. A possible explanation is that both students misunderstood the concept of a development session with that of a commit. Each time they made a commit to the repository, they closed the current development session. This is an error from the standpoint of standard development procedures. Thus, our system was able to spot an undesired user behavior that other systems could not identify. This means that, if applied online, it would be able to immediately notify the wrong behavior to the user, which may help them in refining their development habits. This confirms that WoMan's input formalism and processing functions are particularly suited for application to usability support. In addition, it is worth mentioning that for user E, it was somehow harder for processing. Whilst consisting of just 32 cases, it took six times the runtime of user F, having half the cases, and whilst their average number of activities per case are almost the same, the model learned that the former involves many more tasks and transitions.

5.4. Programmer Learning Curve

We learned a model for each user so as to compare the behavior of different students on the same task. In the following, we will discuss the models and behavior of all users, including A, despite it being an outlier. Indeed, since our perspective is checking whether our approach may help in understanding the coding behavior of students, we would like to understand the behavior of any student. Moreover, studying the behavior of outliers may be particularly interesting, since they might correspond to very good or seriously wrong behaviors. In the former case, we might learn lessons to distribute to other students in the form of suggestions and indications and in the latter case, we might identify errors and be able to propose to student ways for improving their behavior.

Since WoMan is an inherently incremental system, learning naturally adopts a prequential approach as each case is processed in turn and immediately contributes to the adjustment/refinement/improvement of the model. Ths, the next case already takes advantage of an improved model. This brings the expectation that, as long as more and more cases are processed, the most stable part of the user's behavior has been learned, and thus fewer refinements are applied to the model. Still, we should consider that in our formalization, the cases correspond to sessions, and thus different sessions may concern different phases of the project development, possibly associated with different actions. Therefore, we cannot expect full convergence of the model, and new combinations of activities might be found until the last case.

Figure 2 shows the learning trend while discovering the models for the different students. Recall that the log for each user reports all the working sessions they executed while developing a project and that learning happened incrementally using a prequential approach (i.e., each case in the log was compared to the model learned so far from all the previous cases in the log, and used to refine it before moving to the next case). The x axis in all graphics counts the number of cases in the log of a user, while the y axis counts the number of new model components discovered in each case: The blue represents the tasks, while red shows the transitions. The peaks denote new behavior; the higher they are, the more different the behavior that is introduced in the corresponding case. Ideally, the plots should tend to zero when moving toward the right-hand side, since fewer new components

would denote that the model has already been captured and fewer new behaviors are identified over time. Still, we should keep in mind that the different sessions cover different phases of the development lifecycle, and thus we may expect different behaviors to emerge in time just because of the different tasks to be carried out in the different phases.

