Understanding Refactoring Tasks over Time: A Study Using Refactoring Graphs

Aline Brito, Andre Hora, Marco Tulio Valente

[1] ASERG Group, Department of Computer Science (DCC),
Federal University of Minas Gerais (UFMG) – Belo Horizonte, Brazil

{alinebrito,mtov,andrehora}@dcc.ufmg.br

Abstract. *Refactoring is a fundamental practice in modern software development. Therefore, it is essential that practitioners have a solid understanding of refactoring, both in theory and in practice. In this paper, we rely on a graph-based abstraction—called refactoring graphs—to visualize and analyze refactorings performed by students in a canonical refactoring application: the Video Store System, proposed by Fowler. We asked 46 students to perform the refactorings included in this example under two scenarios: according to a list of explicit guidelines, where most students performed the tasks successfully; and by using flexible ones, in which a part of the students faced problems identifying the appropriate operations. We conclude by presenting discussions and implications.*

1. Introduction

Refactoring is a common and indispensable practice during software evolution. Developers frequently refactor the source code for different proposes [Murphy-Hill et al. 2009, Pantiuchina et al. 2020, Silva et al. 2016]. As a consequence, a large number of studies investigate refactoring practices in the last 30 years [Abid et al. 2020]. Many of those studies focus on refactoring comprehension, for example, assessing motivations, benefits, impact, and challenges of refactoring operations [Pantiuchina et al. 2020, Kim et al. 2014, Sellitto et al. 2022]. There are also approaches and tools to understand source code affected by refactoring [Silva et al. 2020, Tsantalis et al. 2020, Brito and Valente 2021, Grund et al. 2021]. However, typically, prior literature does not rely on abstractions to extract, visualize, and understand complex refactoring tasks performed over time, which is a relevant aspect for researchers and practitioners [Bogart et al. 2020, AlOmar et al. 2021a].

Therefore, in this paper, we explore an abstraction called *refactoring graphs* [Brito et al. 2020, Brito et al. 2021] to visualize, interpret, and evaluate refactoring tasks performed over time. For this purpose, we rely on a canonical refactoring example: a Video Store System, proposed by Fowler [Fowler 1999]. This example is used by Fowler to present and discuss refactoring operations, practices, and benefits. Specifically, we invited 46 students from a Software Engineering undergraduate course to perform refactoring operations in the Video Store System under two distinct scenarios. The first one includes a list of *explicit* and well-defined guidelines, i.e., refactoring tasks with detail instructions about the piece of code that should be refactored and the expected operations. The second scenario comprises *flexible* and open guidelines, in which we asked the students to implement the Template Method design pattern [Gamma et al. 1995].

After that, we detect the refactorings performed by the students in distinct commits (i.e., over time) [Silva et al. 2020]. Next, we generated *refactoring graphs* to describe the refactorings performed by each student. In such graphs, the nodes are methods, and the edges represent refactoring operations. Particularly, our guidelines generate a large refactoring graph with 24 vertices and 26 edges.

To better understand the tasks performed by the students, we first rely on a graph-based metric. Specifically, for *explicit guidelines*, we used a similarity metric called *edit distance* to compare each student's graph with our ground truth [Sanfeliu and Fu 1983]. This strategy allowed us to automate the analysis of the tasks performed by the students and to rapidly conclude whether they matched our proposed ground truth. It also allowed us to rapidly identify and analyze the divergences between the student's solution and the ground truth. In the case of *flexible guidelines*, we performed a visual inspection by navigating in the graph-based structure to verify, for example, refactoring operations, sequence of operations, and affected elements. Our key results are summarized as follows:

1. When following *explicit guidelines*, most students performed refactoring tasks successfully (93.5%). The few mistakes refer to refactorings performed in multiples commits.
2. When following *flexible guidelines*, less students implemented the design pattern successfully (70%). Particularly, a significant part of the students faced difficulties identifying the appropriate refactorings operations.

Our contributions are twofold. First, we show that *refactoring graphs* can be used to understand, evaluate, and visualize large refactoring tasks. In some cases, this analysis can be automated using standard graph metrics, such as *edit distance*. Second, we complement research on refactoring practices, mainly on the educational side, by exploring the student's understanding of refactoring.

Structure: Section 2 includes details about the Video Store System and the representation of refactoring operations as graphs. Section 3 describes the study design, while Section 4 shows the results. Section 5 states threats to validity, and Section 6 presents related work. We conclude by prospecting future studies in Section 8.

2. Background

2.1. Video Store System

In the study described in this paper, the students followed the steps proposed by Fowler to refactor a well-known example designed to illustrate the benefits and the mechanisms of refactoring: a system to calculate and print clients' charges at a Video Store [Fowler 1999].[1] Basically, the system prints the movies rented by a client with prices and the number of rented days. The price depends on the movie's category (i.e., regular, children, and new releases) and the number of rented days. The initial system version contains three classes:

- *Movie*: class that represents a movie with the respective data, i.e., title, price, and category.

[1]In the 2nd edition of his book, Fowler changed the system to JavaScript. However, we decided to use the original system since it is still widely known.

- *Rental*: class that represents a client renting a movie. It includes information about the movie and the number of rented days.
- *Customer*: class to represent the store' clients. It comprises information as the client's name and a list of rented movies. There is also a method, named `statement()`, which prints the statement in text format.

