Mining Developers' Workflows from IDE Usage

Constantina Ioannou, Andrea Burattin, and Barbara Weber

Technical University of Denmark,
Kgs. Lyngby, Denmark

Abstract. An increased understanding of how developers' approach the development of software and what individual challenges they face, has a substantial potential to better support the process of programming. In this paper, we adapt Rabbit Eclipse, an existing Eclipse plugin, to generate event logs from IDE usage enabling process mining of developers' workflows. Moreover, we describe the results of an exploratory study in which the event logs of 6 developers using Eclipse together with Rabbit Eclipse were analyzed using process mining. Our results demonstrate the potential of process mining to better understand how developers' approach a given programming task.

Keywords: process mining, tracking IDE interactions, developers' workflows, source code

1 Introduction

Increasing the productivity of software development has traditionally been an important concern of the software engineering field. This includes software development processes (e.g., agile and lean development), development principles and practices (e.g., test-driven development, continuous integration), tools like integrated development environments (IDEs), but also human factors. Considering the tremendous productivity differences between developers of 10:1 [4], there is substantial potential to better support the process of programming by better understanding how developers' approach the development of software and what individual challenges they face.

The process of programming is highly iterative, interleaved and loosely ordered [7]. Developers need to understand the requirements presented to them and form an internal representation of the problem in working memory by extracting information from external sources [3]. Based on the requirements a solution design is developed [3]. This includes at the general level the decomposition of requirements into system structures, i.e., modules and, on a more detailed level, the selection or development of algorithms to implement different modules [18]. The solution design is then implemented using a specific development environment and a particular programming language [18, 21] and it is evaluated whether the developed solution is suitable to solve a problem [9, 21, 6]. Depending on the development process used, the development principles and practices, the used IDE and programming language as well as personal preferences, experience, and capabilities the process of programming varies.

In this paper we show the potential of process mining to better understand how developers' approach the creation of a solution for a given programming task using the IDE Eclipse. The contribution of this paper is twofold. First, the paper provides adaptations of an existing Eclipse plugin, i.e., Rabbit Eclipse, to produce event logs that can be used for process mining purposes. Second, it describes the results of an exploratory study in which the event logs of 6 development sessions were analyzed. The work does not only have potential to better understand the processes developers follow to create a solution to a given programming task using the IDE Eclipse.

In the future we can use the developed plugin to compare how the usage of different development principles and practices impacts the way how developers solve a given problem and use conformance checking to identify deviations from best practices. Moreover, when integrated with eye tracking, we cannot only determine how developers interact with the IDE and the different source code artifacts, but additionally where they have their focus of attention.

2 Background and Related Work

In Section 2.1 we discuss existing research on tracking IDE usage. In this paper we use the IDE Eclipse together with the Rabbit Eclipse plugin to collect the interactions of developers with the IDE (cf. Sect. 2.2) that are then used for process mining (cf. Sect. 2.3).

2.1 Tracking IDE Usage

Research recording the interactions of a developer with the IDE including their analysis and visualization are related to our work. For example, [17] provides quantitative insights into how Java developers use the Eclipse IDE. Moreover, [16] developed DFlow for recording developers' interactions within the IDE Pharao including their visualization and applied it to better understand how developers spend their time. Similarly, Fluorite [23] and Spyware [19] were implemented in order to collect usage data from the Eclipse IDE and to replay and backtrack developers strategies visualizing code histories. Unlike our research the focus is on a single development session, rather than abstract behavior derived from a set of sessions. Most closely related to our work is [1] in which frequent IDE usage patterns have been mined and filtered in order to form usage smells. More precisely, this approach identifies time-ordered sequences of developer actions that are exhibited by many developers in the field. However, the focus of [1] is on developers' interactions with the IDE only. In contrast, our proposal considers the inter-relationships of interactions and source code artifacts, thus, emphasizing the way developers solve a programming task rather than how developers use the IDE.