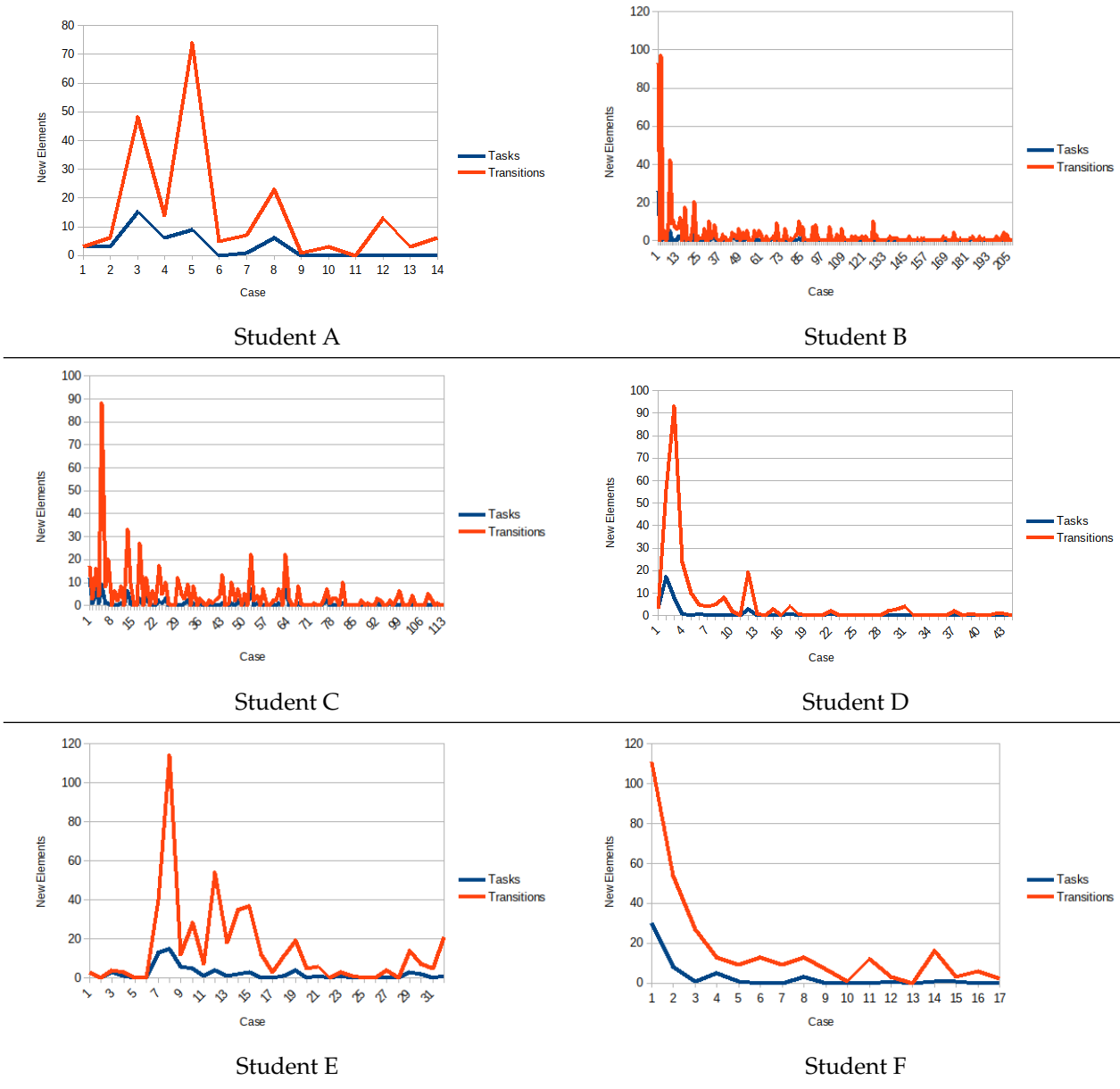

Figure 2. Trend in incremental prequential process model learning for different students.

The plots for transitions and tasks for each user have a similar shape, only for tasks with peaks that are much lower, as expected (since the possible combinations of tasks are much more than the number of tasks). The expected convergence takes place in all graphics for tasks. This makes sense because tasks correspond to the operations allowed by the interface, and most of them are used since the very early phases of the projects. In addition, for transitions, a trend to convergence is evident, albeit for two students (A and E) and we note a raise in the peaks just towards the end of the plot, showing that they carried out different combinations of activities in the last phases of the project. All plots have one prominent peak, which is much higher than the other ones and of a height around 100. Apart from this, we can distinguish 3 kinds of plots: Plots where most of the behavior is learned in the very initial cases, and then little novelty happens subsequently (B and

F), plots where most of the behavior is learned very early but not immediately (C and D), and plots where most of the behavior is learned after an initial offset, in which very little happens (A and E). Note that the last group corresponds to the students with fewer standard statistics in Table 1, and also to the plots with fewer convergence in Figure 2. In general, most of the behavior is learned in the first 10 cases (with the exception of user E): The transitions learned in the initial cases probably represent the most standard and frequent operations that are required throughout the coding task. Then, the peaks become lower and more sparse, denoting slight deviations from the models and new portions of behavior. Note that user E was also peculiar for the statistics in Table 1.