The new code design is a consequence of several refactoring operations to improve existing code and incorporate a new feature. Among these refactorings, most cases aim to decompose the large method `statement()` by extracting and moving distinct pieces of the code. Other refactorings generate the class `Price` and their subclasses in seven main steps, as proposed by the State design pattern. In this case, the goal is to remove a conditional statement that determines the price of each video category. There are also classes to print the statement in different formats (i.e., `HTMLStatement` and `TextStatement`), as proposed by the Template Method pattern.[2]

2.2. Refactoring Graphs

We use refactoring graphs to visualize and understand the operations performed by the students in our experiment [Brito et al. 2020, Brito et al. 2021]. Formally, a refactoring graph G contains a set of disconnected subgraphs $G' = (V', E')$. In these graphs, the vertices represent the full signature of methods. The edges refer to the refactoring type (e.g., extract method), and it includes additional information about the operation (e.g., author name, date, and commit). Figure 1 shows an example: extracting method `amountFor(Rental)` from `statement()`, and moving it with a new name to a class `Rental` creates a sequence with two edges in the subgraph. A second refactoring to extract and move the method `getFrequentRenterPoints()` from `statement()` generates another node in the graph.

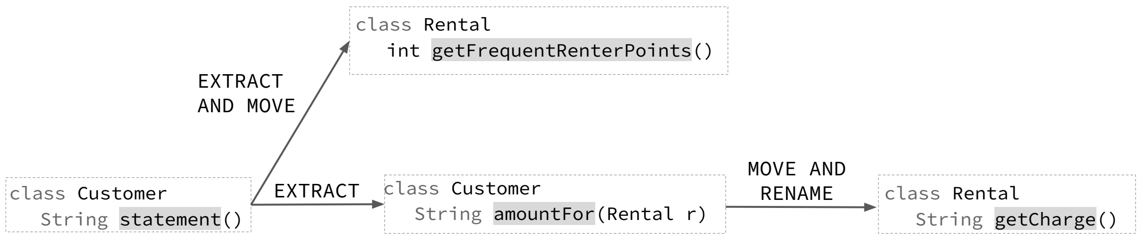


Figure 1. Example of a fragment of ref. subgraph from Video Store System

In this context, following all the steps to refactor the Video Store System creates a large subgraph with 24 vertices and 26 edges, as presented in Figure 2. As we can notice, 14 edges refer to *extract* operations, generating one or more methods, i.e., new vertices in the subgraph. The subgraph also includes two *pull up* methods, six *pull up signatures* (when only the signature is pulled up, preserving the method body in the subclass), and three *push down implementations* (when the signature is kept in the superclass, as an abstract method). There is a single operation called *move and rename* method, which moves a method to a distinct class, changing its name.

[2]The class diagram of the system before and after applying the refactoring tasks is publicly available at: `github.com/alinebrito/refgraph-fowler-study#class-diagram`

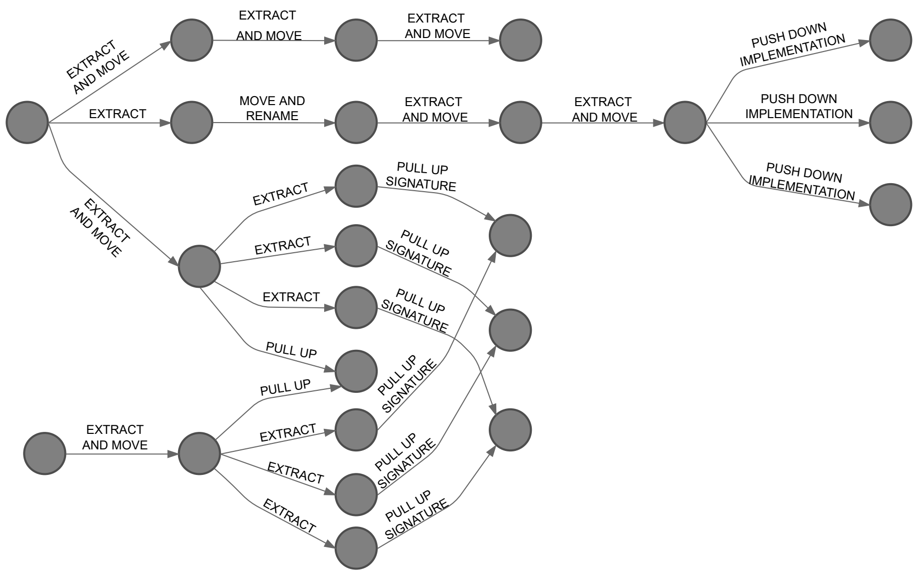


Figure 2. Refactoring subgraph from Video Store System (*ground-truth*)

3. Study Design

3.1. Study Participants

We perform our study with 46 undergraduate students in Computer Science (10 women and 36 men), which we label from $S01$ to $S46$. Specifically, they are students concluding a Software Engineering course. In this course, they have studied topics such as refactoring, testing, design patterns, and clean code. The students could decline participation in the study. Also, we inform the participants that the results could be reported exclusively in an anonymous format. The period to perform the proposed refactoring exercise was one week. The results do not include three students who completed the set of tasks in less than ten minutes.