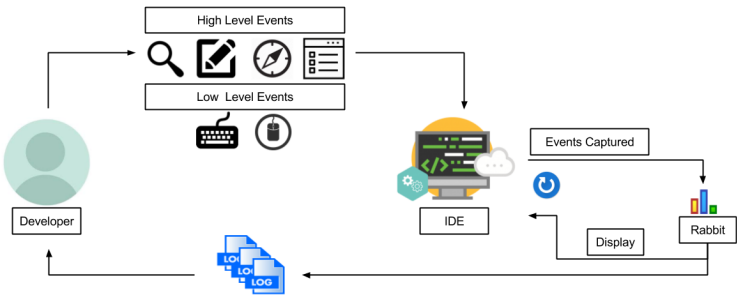

Fig. 1. Model: Rabbit Eclipse

2.2 Rabbit Eclipse

Rabbit Eclipse is a statistical plugin, capable of recording developers' interaction without interrupting their process, within the Eclipse IDE. Fig. 1 gives an overview of the instrumentation approach employed in this paper.

When implementing a software within Eclipse, developers generate through their interactions with the IDE various low and high level events. Low level events are keyboard shortcuts and mouse clicks, whereas high level events are related to context menu or wizard interactions. Whenever an event is observed, Rabbit Eclipse is triggered to capture and analyze the interaction. These interactions are then stored in event logs and upon request of a developer the data collected by Rabbit Eclipse can be displayed graphically within Eclipse.

The structure of event entries are presented in Fig. 2. For each type of event entry an event log is produced. At the top of the hierarchy are the classes `DiscreteEvent` and `ContinuousEvent`, which distinguish between the main types of interactions recorded, i,e. instant and continuous interactions. `Command` and `Breakpoint` events are listed as discrete interactions. On the other hand, interactions such as switching between files, views, perspectives, launching, Java elements and session inherit from `ContinuousEvent`, since the duration for such activities is relevant.

2.3 Process Mining

Process mining is the bridge between model-based process analysis and data oriented analysis techniques such as machine learning and data mining [22]. In order to be amenable for process mining the event logs produced should conform to the minimum data model requirements [8]. These are: the case id, which determines the scope of the process, an activity which determines the level of detail for the steps and timestamp, which determines when the activity took place. Using process discovery a process model explaining the behavior of the recorded log can be derived. Moreover, using conformance checking deviations of the event log when compared to a reference behavior can be identified.

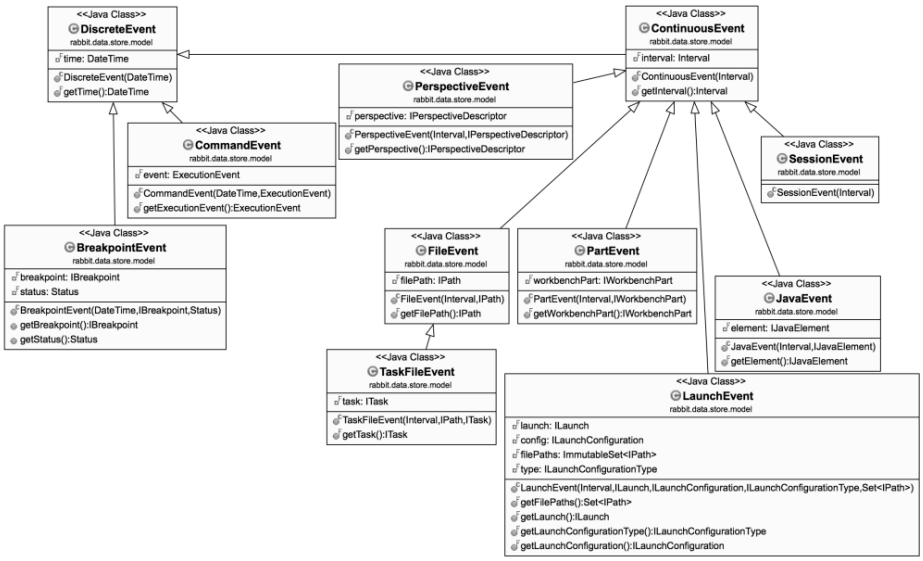

Fig. 2. UML Class diagram of the hierarchy of events as recorded by Rabbit Eclipse

