Application of Constructive Modeling and Process Mining Approaches to the Study of Source Code Development in Software Engineering Courses

Viktor Shynkarenko, and Oleksandr Zhevaho

Original scientific article

***Abstract*—We present an approach of constructing a source code history for a modern code review. Practically, it is supposed to be used in programming training, especially within initial stages. The developed constructor uses constructive-synthesizing modeling tools to classify a source code history by fine-grained changes and to construct an event log file aimed to provide information on students' coding process. Current research applies Process Mining techniques to the software engineering domain to identify software engineering skills. By better understanding of the way students design programs, we will help novices learn programming. This research provides an innovative method of using code and development process review in teaching programming skills and is aimed to encourage using code review and monitoring coding practice in educational purposes. The standard method of evaluation takes into consideration only a final result, which doesn't meet modern requirements of teaching programming.**

***Index Terms*—source code history, constructive-synthesizing modeling, process mining.**

I. Introduction

Programming requires a set of competencies, and studying them is fundamental in computer science education. Understanding the way students develop software and challenges they face, has great potential for improving the quality of teaching coding skills. In the process of teaching the basics of software engineering, it is significant to spot emergent problems and help with their elimination, to control individuality and quality of solving tasks, and to be aware of the difficulties each student faces while learning. The standard approach of evaluation takes into consideration only a final result, which doesn't meet modern requirements of teaching programming.

The quality of software is directly associated with the quality of corresponding developing process. One of the most examined methods of refining a program quality is using code review [1]. A modern code review, frequently used in practice these days, is informal, tool-based, and asynchronous [2]. Typically, this approach is implemented in software companies. Their experience may be effective in computer science education. Over the past few years, there have been studies on the effective using of code review in teaching programming carried out [3, 4]. The findings show that code reviews can assist students in looking back at their performance and improving their software development skills.

In our previous works [5–7], we have introduced the tool for automatic monitoring and visualizing software development and debugging processes. We suggest automating this process by implementing specially designed extension for Microsoft Visual Studio Integrated Development Environment (IDE) to provide the possibility of determining style characteristics of every student in the classroom and each individual work.

The study provides an innovative method of using code and software development process review in teaching programming skills and is aimed to stimulate using code review and monitoring coding practice in educational institutions. We present an approach of constructing a source code history using constructive-synthesizing modeling (CSM) for a modern code review. We have developed a constructor that classifies a source code history by fine-grained changes and constructs an event log file. We use the Process Mining approach to determine software engineering skills. By analyzing IDE usage data, we aspire to provide software development process with novel insights. Based on Process Mining techniques, we hope to discover coding patterns, programmers' behavior, and to detect problems. The development process model obtained using Process Mining will provide teachers with characteristics that indicate design flaws of functionally correct code which can affect its quality – so-called code smells.

The rest of this paper is structured as follows. An overview of related studies is provided in Section II. Section III and

Manuscript received February 25, 2021; revised October 1, 2021. Date of publication December 3, 2021. Date of current version December 3, 2021. The associate editor prof. Dinko Begušić has been coordinating the review of this manuscript and approved it for publication.

Authors are with the Computer and Information Technologies Department, Dnipro National University of Railway Transport named after academician V. Lazaryan, Dnipro, Ukraine (e-mails: shinkarenko_vi@ua.fm, marakonec@gmail.com).

Digital Object Identifier (DOI): 10.24138/jcomss-2021-0046

Section IV provides more details on the proposed approach. An illustrative example is presented in Section V. Finally, conclusions are presented in Section VI.

II. Related Work

Empirical studies in software development are most often based on data extracted from version control systems (VCS) [8, 9], but not on IDE, since they don't track developer's activities. Analysis of source code changes committed to VCS is the most usual way to analyze software production data. In recent years, there have been several reports of successful using of VCS in computer science courses [10, 11]. However, the VCS can't evaluate contribution of a student to the ultimate result, because they don't provide information on source code production process. VCS only capture a history of changes between commits. Negara et al. [12] shows that history from VCS doesn't reflect real code evolution.