3.2. Refactoring Tasks & Research Questions

Each student received the three initial classes—`Movie`, `Rental`, and `Customer`—and the instructions to perform the refactorings. We separate the study into two main parts: *explicit* and *flexible* guidelines.

The first part refers to *explicit guidelines*, in which the students received explicit information about the expected refactoring tasks. Specifically, these guidelines indicate the piece of code that should be refactored, the refactoring type, and the signature of the new methods. There are 15 main steps. Three of them introduce new methods or classes, and the other cases involve refactoring operations to make the source code cleaner or to apply changes in the inheritance hierarchy. There is also large refactoring to replace a conditional to polymorphism, which results in a new `Price` hierarchy. Each step finishes with a commit indicating the conclusion of a refactoring task.

After finishing the first part, there are two similar methods in class `Customer`, `statement()` and `htmlStatement()`, which print the statement in ASCII and HTML formats, respectively. Therefore, in the second part of the study, we described to the students the problems of this design structure. For example, it is necessary to duplicate the code to print the statement in other formats, such as CSV, JSON, etc. To mitigate this issue, Fowler proposes the usage of a Template Method. To implement this pattern, it is necessary to extract and move a set of similar methods to new classes. Then, multiple extract operations are needed to make the methods identical. As a final operation, a set of pull up operations should be performed to generate the template code.

Therefore, the second part includes a set of *flexible guidelines*. We invite the students to implement the Template Method pattern to eliminate duplicated code as we previously described. Different from the *explicit guidelines*, we do not indicate operations or details about the piece of code that should be refactored. Each student received only high-level instructions to refactor the methods, in order to create a template method.

We report each guideline results in distinct research questions, which are described in Section 4. Specifically, we present the findings from *explicit guidelines* in RQ1, and the results from *flexible guidelines* in RQ2. We use refactoring graphs as an abstraction to understanding and visualizing the sequence of refactoring operations performed by the students. In RQ1, we rely on a graph similarity metric to detect results that diverge from our ground truth [Sanfeliu and Fu 1983]. In RQ2, we perform a manual inspection of the refactoring subgraphs created from the refactorings performed by the students.

3.3. Detecting Refactoring Operations

We used REFDIFF to identify the refactorings performed by the students [Silva et al. 2020]. As mentioned in a previous section, our study concentrates on six distinct refactoring operations over methods (i.e., *extract*, *extract and move*, *move and rename*, *push down implementation*, *pull up*, and *pull up signature*). REFDIFF is a tool that detects refactorings by analyzing the commits of git-based projects. Therefore, for the 46 projects (i.e., one project per participant), we iterate in the list of commits, comparing each commit with its parent one.

3.4. Execution time

We also computed the time to generate the refactoring graphs. To this purpose, we used a Core i7-8550U workstation with 16 GB of RAM, 5,400 RPM HDD, and Ubuntu 18.04. The process to build our ground truth executed in 18 seconds. As we can observe in Figure 3, regarding the students' projects, the execution times are also low, ranging from 20 to 31 seconds, with a median of 23 seconds. These runtime measures consider the whole process, which includes getting the list of commits, detection of refactoring operations by RefDiff, and generation of refactoring subgraphs. In total, we were able to generate all refactoring subgraphs (of all students) in approximately 18 minutes.

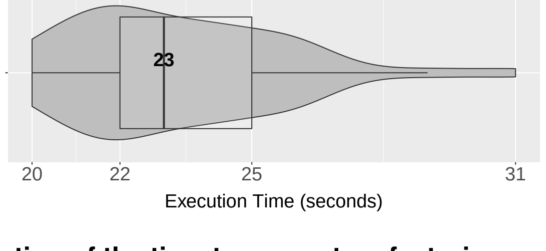


Figure 3. Distribution of the time to generate refactoring graphs per student

4. Results

In this section, we discuss our results. We divide the discussion in two parts: refactorings performed using *explicit guidelines* and refactorings performed using *flexible guidelines*. We rely on refactoring graphs to interpret the refactoring tasks performed by the students.

4.1. (RQ1) How do Students Apply Refactorings When Following Explicit Guidelines?

The instructions from the *explicit guidelines* generate ten edges in the expected refactoring subgraph (top of Figure 2). We compare this subgraph with our ground truth, reporting the number of extra and missing edges. We also assess the similarity by computing the subgraph edit distance. This distance is defined as the minimum number of addition, deletion, and replacement of vertices or edges that makes the student's subgraph identical to the ground-truth graph. As shown in Table 1, only three students made mistakes (6.5%). The other 43 students (93.5%) completed the tasks successfully, i.e., the distance to the ground truth is zero.

Table 1. Subgraphs with mistakes in the explicit guidelines

Student	**Edges**			**Distance**
	Total	**Extra**	**Absent**	
S02	7	-	3	6
S22	9	-	1	1
S44	12	2	-	2

These three students only made minor errors. For example, in the sixth refactoring task, the students should extract and move the method `Rental.getFrequentRenter-Points()` from `Customer.statement()`. However, $S22$ performed this refactoring in two steps. In the first commit, the student copied the code from `statement()` to `get-FrequentRenterPoints()`. In other words, the student created a duplicated code. However, in a second commit, the student removed the duplication, replacing the code with a call to the new method.