Closely related to our work presented in this paper is the emerging area of software process mining. With the increasing availability of software execution data, the application of process mining techniques to analyze software execution data is becoming increasingly popular. The potential of software process mining was first demonstrated by [10, 20]. More specifically, source code repositories were mined to obtain insights into the software development processes development teams employed. Moreover, [2] suggests the usage of localized event logs where events refer to system parts to improve the quality of the discovered models. In addition, [11] proposes the discovery of flat behavioral models of software systems using the Inductive Miner [13]. In turn, [15] proposes an approach to discover a hierarchical process model for each component. An approach for discovering the software architectural model from execution data is described in [14]. Finally, [12] allows to reconstruct the most relevant statecharts and sequence diagram from an instrumented working system. The focus of all these works is software development processes or the understanding of the behavior of the software, while our focus is to use process mining to understand how a developer solves a programming task.

3 Extending Rabbit Eclipse for Process Mining

Although Rabbit Eclipse provides broad and profound statistical results on developers' interactions within the Eclipse IDE, the data provided are not sufficient to enable the mining of developers' interactions as envisioned. In particular, as previously highlighted, timestamp and case id notions were not included in the

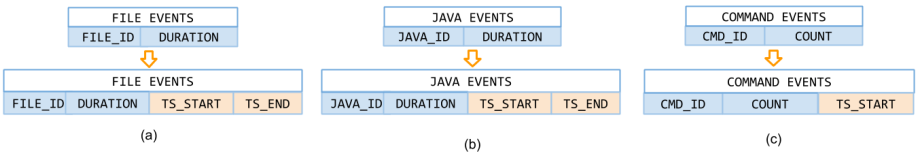

Fig. 3. Timestamp modifications on three events

collection. Therefore, we needed to expand some events in order to enable their usage in the context of process mining. Firstly, all the event logs were customized to include timestamp and case id by modifying classes `DiscreteEvent` and `ContinuousEvent`. Further, due to the nature of Rabbit Eclipse the collected interactions have no actual relation between them. To resolve this constraint focus was given to change the interpretation of `FileEvent`, `JavaEvent`, and `CommandEvent` (cf. Fig. 2) which seemed the most promising with respect to our goal. The rest of this section presents the adjustments introduced to enable the process mining investigations envisioned.

3.1 Adaptations to Enable Process Mining.

To enable the extraction of a workflow from our data, using process mining techniques, timestamps and case id needed to be included in the recordings.

As mentioned, the event logs for `ContinuousEvents` (both `FileEvent` and `JavaEvent`) contain durations, which means that entries referring to the same interaction were merged, whereas, concerning `DiscreteEvents` (`i.e., CommandEvents`), event logs report the frequency of such interactions. We instrumented Rabbit Eclipse to allow the collection of all timestamps, as shown in Fig. 3. Specifically, for `ContinuousEvents` timestamps to indicate start and end time were introduces (i.e., each event represented as time interval) and for `DiscreteEvents` a single timestamp was added (i.e., each event as time instant).

Additionally, Rabbit Eclipse does not have the notion of a case id. Therefore, we decided to artificially add one. Specifically, we assumed to have each developer referring to one different case id as approximation of a single development session.

With these changes in place, we were able to associate Rabbit Eclipse recordings to user's actions. Thus, we obtained a proper event log: we shifted the scope of Rabbit Eclipse from logging file changes (suitable for software process mining) to logging user's interaction with the IDE (suitable for process mining).

3.2 Mapping Commands to Resources.

By default, Rabbit Eclipse has no capability to establish links between the commands and the resource where these commands were performed. Therefore, we instrumented Rabbit Eclipse to be able to take this element into account together with timing information. The result is an augmented version of the `CommandEvent`, as depicted in Fig. 4. This augmentation needed to consider

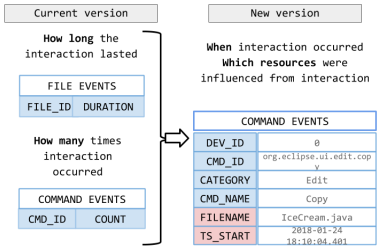

Fig. 4. Command modifications to track interactions

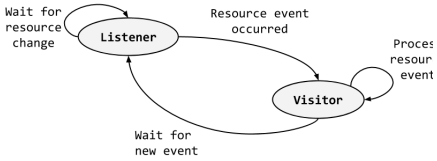

Fig. 5. Listener/visitor design pattern for the implemented tracker

different scenarios which are possible, including commands involving single or group of resources (such as renaming a folder or moving files). To better achieve our goal, we also implemented an interaction tracker, which listens for resource change events (see Fig. 5). Once the new tracker is triggered, it processes the event to identify and store affected commands.