Figure 3 shows the learning trend when learning a single overall model from all sessions of all students, in the following order: B–A–C–D–E–F. The yellow line shows the extent of data for each user, and the user change point in the cases on the x axis. In particular, the first user, determining the initial model to be refined by the others, was user B. Data of user B were provided first since it was associated with the largest number of cases and to the most complex single user model, and thus we wanted to check whether this model could also include the behavior of all the others. In such a case, after the first user, we would obtain an already 'stable' model, meaning that it exhibited a superset of behaviors of all the others. The plot shows that this is not the case, as each user introduces many new peaks, albeit lower (about 10–20 new transitions), meaning that there is still a portion of 'personal' behavior for each of them. Only user E has few low peaks, which is very different from the plot for its own model, where higher peaks, and specifically a very prominent one, are present.

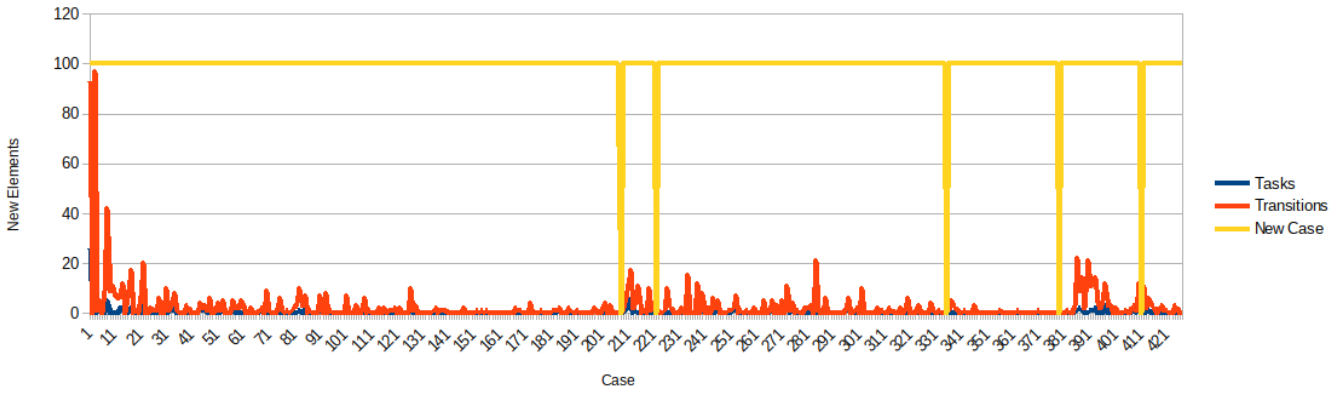

Figure 3. Trend in incremental prequential process model learning for all students.

6. Threats to Validity

In this section, we discuss the main threats to the validity of our study, focusing our attention on the threats to external, internal, and construct validity:

6.1. External Validity

Global generalization is not possible for several reasons. Firstly, it it not possible due to the small number of experimental subjects involved in the experimentation, which was a total of six students. Moreover, as a programming language, we considered Java and therefore results could be considered as useful only in basic Java programming courses in an academic context. All the assignments were created and revised by the teaching staff to mitigate the possible lack of quality and balance. We are aware that different assignments might have different complexity and, for this reason, we paid attention to creating assignments with a similar level of complexity. The generalization to a larger population of students and other different populations of students (e.g., cohorts with different levels of motivation, or different educational backgrounds, different programming languages, etc.), is worthy of future experimentation.

6.2. Internal Validity

A possible threat was that the subjects belonged to different academic years (a student can follow a course in a year following the year defined in the study plan). To avoid this threat, we selected only students enrolled to the second year. Another possible threat is that some subjects may have already taken a Java programming course in secondary school, implying that the results may have been skewed by these subjects. To avoid this situation, we selected only students with no knowledge of Java language. Moreover, subjects were not aware of the experimental hypotheses and were not rewarded for participation in the experiment.