There is a similar mistake in $S02$'s subgraph. In the thirteenth refactoring task, the students received instructions to change method `Price.getCharge()` by pushing down the implementation to three subclasses. But, $S02$ also performed this refactoring in two steps, creating a duplicated code. In the first commit, the student copied the code to the subclasses. In a subsequent commit, the student updated the superclass, keeping only the method signature. Since the student did not perform the three *push down implementations* in a single commit, our refactoring mining tool had not detected the refactorings. As a result, the distance between the refactoring graphs is six. In other words, it is necessary to add three edges and three vertices to make the subgraph identical to the ground truth, as shown in Figure 4.

Finally, $S44$ reverted a refactoring task, which generates two extra edges in the subgraph. In this case, the reason is that the student created the new method with only a piece of the expected code. Therefore, $S44$ reverted the operation and after that performed it correctly.

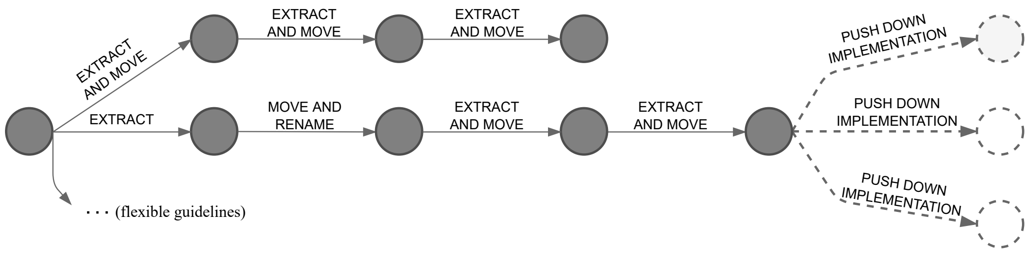


Figure 4. Example of a student's refactoring subgraph. Dashed lines represent missing edges and vertices (*explicit guidelines*, S02).

Summary: When following explicit guidelines on refactoring operations, 93.5% of the students perform the refactoring tasks successfully. The few mistakes refer to refactorings performed in multiple commits. To evaluate the tasks, we used a similarity metric to compare each student's subgraph with the ground truth. We were able to obtain each assessment in a few milliseconds.

4.2. (RQ2) How do Students Apply Refactorings When Following Flexible Guidelines?

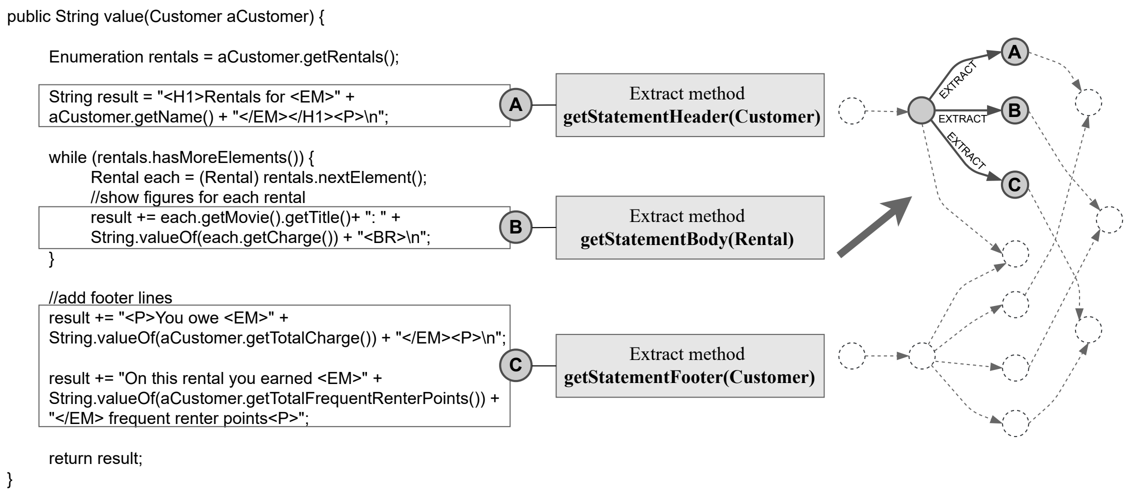
The *flexible guidelines* can be finished with three refactoring tasks. These tasks refer to the implementation of the Template Method pattern, which generates 16 edges in the ground truth refactoring subgraph (bottom of Figure 2). For this study part, we performed a qualitative analysis by manually inspecting the 46 subgraphs and the source code to investigate the students' strategies. In the following paragraphs, we describe the results. Overall, 32 students (70%) performed the *flexible guidelines* successfully.

Task 1. In the first task, the students received instructions to create the superclass `Statement`, as well as the respective subclasses (i.e., `TextStatement` and `HtmlStatement`). Next, they were instructed to extract and move method `statement()` to subclass `TextStatement`, which must contain the code to print the statement in ASCII format. Similarly, it is necessary to extract and move method `htmlStatement()` to subclass `HtmlStatement`, which contains the code to print the statement in HTML format. The two moved methods should have the name `value()`. Most students performed these steps correctly, generating two edges in the subgraph (44 students, 96%). In two cases, minor mistakes refer to unfinished refactorings.