3.3 Augmentation of Commands with Java Details.

The version of the `JavaEvent` class available in the current version of Rabbit Eclipse was not able to provide information referring to the specific Java constructs being modified. This data, however, could contain very important information for our analysis. To extract this information, we instrumented Rabbit Eclipse to inspect the Abstract Syntax Tree (AST) of each modified Java class. This enables the Rabbit Eclipse plugin to capture the modified methods and correspondingly update the commands.

4 Exploratory Study

This section explains the exploratory study we conducted to evaluate the adapted version of the Rabbit Eclipse plugin.

4.1 Study Design and Execution

Participants. Six participants were included in the case study. One participant is a software developer in a medium sized company, while the other five are newly graduated master students from Computer Science and related fields. All of them primarily develop in C, C++ and Java, mainly for embedded systems. Their age ranges from 25 to 29, and they have between 6 months to 2 years of experience using Eclipse.

Task. Participants had to work on a fairly simple programming task that takes around 30min/1hour for completion. The task required the participants to first

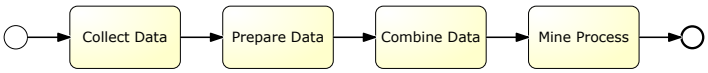

Fig. 6. Data collection and analysis procedure followed

install our version of Rabbit Eclipse and then to implement an inheritance hierarchy of classes that derive from an abstract superclass using Java as a programming language. As depicted in Fig. 14, the task consists of five classes, a superclass called `DessertItem` and four derived classes, i.e., `Cookie`, `Candy`, `IceCream` and `Sundae`. Participants were provided with a description of the classes to be implemented including the class diagram (cf. Appendix. A) and a test file called `DessertShop`. To avoid inconsistent naming between participants, participants were encouraged to strictly follow the class and method naming as shown in the class diagram. While working on the task participants were not allowed to ask for help or explanation since this could affect their way of thinking. After the participants finished their implementation, they were requested to send the files collected from the tool. Thereafter, the required data set for process mining was retrieved.

4.2 Data Collection and Analysis Procedure

Fig. 6 illustrates the data collection and analysis procedure we employed.

Step 1: Collect Data. To collect data concerning a developer's interactions with the IDE we asked participants to install our version of Rabbit Eclipse. During the implementation of the task all interactions of the participants with the IDE were then recorded using Rabbit Eclipse.

Step 2: Prepare Data. Throughout the second step of the approach and after receiving the exported raw data sets from all participants, the data set requires refinement before it can be used for process mining. To begin with, any unrelated, captured interactions with projects in the developer's current workspace were removed from the data set. Next, since process mining requires homogeneity among the data set, any inconsistencies in the data sets were detected and adjusted. An example of inconsistency is when a participant, instead of using correctly the requested naming `Test.java`, used `test.java`. In addition, the XML formatted event logs are converted to CSV format.

Step 3: Combine Data. The refined data are combined into one file and are imported to Disco.

Step 4: Mine Process. Disco was then used to analyze the data and settings were configured so that for all experiments the case id used was the developer's id and the timestamp used was the start time of events. Further, specific settings defined for each experiment are displayed in Table 1.