In [13, 14], the authors investigated the quality of students' programs. The findings expose many quality flaws and lack of expected quality improvement among the source codes of first- and second-year students. Several published studies have shown that continuous monitoring leads to significant improvements in students' achievements [4, 15]. Snipes et al. [16] provided practical guidance of using the IDE. They showed that the tools for collecting IDE usage data provided a more detailed comprehension of developers' work than was possible previously. To collect information about the development process in the IDE, events are typically used [17–20]. However, most of those tools only track invocation of commands during coding sessions, without more detailed context data. In [21], tools are implemented to record interactions with IDE and to combine this data with more extensive context information.

Only a few studies have reported about using code review in computer science courses [1, 22, 23]. However, the findings suggest that using code reviews by students increases their self-confidence and that the benefits gained in the classroom are similar to the benefits found in production.

Over the last years, there have been several languages suggested for modeling a development process [24]. This work contemplates the use of CSM in constructing source code history. Fundamentals of CSM provided in [25–28] allow modeling any construction and process. Also we use the Process Mining approach to extract a model of a program development process.

Over the recent decade, Process Mining has become a modern field of research that focuses on analysis of processes via event data. Process Mining is aimed to discover, monitor, and enhance processes that occur when applying data from an event log received from an information system [29]. The IEEE Task Force has published the Process Mining Manifesto [30]. This manifesto was supported by 53 organizations and 77 Process Mining experts. It is aimed to advance the topic of Process Mining. What's more, by determining a set of leading rules and listing critical issues, this manifesto is to serve as a manual for software engineers, scientists, and end-users. Rubin et al. [31] shows that Process Mining may be equally applied to software. Process Mining methods have already been used to research software development process [32, 33]. Our goal is to introduce novel insights into software development process by analyzing the ways developers use their IDE. Better understanding of the ways students produce code will help us assist novices in studying programming.

III. Constructive Modeling of the Software Development Process

The process of classifying source code change history by fine-grained changes and creating an event log file is a constructive process and consists of elementary actions. Therefore, CSM is used to formalize it. In addition, the reason for choosing this approach is that it interacts well with Process Mining.

Previously, a wide range of tasks in which CSM was applied was shown, which indicates the universality and prospects of CSM application for solving problems in various subject areas, as well as the high generality and typicality of the procedures of this modeling method [34]. CSM formalization allows describing not only the structure of objects but also their properties, to determine the permissible operations on them.

The development of the constructor involves definition of heterogeneous extensible carrier, relation and corresponding operations signatures such as binding and converting carrier elements, substitution, inference, operations on attributes, and also a finite set of requirements of informational support of construction. Informational support of construction includes: ontology, goal, rules, constraints, initial and completion conditions of construction.

Particular qualities of CSM are [25–28]: attribution of elements and operations, extensible carrier, and the model of the executor as its algorithms.

The main points of ontological maintenance of CSM are presented in [34].

In its informal submission, the ontology of the generalized constructor is discussed in [25, 26]. The work provides only the components that are necessary for the upcoming presentation.

The first stage of design is specialization of a generalized constructor. Specialization defines such domain ontology as: the carrier's semantic nature, the goal, the finite set of operations, their semantics and attributes, the order of execution and restrictions [25].

In our previous work we presented the constructor which purpose was to construct a history of source code changes [6]. In current work, we present a constructor-converter from source code changes history to an event log file. The purpose of the constructor is to classify history of source code by fine-grained changes and to create an event log file. Inference begins with initial conditions – non-terminal σ.

Completion conditions – all source code changes are classified and an event log file is generated. Specialization of a generalized constructor based on a constructive-synthesizing approach of constructing an event log file can be considered as:

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_S\!\mapsto C_{EL} = \langle M_{EL}, \Sigma_{EL}, \Lambda_{EL} \rangle, \quad (1)$$

where ${}_S\!\mapsto$ is a specialization operation (performed by an external executor), M_{EL} is a heterogeneous replenishable carrier, which includes a set of terminals and non-terminals, Σ_{EL} is a set of relations and relevant operations, Λ_{EL} is informational support of construction.