6.3. Construct Validity

The measures used to provide values for students' productivity were based on the evaluation of exams, as has been done for several years, so we believe that the evaluation process can be considered suitable. All the exams were examined by a researcher with a 10-year experience in the field of Java programming language.

7. Conclusions

This paper presents an incremental process mining approach for the analysis of coding activities in real-time. The aim is to analyze novice programmer coding activities from data extracted by the IDEs and VCS and provide them, command by command, development session by development session, timely feedback. The described case study shows that the proposed approach provides some useful information on coding behavior. The numeric/statistical analysis of the process mining system's outcomes provides a first indication of the different behaviors of users, allowing us to identify users that are very different from others. The incrementality of the system provides also indications on when and how the new behaviors happen during the coding activities of the users. For example, we found that most of the new behavior appears in the initial set of coding sessions for all users, but that some of them go on to exhibit new behaviors throughout the coding process, possibly connected to the different actions that are required in different stages. By comparing the statistics and learning curves to the system's outputs (and specifically the learned models), we could also gain an understanding of the good and bad actions of each user, and map them to the type of user. For example, novice programmers are not prone to debug the source code to find and fix bugs, and do not usually check for the changes made in the source code. Moreover, timely feedback has to be made on the very first development sessions because, in general, most of the coding behavior is learned in the first 10 development sessions. These results can potentially be useful for the timeliness and contextuality of the feedback provided by the IDE.

In future, our first goal is to automatize feedback to students. We also have the intention to further analyze the transitions of models discovered, to apply process mining techniques to perform a quantitative analysis of the team working dynamics, and, also, to use conformance checking to identify deviations of students' activities from a reference model. Moreover, it could be useful to expand our study to programmers' activities performed outside the IDE, such as a forum, chat, so as to identify and categorize opinions expressed in a piece of text to determine whether the programmers' attitude towards a particular topic is positive, negative, or neutral [34,35]. Finally, a larger number of experimental subjects will be necessary to strengthen the validity of obtained results.