All participants were able to fulfill the task within the given time frame (i.e., all programming sessions lasted between 30 minutes and 1 hour). Most of the

Table 1. Settings used for Disco to obtain the four results

#	Result's Name	Participants	Event Log	Activity	Attribute
1	Most Common Commands	5	cmds	command	-
2	Developers' Workflow	5	cmds	file	command
3	Classes Workflow	5	cmds	command	file
4	Source Code Workflow	4	javas	method and file	-

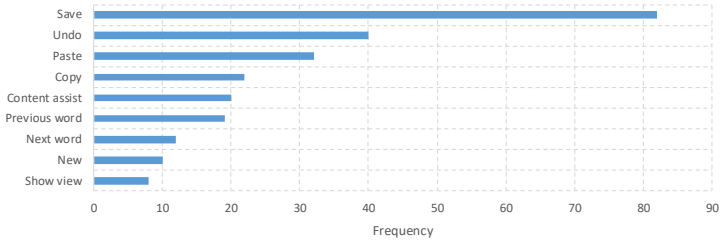

Fig. 7. Most common commands used

participants used expected methodologies and commands, however, as indicated in Tab. 1 for all analyses one of the participants had to be excluded because of generating a considerable amount of noise events, as well as for the fourth analysis two participants were excluded due to failed recordings from the tool.

4.3 Results

Most Common Commands. A statistical analysis of the entire sample using Disco showed that in total 323 commands were executed for all five participants (most common are shown in Fig. 7). When we observe the distribution of command interactions retrieved, we can see that only a small amount of the available Eclipse standard commands were executed. In fact, only 31 different commands occurred out of the 350 available. A possible explanation for that might stem from the simple nature of the given task. Moreover, our analysis showed that participants tend to use and repeat similar commands. Out of 31 different commands used, the most common are: Save, Undo and Paste which concurs with results of previous studies [17].

Developers' Workflow. Fig. 8 displays the connection between file switching and command interactions. The file resources implemented throughout the task are indicated as nodes and the command interactions leading from one file to another as edges. The diagram shows that half of participants begun their implementation by interacting with `DessertItem.java` which was the superclass and then moved to the implementation of subclasses, while the other half, begun interacting with subclasses (i,e. `Candy.java`, `Sundae.java`) and then moved to the superclass. These two approaches participants followed (cf. Fig, 9) are denoted in literature [5] as "top-down" versus "bottom-up" approach. When following a

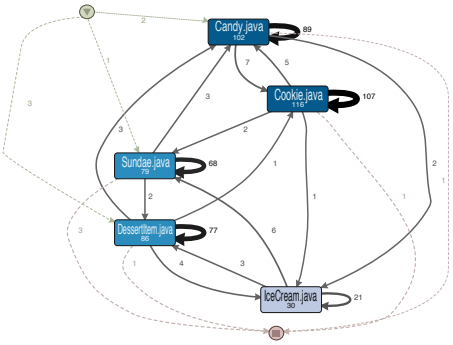

Fig. 8. Workflow Diagram depicted by Disco

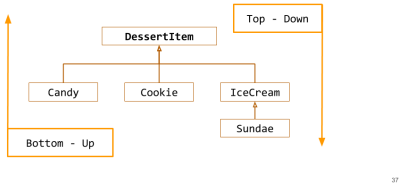

Fig. 9. Class outline with corresponding programming techniques

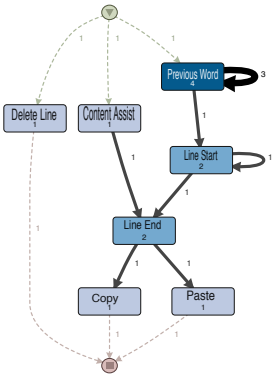

Fig. 10. Workflow Diagram of Ice Cream.java

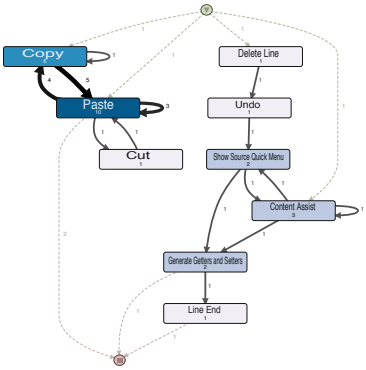

Fig. 11. Workflow Diagram of Cookie.java

top-down approach a developer begins with the main problem and then subdivides it into sub problems. Whereas when employing a bottom-up approach the developer begins with sub-problems building up to the main problem.