Task 2. In the second task, we invited the students to perform all refactoring operations in order to make the two methods identical, i.e., `HtmlStatement.value()` and `TextStatement.value()`. For each method, we expect at least three extract operations, as presented in Figure 5.

The first operation extracts a method to build the statement's header, returning the client's name in a specific format (HTML or ASCII). The second refactoring generates a method to retrieve the movie's title and charge. Finally, the third extracted method returns the statement's footer, which contains information about the price and the client's renter points. In Figure 5, vertices A, B, and C indicate the operations to decompose `HtmlStatement.value()`. Similarly, there are three extract operations to decompose method `TextStatement.value()`.

We detect that 33 students (72%) performed the expected operations, i.e., they



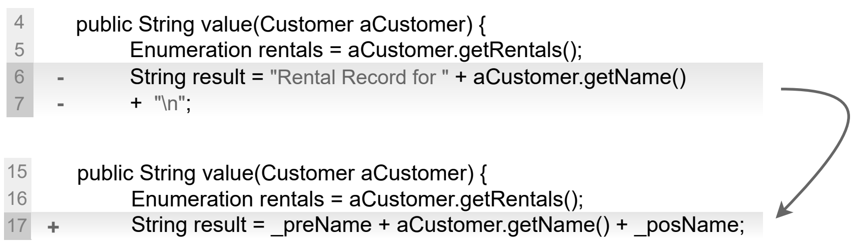
```
public String value(Customer aCustomer) {

    Enumeration rentals = aCustomer.getRentals();

    String result = "<H1>Rentals for <EM>" +
    aCustomer.getName() + "</EM></H1><P>\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += each.getMovie().getTitle()+ ": " +
        String.valueOf(each.getCharge()) + "<BR>\n";
    }

    //add footer lines
    result += "<P>You owe <EM>" +
    String.valueOf(aCustomer.getTotalCharge()) + "</EM><P>\n";

    result += "On this rental you earned <EM>" +
    String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
    "</EM> frequent renter points<P>";

    return result;
}
```

A
B
C
Extract method getStatementHeader(Customer)
Extract method getStatementBody(Rental)
Extract method getStatementFooter(Customer)
EXTRACT
EXTRACT
EXTRACT

Figure 5. Refactoring operations to decompose method value() in subclass HtmlStatement (*flexible guidelines*, *task 2*)

decomposed the methods by extracting three new ones in each subclass. Interestingly, five students (11%) used a different strategy. They divided the statement footer into two methods. The first method returns the price and the second one returns the client's renter points. As a consequence, these students created eight vertices in the subgraph (i.e, four *extract* operations in each subclass).

Eight students (17%) did not perform extract operations to remove the code duplication. Among these cases, five students (11%) added fields. For example, $S22$ used fields to customize the strings, as presented in Figure 6. The values of the fields are defined in the constructors. For instance, field `_preName` receives the value *"<H1>Rentals for <EM>"* in subclass `HtmlStatement`. The same field receives the value *"Rental Record for"* in the subclass `TextStatement`. Finally, $S17$ only added empty methods, and two students did not make the methods identical.



```
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
-       String result = "Rental Record for " + aCustomer.getName()
-       + "\n";

    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
+       String result = _preName + aCustomer.getName() + _posName;
```

Figure 6. Textual diff produced by GitHub. S22 added two fields (_preName and _posName) in the header line (*explicit guidelines*, *task 3*)

Task 3. Assuming the students performed the refactoring tasks as expected, we should have in this step two identical methods `value()` in `TextStatement` and `HtmlStatement`. As a final operation, we requested the students to remove this duplicated code. It is expected at least six *pull up signatures* and two *pull up* methods in this last task, which generates eight edges in the subgraph. Among the 46 students, 74% completed this task successfully (34 occurrences).

Summary: When following flexible guidelines for refactoring operations, most students implemented the *Template Method* pattern successfully (70%). Most mistakes refer to students who faced problems identifying the appropriate operations. Refactoring graphs assisted in visualization and understanding the sequence of refactoring tasks performed by the students.

5. Threats to Validity

5.1. Construct Validity

Refactoring tasks. First, the communication among the students is a possible threat in our study. However, the participants received instructions to avoid sharing their source code. To mitigate this threat, we removed outliers from our results. The period to perform the proposed refactoring exercise was one week. For the *explicit guidelines* (RQ1), about 75% of the students complete the tasks up to 198 minutes (i.e., approximately three hours), with a median of 106 minutes. In case of *flexible guidelines* (RQ2), 75% of the students spent about one hour to conclude the three refactoring tasks (i.e., 65 minutes), with a median of 44 minutes. The highest intervals probably refer to students that paused the refactoring tasks for a while, returning on a distinct day. Considering the distribution of the time, we eliminated the projects of three students, who performed the *explicit* or *flexible* guidelines in less than ten minutes since this is not a viable time to conclude the refactoring tasks. Second, the refactoring tasks include examples and discussions, which are spread over eleven chapters of Fowler's book. In other words, it is not trivial to obtain the answers directly from the book. Still, the usage of Fowler's book by students is a possible threat. We relied on two different strategies to alleviate this threat, i.e., creating not only *explicit* but also *flexible* (or open) guidelines. All students followed these guidelines, and we obtained different results. Particularly, when following the flexible guidelines, a significant part of them faced problems performing the refactoring tasks. Third, the Video Store System may represent a nonessential software project nowadays. We decided to keep this system to preserve consistency with Fowler's catalog, which is widely used in Software Engineering courses. In future studies, we plan to use the new book version [Fowler 2018], which relies on JavaScript and adaption of the system to another context.