The terminal symbols with their attributes are:

- ${}_{traces}log$ is an event log file. Its attributes are: *traces* – an array of traces that comprises a chain of activities;
- ${}_{trace}{}^{index,l}traces$ is an array of traces. Its attributes are: *index* – index of trace *trace* in the array; l – array size;
- ${}_{id,tsdt,tfdt,tpn,tdn,events}trace$ is an array of events created by a single execution of a process. Its attributes are: *id* – identifier; *tsdt* – start timestamp; *tfdt* – finish timestamp; *tpn* – project; *tdn* – developer; *events* – an array of events that occurred during a development session;
- ${}_{event}{}^{index,l}events$ is an array of trace events. Its attributes are: *index* – index of an event *event* in the array; l – array size;
- ${}_{ten,tet,tec}event$ is a trace event. Its attributes are: *ten* – name, *tet* – context, an object of a dynamic structure with information about the environment of an event.

The introduced attribute operations are:

- $\prec(L, nr)$ – operation of adding a new record nr to the array L;
- $\circ(t)$ – operation of setting attribute values of the terminal t by an external performer;
- $\propto(cs, i, ch)$ – operation of getting an element by the index i from the cs array and setting its value to ch;
- $\cong(cs, i, cat)$ – operation of getting an element by the index i, from the cs array, classifying its value and setting it to cat;
- $\pm(events, traces)$ – operation of grouping *events* by *traces*;
- $\mp(traces, \log)$ – operation of saving the traces to an event log file.

The signature $\Sigma_L = \langle \Xi, \Theta, \Phi, \{\rightarrow, \hookleftarrow\}, \Psi \rangle$ consists of finite operations sets and relevant relations, where $\Xi \supset \{\bullet, :\}$ are operations of binding and transforming carrier elements, $\Theta = \{\Rightarrow, |\Rightarrow, ||\Rightarrow\}$ are operations of substitution and inference, Φ are operations over attributes, and also relations of substitution $(\rightarrow)$ and attributiveness $(\hookleftarrow)$, $\Psi = \{\psi_i : \langle s_i, g_i \rangle\}$ is a finite set of substitution rules, s_i is a sequence of substitution relations, g_i is a finite set of attribute operations. If attribute operations are not performed, the substitution rule will look like $\langle s_i, \varepsilon \rangle$, where ε is the empty symbol.

The interpretation is association of an algorithm that performs a certain algorithmic structure with an operation signature. In the interpretation process, a constructor model and an internal executor are connected. It results into a constructive system. The external performer carries out the interpretation operation [25, 26].

For self interpretation C_{EL} needs to clarify a basic algorithmic structure (BAS) [25, 26].

Let the next BAS be approachable:

$$C_{A,EL} = \langle \mathrm{M}_{A,EL}, V_{A,EL}, \Sigma_{A,EL}, \Lambda_{A,EL} \rangle, \quad (2)$$

where $V_{A,EL}$ is a finite set of basic algorithms of an internal performer of construction.

The following algorithms introduce the implement operations on attributes:

$$A_1|^{L}_{L,nr},\ A_2|^{t}_{t}, A_3|^{ch}_{cs,i},\ A_4|^{cat}_{cs,i},\ A_5|^{traces}_{events}, A_6|^{log}_{traces}, \quad (3)$$

where A_i is an identifier of algorithm and X_i, Y_i are sets of its definitions and values.

The creation of a constructive system is:

$$\begin{aligned} &\langle C_{EL} = \langle M_{EL}, \Sigma_{EL}, \Lambda_{EL} \rangle, \\ &C_{A,EL} = \langle M_{A,EL}, V_{A,EL}, \Sigma_{A,EL}, \Lambda_{A,EL} \rangle \rangle \\ &{}_I\!\mapsto C_{I,EL} = \langle M_{I,EL}, \Sigma_{I,EL}, \Lambda_{I,EL} \rangle, \\ &\Lambda_{I,EL} = \Lambda_{EL} \bigcup \{(A_1|^{L}_{L,nr} \hookleftarrow \prec), (A_2|^{t}_{t} \hookleftarrow \circ), \\ &(A_3|^{ch}_{cs,i} \hookleftarrow \propto), (A_4|^{cat}_{cs,i} \hookleftarrow \cong), \\ &(A_5|^{traces}_{events} \hookleftarrow \pm), (A_6|^{\log}_{traces} \hookleftarrow \mp)\}, \end{aligned} \quad (4)$$

where ${}_I\!\mapsto$ is an interpretation operation.