Classes Workflow. As observed in Fig. 8, class `Cookie.java` has the highest amount of interactions, whereas, `IceCream.java` has the least. To explore further this aspect we applied process mining focusing on these two classes separately. In Fig. 10 and Fig. 11 the generated workflow diagrams are shown. In this case, command interactions appear as nodes and the switching between them is represented as edges. From Fig. 10 we can infer the absence of high interaction traffic and this is expected since `IceCream.java` was fairly simple to implement. On the other hand, Fig. 11 illustrates `Cookie.java` which is more demanding: this is reflected in a more complex routing of the interactions, including self loops and repetition. This suggests that there is a diversity in the approach used when dealing with classes of different difficulty level.

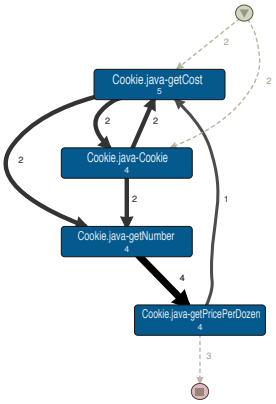

Fig. 12. Workflow Diagram Cookie.java

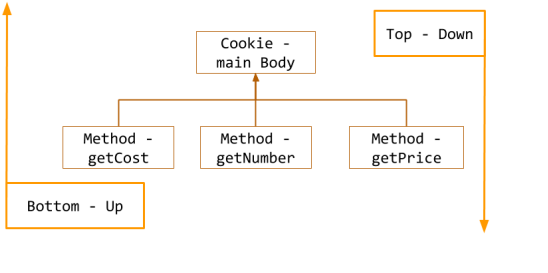

Fig. 13. Class outline with corresponding programming techniques

Source Code Workflow. In Fig. 12 an attempt to process mine the source code interactions in `Cookie.java` is displayed. The method names of the class are indicated as nodes and the arrows indicate the flow participants followed. Participants followed two patterns (cf. Fig, 13), either they begun by building the body of the class and then implementing the methods in detail or the opposite. This was observed not only within the `Cookie.java` but also in the other implemented classes. Therefore this realization, implies that the workflow techniques mentioned (top down and bottom up) are applied not only for the class design but also for the methods, showing potential in performing process mining on source code development.

5 Summary and Conclusions

In this paper we presented an extension of the Rabbit Eclipse plugin that is able to collect developers' interactions with Eclipse that can be used for process mining. Moreover, we presented the results of an exploratory study where 6 developers developed a small piece of software using Eclipse together with Rabbit Eclipse. The analysis of the data using process mining allowed us to identify the most commonly used commands. Moreover, we could observe that the participating developers employed different object oriented programming techniques, i.e., top down and bottom up, to solve the programming task. In addition, we could identify differences in creating single classes. Our results demonstrate that it is possible to mine developers' workflows from their interactions within an IDE. However, it has to be noted that the existing work is subject to several limitations such as the low number of subjects and the task difficulty.

In the future we plan to extend our study with more subjects and more complex tasks. Another avenue of future research is the integration with eye tracking, thus allowing us to complement our results with data on where developers fo-

cused their attention. In addition, future work will consider (retrospective think aloud) to obtain insights into the developer's thinking process.