Detection of refactoring operations. We removed a single refactoring operation and two projects due to compilation errors or false positives. These cases represent a small piece of the sample. Thus, it is not likely to influence our results. In addition, we used REFDIFF to detect the refactorings [Silva et al. 2020]. We adopted this tool due to its support to multiple programming languages. Therefore, it is possible to replicate the study with the refactorings described in the last edition of Fowler's book, which relies on JavaScript [Fowler 2018]. It is also possible to adapt the guidelines to other programming languages, such as *C* [Silva et al. 2020] and *Go* [Brito and Valente 2020].

5.2. Internal Validity

Student's mistakes. We used *refactoring graphs* as a visual instrument to understand and evaluate the refactoring operations performed by the students. Also, we used a similarity metric called *graph edit distance* to compare our ground-truth subgraph with the students' subgraphs. For this purpose, we rely on a well-known Python library that implements this similarity algorithm.[3] However, our results also rely on the manual inspection of the commits of each student. Thus, we reinforce their subjective nature. The refactoring operations performed by the students (and our analysis) are publicly available at: `alinebrito.github.io/refgraph-fowler-study`

5.3. External Validity

Generalization of the results. Our guidelines relied on the Java programming language and on a canonical refactoring example to introduce refactoring concepts, i.e., the Video Store System proposed by Fowler [Fowler 1999]. Thus, as unusual in empirical software studies, the results can not be generalized to other scenarios, such as different refactoring tasks or programming languages.

Sample size and students' experience. The 46 participants might not be a representative sample. However, they have previously studied several topics in the Software Engineering course (e.g., refactoring, testing, design patterns, and clean code).

6. Related Work

Refactoring is a fundamental practice in software development and evolution. For this reason, several studies focused on this research line over the last 30 years [Abid et al. 2020]. Among them, there are several studies aiming to understand refactoring activities [Kim et al. 2012, Kim et al. 2014, Mahmoudi et al. 2019, Pantiuchina et al. 2020, Paixao et al. 2020]. However, there is also a need for abstractions and tools to organize, visualize, and evaluating large and complex refactorings [Bogart et al. 2020, AlOmar et al. 2021a].

Most studies focus on developers' perception of refactoring. Murphy-Hill *et al.* [Murphy-Hill et al. 2009] used four Eclipse IDE datasets to investigate how developers perform refactoring operations. Considering these metadata, the authors discuss the frequency of refactoring operations, the usage of tools, and refactoring occurring in batches (i.e., in a 60-second time window). Among the main findings, the authors point out that 40% of refactorings occur in batches and developers alternate refactorings with other software activities. Silva *et al.* [Silva et al. 2016] investigated the motivations behind refactoring operations by performing a *firehouse* interview with 195 developers. The authors report 44 distinct reasons, such as improving testability or readability and eliminating duplicated pieces of code. The study also discusses the usage of tools for supporting refactoring, reinforcing a previous study about the prevalence of manual refactorings [Murphy-Hill et al. 2009]. A recent study expands this catalog, by adding 26 new motivations [Pantiuchina et al. 2020]. In this case, the author analyzed a set of 551 pull requests and the respective commits. However, the mentioned studies do not explore abstractions to visualize and understand refactoring tasks over time. In this paper, we rely on *refactoring graphs* to extract and interpret large refactoring tasks performed by students. In the wild, *refactoring graphs* have been used to explore refactorings performed over time, showing that a significant part of the operations is not a sole transformation [Brito et al. 2020, Brito et al. 2021].

On the educational side, research on refactoring covers distinct scenarios. Demeyer *et al.* [Demeyer et al. 2005] propose examples for students. López *et al.* [López

[3]networkx.org/documentation/stable/reference/algorithms/similarity.html

et al. 2014] propose activities to teach refactoring. The study suggests a set of tasks, such as reading texts, comprehension of refactoring catalogs, and practical exercises. Other studies discuss approaches and lessons to promote refactoring practices [Stoecklin et al. 2007, Hebig et al. 2020]. There are also tools to improve the student's perception of refactoring [Agrahari and Chimalakonda 2020]. However, the mentioned studies do not consider graph-based concepts to evaluate refactoring activities over time.

Keuning *et al.* [Keuning et al. 2020] performed a study with 133 students from the second year of a Software Engineering course. The experiment uses a tutoring system, which contains a set of refactoring exercises and provides feedback during the refactoring tasks. Specifically, there is an option to check the student progress, and a second functionality to get hints (i.e., to diagnose and provide suggestions). The students' mistakes were extracted from log entries of the system. However, most of them are outside the scope of refactoring. For example, the authors report compiler errors due to language unsupported constructs, runtime errors, and failed tests. Besides, each of the five exercises involved an isolated issue. For example, two tasks are from a website that contains a list of code problems. In our paper, we use a canonical refactoring example (proposed by Fowler [Fowler 1999], with 18 refactoring tasks) and leverage refactoring graphs to assess refactoring operations performed over time.