A constructor concretization includes a definition of specific rules, restrictions, starting terms, terms of construction termination and such concretization carrier element basis as: finite sets of its non-terminal and terminal characters with their properties and values of properties. After interpretation and concretization operations, executed by an external performer, a constructive system will have all it needs for autonomous creation of constructions [25, 26].

We carry out the following concretization of the constructor C_{EL} designed for creation of an event log file with classified source code changes:

$$\begin{aligned} &C_{I,EL} = \langle M_{I,EL}, \Sigma_{I,EL}, \Lambda_{I,EL} \rangle_K \mapsto \\ &C_{K,EL} = \langle M_{K,EL}, \Sigma_{K,EL}, \Lambda_{K,EL} \rangle, \end{aligned} \quad (5)$$

where ${}_K\!\mapsto$ is a concretization operation.

The record of sequential execution of the rules will be denoted as $\prod_{i=1}^{n} s_i$.

Substitution rules are described below (6, 7).

Rule s_1 receives a history of source code changes cs from an external executor and sequentially executes rule s_2 for each change. Rule s_2 classifies the change and assigns a category to an event file

$$s_1 = \langle \sigma \to cs \bullet \prod_{i=1}^{l_{\perp} cs} \alpha_i \rangle, g_1 = \langle \circ(cs) \rangle \tag{6}$$

$$\begin{aligned} &s_2 = \langle \log \alpha_i \to \varepsilon \rangle, g_2 = \langle \propto (cs, i, ch), \\ &\cong (cs, i, cat), ten_{\leftarrow} \lrcorner event = cat, tet_{\leftarrow} \lrcorner event = t_{\leftarrow} \lrcorner ch, \\ &context_{\leftarrow} \lrcorner event = chc_{\leftarrow} \lrcorner ch, \prec (events, event), \\ &\pm (events, traces), \mp (traces, \log) \rangle. \end{aligned} \tag{7}$$

It results into creation of an event log file with classified source code changes.

IV. Application of Process Mining Approach to Software Development Process

An event log works as a starting point for Process Mining. Each event in such log applies to an activity that may be performed on a resource at specific time and for a specific case. An event log is structured as a set of traces, where each trace comprises a chain of activities created by a single execution of a process (a case). At least, an event record includes an identifier of a case of a process to which an event is applied, a timestamp, and a variety of complementary attributes. The description of every attribute kept in the event log is shown in Table I.

TABLE I
Description of Attributes kept in the Event Log File

Attribute	Level	Description
Name	Trace	Session identifier
StartedDateTime	Trace	Session start timestamp
FinishedDateTime	Trace	Session finish timestamp
Project	Trace	Project identifier
Developer	Trace	Developer identifier
Activity	Event	Change type
Timestamp	Event	A time marker when the event occurs
Context	Event	Context of event

Over the past decade, Process Mining has become a modern field of research that focuses on analysis of process using event data.

In this paper, we decided to store an event log in eXtensible Event Stream (XES) format [35], which is a standard format for Process Mining, designed by IEEE Task Force for logging events. The file contains classified source code changes according to the types shown in Table II.

We use Process Mining discovery techniques to construct a model of a program development process. Extracting a process model allows one to get the way and order a process was performed. We use ProM to extract a development process model from an event log. ProM is a common open-source framework, de-facto standard, for implementing Process Mining [36].

TABLE II
Types of Source Code Changes

Element type	Change type
Class	add, remove, rename, move
	add/remove/change comment
	add/remove/change modifier (abstract, static, etc.)
	change of accessibility
	add/remove/change inheritance
	add/remove/change attribute
Interface	add, remove, rename, move
	add/remove/change comment
	change of accessibility
	add/remove/change inheritance
Field	add, remove, rename
	add/remove/change comment
	add/remove/change modifier (const, static, etc.)
	change of accessibility
	change type
	add/remove/change initializer
	add/remove/change attribute
Method	add, remove, rename
	add/remove/change comment
	add/remove/change modifier (abstract, static, etc.)
	change of accessibility
	add/remove/change attribute
	add/remove/rename parameter
	change parameter type
	change parameter order
	add/remove/change parameter assignment
	change return type
	add/remove/change parameter modifier (ref, out, etc.)
Method body	add/remove/change inline comment
	add/remove/rename variable
	add/remove/change variable assignment
	change variable type
	add/remove/change variable modifier (const, etc.)
	add/remove/change method invocation
	add/remove/change object instantiation
	add/remove/change if/else/assignment/catch/throw/switch/case/return/lock/using/yield statement
	add/remove/change postfix/prefix expression
	add/remove/change for/foreach/do/while loop
	add/remove continue/break/try/finally/default statement