Author Contributions: Conceptualization, P.A., M.L.B., M.C., D.R. and S.F.; methodology, P.A., M.L.B., M.C., D.R. and S.F.; software, P.A., M.L.B., M.C., D.R. and S.F.; validation, P.A., M.L.B., M.C., D.R. and S.F.; formal analysis, P.A., M.L.B., M.C., D.R. and S.F.; investigation, P.A., M.L.B., M.C., D.R. and S.F.; resources, P.A., M.L.B., M.C., D.R. and S.F.; data curation, P.A., M.L.B., M.C., D.R. and S.F.; writing—original draft preparation, P.A., M.L.B., M.C., D.R. and S.F.; writing—review and editing, P.A., M.L.B., M.C., D.R. and S.F.; visualization, P.A., M.L.B., M.C., D.R. and S.F.; supervision, P.A., M.L.B., M.C., D.R. and S.F.; project administration, P.A., M.L.B., M.C., D.R. and S.F.; funding acquisition, P.A., M.L.B., M.C., D.R. and S.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Experimental data are available from the authors upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ardimento, P.; Bernardi, M.L.; Cimitile, M. Malware Phylogeny Analysis using Data-Aware Declarative Process Mining. In Proceedings of the 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS 2020), Bari, Italy, 27–29 May 2020; pp. 1–8. [CrossRef]
2. Leemans, M.; van der Aalst, W.M.P. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In Proceedings of the 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), Ottawa, ON, Canada, 30 September–2 October 2015; pp. 44–53. [CrossRef]
3. Liu, C.; van Dongen, B.; Assy, N.; van der Aalst, W.M.P. Component behavior discovery from software execution data. In Proceedings of the 2016 IEEE Symposium Series on Computational Intelligence (SSCI 2016), Athens, Greece, 6–9 December 2016; pp. 1–8. [CrossRef]
4. van der Aalst, W. Big software on the run: In vivo software analytics based on process mining (keynote). In Proceedings of the ICSSP 2015: International Conference on Software and Systems Process 2015, Tallinn, Estonia, 24–26 August 2015; pp. 1–5.
5. Hundhausen, C.D.; Olivares, D.M.; Carter, A.S. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *ACM Trans. Comput. Educ.* **2017**, *17*, 1–26. [CrossRef]
6. Ardimento, P.; Bernardi, M.L.; Cimitile, M.; Maggi, F.M. Evaluating coding behavior in software development processes: A process mining approach. In Proceedings of the 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP), Montreal, QC, Canada, 25–26 May 2019; pp. 84–93. [CrossRef]
7. Ferilli, S. Woman: Logic-based workflow learning and management. *IEEE Trans. Syst. Man Cybern. Syst.* **2013**, *44*, 744–756. [CrossRef]
8. Carolis, B.D.; Ferilli, S.; Redavid, D. Incremental Learning of Daily Routines as Workflows in a Smart Home Environment. *ACM Trans. Interact. Intell. Syst. (TiiS)* **2015**, *4*, 1–23. [CrossRef]
9. Ferilli, S.; Angelastro, S. Activity prediction in process mining using the WoMan framework. *J. Intell. Inf. Syst.* **2019**, *53*, 93–112. [CrossRef]
10. Ardimento, P.; Bernardi, M.L.; Cimitile, M.; De Ruvo, G. Mining Developer's Behavior from Web-Based IDE Logs. In Proceedings of the 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Naples, Italy, 12–14 June 2019; pp. 277–282. [CrossRef]
11. Ardimento, P.; Bernardi, M.L.; Cimitile, M.; De Ruvo, G. Learning analytics to improve coding abilities: A fuzzy-based process mining approach. In Proceedings of the 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), New Orleans, LA, USA, 23–26 June 2019; pp. 1–7. [CrossRef]
12. Mylyn. Available online: https://www.eclipse.org/mylyn/ (accessed on 19 November 2021).
13. Murphy, G.C.; Kersten, M.; Findlater, L. How are Java software developers using the Elipse IDE? *IEEE Softw.* **2006**, *23*, 76–83. [CrossRef]
14. Soh, Z.; Khomh, F.; Guéhéneuc, Y.; Antoniol, G. Towards understanding how developers spend their effort during maintenance activities. In Proceedings of the 2013 20th Working Conference on Reverse Engineering WCRE, Koblenz, Germany, 14–17 October 2013; pp. 152–161. [CrossRef]
15. Bavota, G.; Canfora, G.; Penta, M.D.; Oliveto, R.; Panichella, S. An Empirical Investigation on Documentation Usage Patterns in Maintenance Tasks. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 210–219. [CrossRef]
16. Ying, A.T.T.; Robillard, M.P. The Influence of the Task on Programmer Behaviour. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, Canada, 22–24 June 2011; pp. 31–40. [CrossRef]
17. Murphy-Hill, E.; Parnin, C.; Black, A.P. How We Refactor, and How We Know It. *IEEE Trans. Softw. Eng.* **2012**, *38*, 5–18. [CrossRef]
18. Astromskis, S.; Bavota, G.; Janes, A.; Russo, B.; Di Penta, M. Patterns of developers behaviour: A 1000-h industrial study. *J. Syst. Softw.* **2017**, *132*, 85–97. [CrossRef]