References

1. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE Transactions on Software Engineering*, 43(4):359–371, 2017.
2. Wil van der Aalst. Big software on the run: In vivo software analytics based on process mining (keynote). In *Proceedings of ICSSP 2015*, pages 1–5. ACM, 2015.
3. R. Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6):737–751, 1977.
4. T. DeMarco and T. Lister. *Peopleware: Productive Projects and Teams.* Dorset House Publishing Co., New York, 1987.
5. Yves Pigneur Shusma Patel Dimitri Konstantas, Michel Leonard. *Object-Oriented Information Systems.* Springer, Geneva, Switzerland, 2003.
6. R Guindon, H Krasner, and B Curtis. Cognitive process in software design: activities in early, upstream design. In *Human–computer Interaction INTERACT'87*, pages 383–388. Elsevier Science Publishers B.V. (North Holland), 1987.
7. Raymonde Guindon and B. Curtis. Control of cognitive processes during software design: what tools are needed? In *Proc. CHI'88*, pages 263–268, 1988.
8. IEEE Task Force on Process Mining. Process Mining Manifesto. *Business Process Management Workshops*, pages 169–194, 2011.
9. Elaine Kant and Allen Newell. Problem Solving Techniques for the design of algorithms. *Information Processing & Management*, 20(1–2):97–118, 1984.
10. Ekkart Kindler, Vladimir Rubin, and Wilhelm Schäfer. Incremental workflow mining based on document versioning information. In *Software Process Workshop*, volume 3840 of *LNCS*, pages 287–301. Springer, May 2005.
11. M. Leemans and W. M. P. van der Aalst. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *Proceedings of MODELS*, pages 44–53, Sept 2015.
12. Maikel Leemans, Wil M. P. van der Aalst, and Mark G. J. van den Brand. Recursion aware modeling and discovery for hierarchical software event log analysis (extended). *CoRR*, abs/1710.09323, 2017.
13. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*. Springer, 2013.
14. Cong Liu, Boudewijn F. van Dongen, Nour Assy, and Wil M. P. Aalst. Software architectural model discovery from execution data. In *13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 03 2018.
15. Cong Liu, B. van Dongen, N. Assy, and W. M. P. van der Aalst. Component behavior discovery from software execution data. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Dec 2016.
16. Roberto Minelli and Michele Lanza. Visualizing the workflow of developers. *Proceedings of VISSOFT*, pages 2–5, 2013.
17. G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006.
18. Nancy Pennington, Y. Lee Adrienne, and Bob Rehder. Cognitive Activities and Levels of Abstraction in Procedural and Object–Oriented Design. *Human-Computer Interaction*, 10:171–226, 1995.

19. Romain Robbes and Michele Lanza. SpyWare: A Change-aware Development Toolset. *Proceedings of ICSE*, pages 847–850, 2008.
20. Vladimir Rubin, Christian W. Günther, Wil M. P. van der Aalst, Ekkart Kindler, Boudewijn F. van Dongen, and Wilhelm Schäfer. Process mining framework for software processes. In *Proceedings of ICSP 2007*, pages 169–181. Springer, 2007.
21. David P. Tegarden and Steven D. Sheetz. Cognitive activities in OO development. *Int. J. Human-Computer Studies*, 54(6):779–798, 2001.
22. W.M.P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.
23. YoungSeok Yoon and Brad a. Myers. Capturing and analyzing low-level events from the code editor. *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools - PLATEAU '11*, page 25, 2011.

A Task Description

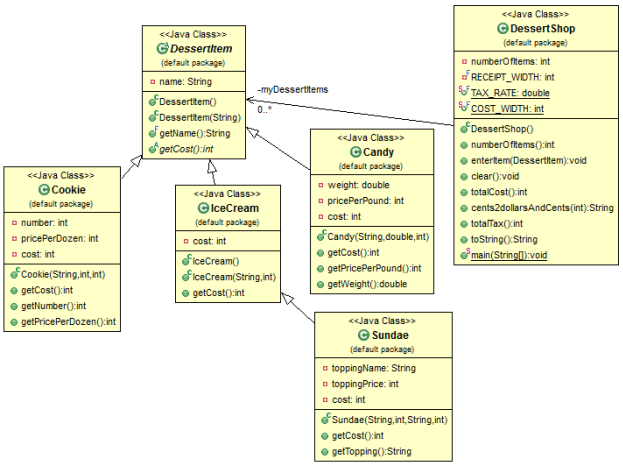

Fig. 14. The given class diagram

The task is the implementation of inheritance hierarchy of classes that derive from an abstract superclass. Please follow the task carefully and develop the required units. It is crucial to follow the given naming for your variables methods and classes as shown in Fig. 14. The following classes are required: (a) DessertItem abstract superclass. (b) Candy, Cookie, IceCream classes which derive from DessertItem superclass and (c) Sundae class which derives from IceCream class. A Candy item has a weight and a price per pound which are used to determine its cost. The cost should be calculated as (cost) * (price per pound). A Cookie item has a number and a price per dozen which are used to determine its cost. The cost should be calculated as (cost) * (price per dozen). An IceCream item simply has a cost, and the cost of a Sundae is the cost of the IceCream plus the cost of the topping. The DessertShop class was given to developers and contained the main functions of the shop and the test for the classes.