ProM's Inductive Visual Miner follows framework directly [37], which allows Directly Follow Graphics (DFG) to be used to discover a development process model. A DFG represents activities as rectangles and links two activities together if one of them directly follows the other. Besides, each edge has a value pointing out the number of entries to an event log.

V. Illustrative Example

In this section we will give a precise example to illustrate the way the approach described in the previous sections can be

applied in real environment.

The task is to write a program to calculate the minimum number of coins needed to give change to a customer. As input, the program accepts an array of coin denominations and the amount of change needed. The result is shown in Listing I.

Evaluation of student's learning effect plays an important role in education, and is usually done by assessing student's final results. Now, on the basis of this program, a teacher should understand and evaluate skills and abilities of a programmer. It's quite difficult to do, estimating a final result only. Therefore, we suggest using not only a code of a program, but also a process of writing to evaluate student's work.

In our previous work [6] we suggested a code-writing history constructor. It was meant to construct a history of writing a program text. As a result, a formed array of chains completely reflects a history of a source code. The history of writing the program from Listing I is shown in Listing II.

Code history is presented as a sequence of chains, each of which contains the following information: a serial number, type of change ('+' – add, '-' – delete, '*' – edit), code, file name, line number in the final version of a program or negative identifier, a timestamp that indicates when changes were made.

The constructor presented in Section 3 classifies a source code history by fine-grained changes and creates an event log file aimed to examine student's coding process. The file contains a sequence of events according to types of source code changes (see Table II).

It results into the process model shown in Fig. 1.

Implementation of above-mentioned techniques provides an explicit picture of developer's behavior during coding sessions. The information extracted from process of writing a program text can be used to automatically provide a student with recommendations for improving their code, give information about effective programming style, and track the tendency of its change.

Analysis of a history of writing a program gives information about the amount of time it took a student to write a text of a program, which parts caused the greatest difficulties. Based on several programs of a particular student, analysis also reveals their style of work and indicates characteristic features.

Analysis of the program (see Listing 1) and the history of its writing (see Listing 2) shows that during the implementation, the programmer initially had a different solution (chains 1 – 33) than the one presented in the result. During testing, after making sure that the chosen solution did not fit well enough, the algorithm was redone (chains 34 – 45). You can also notice that initially the whole program code was written inside the *main* function, and only then, using the *Extract Method* refactoring, the logic for finding the minimum number of coins was moved to a separate method. This style violates the principles of the stepwise refinement method, which must be pointed out to a student as a result of the test. Besides, comments on the method were written at the very end, which also is a sign of a "bad" style. You can also highlight the positive aspects, such as style of naming variables and methods. The names were thought out immediately, and not at the end, when the program is almost completed.