19. IEEE Task Force on Process Mining. Van Der Aalst, W.; Adriansyah, A.; De Medeiros, A.K.A.; Arcieri, F.; Baier, T.; Blickle, T.; Bose, J.C.; Van Den Brand, P.; Brandtjen, R.; Buijs, J.; et al. Process Mining Manifesto. In *Business Process Management Workshops: Proceedings of the BPM 2011 International Workshops, Clermont-Ferrand, France, 29 August 2011*; Springer: London, UK, 2011; pp. 169–194.
20. van der Aalst, W. Process Mining: Overview and Opportunities. *ACM Trans. Manag. Inf. Syst.* **2012**, *3*, 7.1–7.17. [CrossRef]
21. van der Aalst, W. The Application of Petri Nets to Workflow Management. *J. Circuits Syst. Comput.* **1998**, *8*, 21–66. [CrossRef]
22. Pesic, M.; van der Aalst, W.M.P. A declarative approach for flexible business processes management. In *Business Process Management Workshops*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4103, pp. 169–180.
23. Weijters, A.; van der Aalst, W.M.P. Rediscovering Workflow Models from Event-Based Data. In Proceedings of the 11th Dutch-Belgian Conference on Machine Learning (Benelearn 2001), Antwerp, Belgium, 21 December 2001; pp. 93–100.
24. van der Aalst, W.; Weijters, T.; Maruster, L. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Trans. Knowl. Data Eng.* **2004**, *16*, 1128–1142. [CrossRef]
25. de Medeiros, A.K.A.; van Dongen, B.F.; van der Aalst, W.M.P.; Weijters, A.J.M.M. *Process Mining: Extending the α-Algorithm to Mine Short Loops*; Technical Report, BETA Working Paper Series; Eindhoven University of Technology: Eindhoven, The Netherlands, 2004.
26. Wen, L.; Wang, J.; Sun, J. Detecting Implicit Dependencies Between Tasks from Event Logs. In *Frontiers of WWW Research and Development—APWeb 2006*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3841, pp. 591–603.
27. Van der Aalst, W.M.P.; De Medeiros, A.K.A.; Weijters, A.J.M.M. Genetic process mining. In *Applications and Theory of Petri Nets 2005*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3536, pp. 48–69.
28. de Medeiros, A.K.A.; Weijters, A.J.M.M.; van der Aalst, W.M.P. Genetic process mining: An experimental evaluation. *Data Min. Knowl. Discov.* **2007**, *14*, 245–304. [CrossRef]
29. Chesani, F.; Lamma, E.; Mello, P.; Montali, M.; Riguzzi, F.; Storari, S. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. In *Transactions on Petri Nets and Other Models of Concurrency II*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 278–295.
30. Maggi, F.M. Declarative Process Mining. In *Encyclopedia of Big Data Technologies*; Sakr, S., Zomaya, A.Y., Eds.; Springer: Cham, Switzerland, 2019; pp. 625–632. [CrossRef]
31. Maggi, F.M.; Bose, R.P.J.C.; van der Aalst, W.M.P. Efficient Discovery of Understandable Declarative Process Models from Event Logs. In *Advanced Information Systems Engineering*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7328, pp. 270–285.
32. Cattafi, M.; Lamma, E.; Riguzzi, F.; Storari, S. Incremental Declarative Process Mining. In *Smart Information and Knowledge Management*; Studies in Computational Intelligence; Springer: Berlin/Heidelberg, Germany, 2010; Volume 260, pp. 103–127.
33. Yoon, Y.; Myers, B.A. Capturing and Analyzing Low-Level Events from the Code Editor. In Proceedings of the PLATEAU '11: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, Portland, OR, USA, 24 October 2011; pp. 25–30. [CrossRef]
34. Mehta, P.; Pandya, S. A review on sentiment analysis methodologies, practices and applications. *Int. J. Sci. Technol. Res.* **2020**, *9*, 601–609.
35. Pandya, S.; Shah, J.; Joshi, N.; Ghayvat, H.; Mukhopadhyay, S.C.; Yap, M.H. A novel hybrid based recommendation system based on clustering and association mining. In Proceedings of the 2016 10th International Conference on Sensing Technology (ICST), Nanjing, China, 11–13 November 2016; pp. 1–6.