Abid *et al.* [Abid et al. 2015] conducted a study with 23 students. Each group performed a kind of project scheme in a real system. The first scheme contains a set of software changes, ending with refactoring tasks. In contrast, the second scheme starts with refactoring issues. The authors reported better results in the second one. However, the study does not concentrate exactly on refactoring comprehension. The authors focus on the main differences by performing refactoring operations before and after functional improvements of the system. Besides, the tasks are performed by two or three students. Thus, different students' perceptions of refactoring can influence the results. In our study, each result refers to a single student. In addition, our guidelines are based on a well-known refactoring example.

Karac *et al.* [Karac et al. 2019] investigated the impact of task granularity on Test-Driven Development (TDD). Overall, the study encompasses five main steps. For example, there are steps to implement a feature and the respective tests. The last step involves refactoring operations to improve code quality. The study involves 52 students, most of them are novice programs. The tasks rely on programming exercises, which are essentially algorithmic. A group of students performed decomposed tasks. In contrast, the second group received tasks without decomposition in small steps. Among the main results, the authors argue about the importance of breaking larger work into smaller ones, mainly for novice practitioners. However, the study concentrates on the granularity of issues, and the refactoring operations are just a step of the experiment.

7. Discussion

First, among the most recurring students' mistakes, there are refactoring operations completed in two steps (i.e., two commits). Refactorings performed along multiple commits represent a threat to current studies on refactoring practices. Particularly, the ones based on tools such as RefactoringMiner [Silva et al. 2016, Tsantalis et al. 2018, Tsantalis et al. 2020] and RefDiff [Brito and Valente 2020, Silva and Valente 2017, Silva et al. 2020].

These tools detect refactorings by computing the “diff” between one commit and its parent. For example, if we assume three sequential commits $C1$, $C2$, and $C3$, the tools will miss a refactoring that starts in $C1$, but that is only finished in $C2$, as we observed when manually analyzing the results of both RQs. Since studies on refactoring rely on the default configuration of RefactoringMiner and RefDiff [Bibiano et al. 2019, Sousa et al. 2020, Hora et al. 2018, Paixao et al. 2020, AlOmar et al. 2021b, Brito et al. 2018], they do not consider refactorings performed over multiple commits. Therefore, we also envision research on new heuristics and techniques to detect refactoring in multiples commits, as well as the investigation of their impact on mining software studies.

Second, several well-known graph algorithms can be used with refactoring graphs, such as algorithms to mine graph patterns and metrics [Xifeng Yan and Jiawei Han 2002, Leung 2010, Sanfeliu and Fu 1983]. For example, in the first research question, we used a similarity metric called *graph edit distance* to compare our ground-truth subgraph with the students' subgraphs. Refactoring graphs also allow easy navigation on refactoring operations. We can inspect, for example, entity names, visualize sequences of refactoring tasks, and identify missing and extra operations (i.e., edges and vertices). In this context, researchers can rely on refactoring graphs to perform empirical studies on complex and large refactoring practices over time, which is a hurdle recently pointed in the literature [AlOmar et al. 2021a].

8. Conclusion

In this paper, we explore the usage of *refactoring graphs* to represent, visualize, and assess large refactoring tasks over time. For this purpose, we invited 46 undergraduate students from a Software Engineering course to refactor a well-known example proposed by Fowler, a Video Store System [Fowler 1999]. Each student received two groups of refactoring tasks. The first group included explicit guidelines, i.e., the precise indication of the piece of code to be refactored and the expected operation. In contrast, the second group received flexible instructions. Specifically, in this second case, we invited the students to implement a design pattern without details about the piece of code to be refactored.

The refactoring tasks create a large refactoring subgraph with 24 vertices and 26 edges. The genereration of all refactoring graphs demanded about 18 minutes, i.e, a median of 23 seconds per student. After building such graphs, we assessed the refactoring tasks using graph-based metrics and performing visual inspections. We summarize our findings as follows:

- When following *explicit guidelines*, most students performed refactoring tasks successfully (93.5%). The few mistakes refer to refactorings performed in multiples commits.
- When following *flexible guidelines*, most students implemented the proposed design pattern successfully (70%). However, a significant number of students faced problems identifying the appropriate refactoring operations.

Based on our results, we provided implications for different scenarios. Specifically, we discuss our contributions regarding graph-based abstractions to support comprehension of large and complex refactoring operations. Finally, we argue about a possible thread in studies based on state-of-the-art mining tools [Tsantalis et al. 2018, Tsantalis et al. 2020, Brito and Valente 2020, Silva and Valente 2017, Silva et al. 2020], which do not detect refactoring performed in multiples commits.

Further studies can include other programming languages and ecosystems (e.g., the current version of Fowler's book uses JavaScript [Fowler 2018]) and novel techniques to detect refactorings in multiples commits. Also, future works can perform experiments with practitioners using *refactoring graphs*. For example, in this study, we consider the professors' perspectives to understand refactoring operations performed by students. Other studies can contemplate the student or developer viewpoint, using refactoring graphs as a tool to visualize refactoring tasks. The dataset is publicly available at: `alinebrito.github.io/refgraph-fowler-study`

Acknowledgments

This research is supported by grants from FAPEMIG, CNPq, and CAPES.