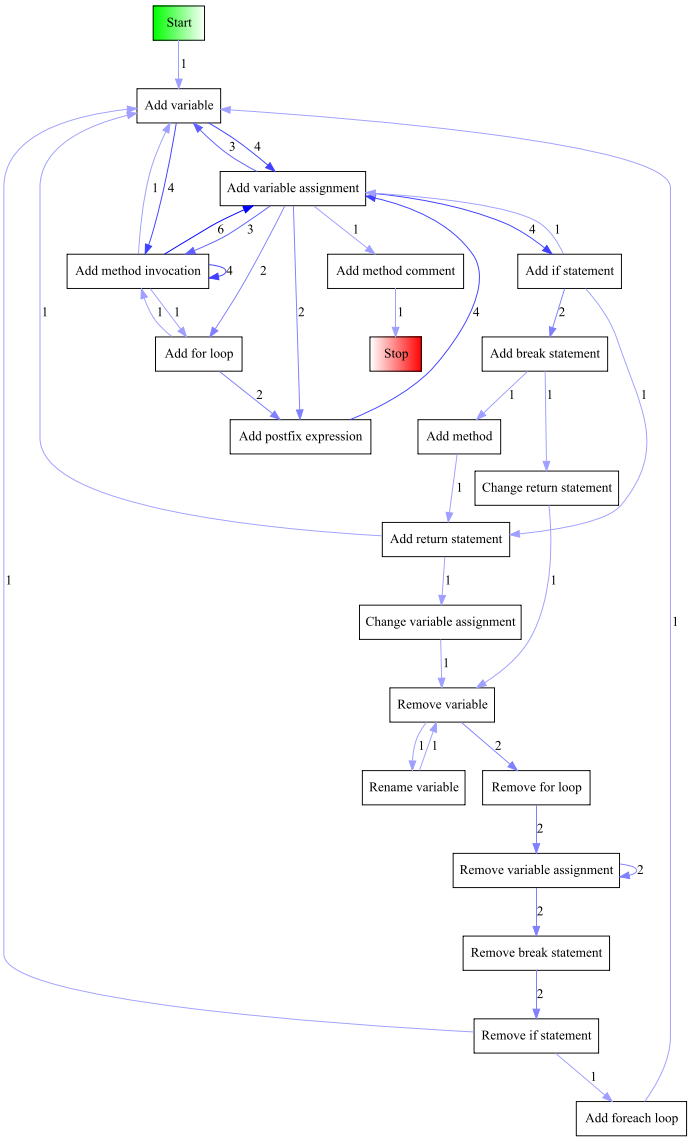

Fig. 1. Process model discovered using the Inductive Visual Miner

A direct experiment was performed on individual students with intuitively known abilities. Three students with a high and three students with a low level of programming skills were selected. The level of students' skills was known in advance and was determined by the teacher's expert evaluations in programming-related disciplines. The experiment aimed to establish the applicability of the method, not to collect statistical information and determine skills.

As a result of the direct experiments, we confirmed the work of our approach. The results obtained correlate well with the predetermined level of the student.

Fig. 2-3 show the process models of students with different skills.

The teacher's analysis of the graphs can lead him to the conclusion that the process model in Fig. 3 is consistent with the stepwise refinement method, while the process model in Fig. 2 violates it.