References

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T., and Dig, D. (2020). 30 years of software refactoring research: A systematic literature review. *ArXiv*, abs/2007.02194.

Abid, S., Abdul Basit, H., and Arshad, N. (2015). Reflections on teaching refactoring: A tale of two projects. In *20th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, page 225–230.

Agrahari, V. and Chimalakonda, S. (2020). Refactor4green: A game for novice programmers to learn code smells. In *42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 324–325.

AlOmar, E., Mkaouer, M., and Ouni, A. (2021a). *Knowledge Management in the Development of Data-Intensive Systems*, chapter Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet? Auerbach Publications.

AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2021b). Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software (JSS)*, 171:110821.

Bibiano, A. C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., and Cedrim, D. (2019). A quantitative study on characteristics and effect of batch refactoring on code smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

Bogart, A., AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2020). Increasing the trust in refactoring through visualization. In *42nd International Conference on Software Engineering Workshops (ICSEW)*, pages 334–341.

Brito, A., Hora, A., and Valente, M. T. (2020). Refactoring graphs: Assessing refactoring over time. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377.

Brito, A., Hora, A., and Valente, M. T. (2021). Characterizing refactoring graphs in Java and JavaScript projects. *Empirical Software Engineering*, 26(6):1–43.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 507–511.

Brito, R. and Valente, M. T. (2020). RefDiff4Go: Detecting refactorings in Go. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 101–110.

Brito, R. and Valente, M. T. (2021). RAID - Refactoring aware and intelligent diffs. In *29th International Conference on Program Comprehension (ICPC)*, pages 1–11.

Demeyer, S., Van Rysselberghe, F., Girba, T., Ratzinger, J., Marinescu, R., Mens, T., Du Bois, B., Janssens, D., Ducasse, S., Lanza, M., Rieger, M., Gall, H., and El-Ramly, M. (2005). The LAN-simulation: a refactoring teaching example. In *8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 123–131.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

Grund, F., Chowdhury, S., Bradley, N., Hall, B., and Holmes, R. (2021). CodeShovel: Constructing method-level source code histories. In *43rd International Conference on Software Engineering: Companion Proceedings (ICSE)*, pages 1–13.

Hebig, R., Ho-Quang, T., Jolak, R., Schröder, J., Linero, H., Ågren, M., and Maro, S. H. (2020). How do students experience and judge software comprehension techniques? In *28th International Conference on Program Comprehension (ICPC)*, page 425–435.

Hora, A., Silva, D., Robbes, R., and Valente, M. T. (2018). Assessing the threat of untracked changes in software evolution. In *40th International Conference on Software Engineering (ICSE)*, pages 1102–1113.

Karac, E. I., Turhan, B., and Juristo, N. (2019). A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development. *IEEE Transactions on Software Engineering (TSE)*, pages 1–16.

Keuning, H., Heeren, B., and Jeuring, J. (2020). Student refactoring behaviour in a programming tutor. In *20th Koli Calling International Conference on Computing Education Research (Koli Calling)*, pages 1–10.

Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 50:1–50:11.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring challenge and benefits at Microsoft. *IEEE Transactions on Software Engineering (TSE)*, 40(7):633–649.

Leung, C. (2010). Technical notes on extending gSpan to directed graphs. Technical report, Singapore Management University.

López, C., Alonso, J. M., Marticorena, R., and Maudes, J. M. (2014). Design of e-activities for the learning of code refactoring tasks. In *2014 16th International Symposium on Computers in Education (SIIE)*, pages 35–40.

Mahmoudi, M., Nadi, S., and Tsantalis, N. (2019). Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *31st International Conference on Software Engineering (ICSE)*, pages 287–297.

Paixao, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (2020). Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *17th International Conference on Mining Software Repositories (MSR)*, pages 1–12.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., and Penta, M. D. (2020). Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 37(4):1–32.

Sanfeliu, A. and Fu, K.-S. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, SMC-13(3):353–362.

Sellitto, G., Iannone, E., Codabux, Z., Lenarduzzi, V., Lucia, A. D., Palomba, F., and Ferrucci1, F. (2022). Toward understanding the impact of refactoring on program comprehension. In *29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1–12.

Silva, D., da Silva, J. P., Santos, G., Terra, R., and Valente, M. T. (2020). RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering (TSE)*, 1(1):1–17.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? Confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858–870.

Silva, D. and Valente, M. T. (2017). RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1–11.

Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A. C., Oliveira, D., Kim, M., and Oliveira, A. (2020). Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *17th International Conference on Mining Software Repositories (MSR)*, pages 186–197.

Stoecklin, S., Smith, S., and Serino, C. (2007). Teaching students to build well formed object-oriented methods through refactoring. In *38th Technical Symposium on Computer Science Education (SIGCSE)*, pages 145–149.

Tsantalis, N., Ketkar, A., and Dig, D. (2020). RefactoringMiner 2.0. *IEEE Transactions on Software Engineering (TSE)*.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE)*, pages 483–494.

Xifeng Yan and Jiawei Han (2002). gSpan: graph-based substructure pattern mining. In *2nd International Conference on Data Mining (ICDM)*, pages 721–724.