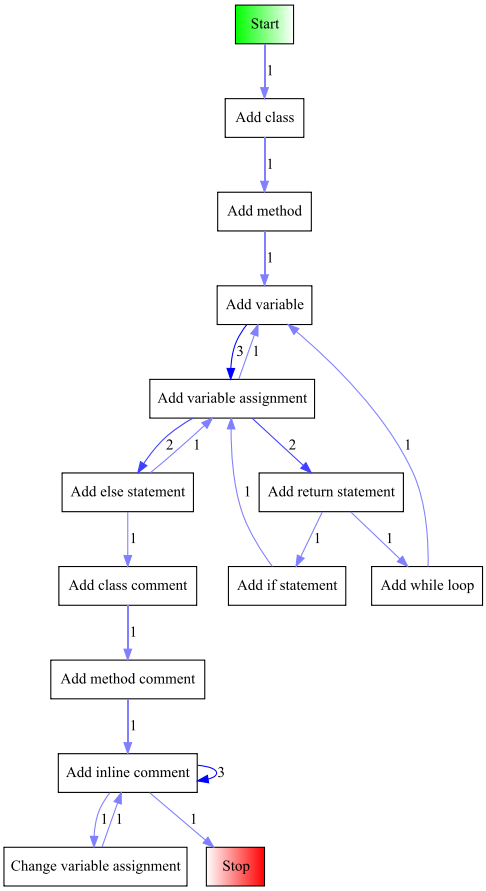

Fig. 2. Process model with deviations from the stepwise refinement method

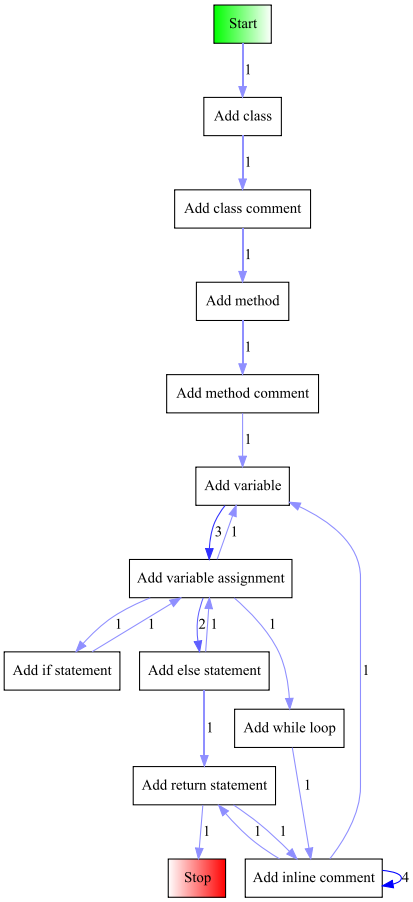

Fig. 3. Process model according to stepwise refinement method

In Fig. 3 each task is detailed in the next iteration, and in Fig. 2 it is expressed in the chaotic writing of the program and making changes to already written code, as well as untimely addition of comments.

The teacher should pay attention to:

– untimely addition of comments, which appears if there is no add comment event after the “Add class” or “Add method” events;

– large number of change events;

– renaming events.

Violation of all this indicates ignoring the method of stepwise refinement, which in turn leads to a significant complication of the program development process.

With the proposed approach, the teacher will be able to evaluate the work based not only on the end result, but on the process of achieving it.

As a result, flaws in the process are shown, and grading is left to the teacher. It is up to the teacher to decide which violations to lower the grades for, and where to make recommendations and monitor their implementation in future works.

VI. Conclusions

We proposed an innovative method of using code and development process review in teaching programming skills. We implemented CSM tools constructor for modeling software development process. The model of source code history was discovered applying Process Mining methods.

It is shown that Process Mining methods are useful for understanding the way programmers perform code production activities, and the difficulties they typically face. Based on the discovered model, we suggest applying adaptive teaching methods for students with varying degrees of academic achievement, which can relieve the process of learning programming skills.

Practical realization of the suggested approach will provide a teacher with the opportunity to conduct a modern code review, which, in our opinion, apart from reviewing a code, should also include an overview of software development process in general. A visual representation will provide a teacher with information about a style of each student’s work with an explanation of their characteristics.

The automated monitoring system presented here and in our previous works is novel in the sense that it controls both student’s programming activities and final functionality of their work.

This paper is the first step towards understanding student’s coding skills. Our ultimate goal is to improve coding skills of novice developers. In a future study, we intend to conduct an experiment for better evaluation of suggested approach in real-life conditions. Our aim is to detect patterns and anti-patterns (smells) on the basis of source code change history and develop an expert system that will make recommendations to students and teachers.

LISTING I
STUDENTS' PROGRAM

```
1 public class Program {
2 public static void Main(string[] args) {
3 int[] cd = new int[5];
4 int change = 0;
5 Console.WriteLine("Enter denominations");
6 for (int i = 0; i < cd.Length; i++) {
7 cd[i] = Int32.Parse(Console.ReadLine()); }
8 Console.WriteLine("Enter the amount of change");
9 change = Int32.Parse(Console.ReadLine());
10 int result = GetMin (change, cd);
11 Console.WriteLine($"Min number: {result}");
12 Console.ReadLine(); }
13 /// <summary> Comment </summary>
14 /// <param name="change">Change</param>
15 /// <param name="cd">Array of denom.</param>
16 /// <returns> The min number of coins </returns>
17 public static int GetMin (int change, int[] cd) {
18 if (cd.Contains(change)) {
19 return 1; }
20 int result = change;
21 foreach(var coin in cd.Where(d => d <= change)) {
22 int count = 1 + GetMin (change - coin, cd);
23 if (count < result) {
24 result = count; } }
25 return result; } } }
```

LISTING II
CODE-WRITING HISTORY

```
1|+|int[] arr…|Program.cs|3|22-11-2020 10:35
2|+|int result = 0…|Program.cs|4|22-11-2020 10:36
3|+|Console.Write…|Program.cs|5|22-11-2020 10:37
4|+|for(int…|Program.cs|6|22-11-2020 10:38
5|+|arr[i] = Int32…|Program.cs|7|22-11-2020 10:39
6|+|Console.Write…|Program.cs|8|22-11-2020 10:40
7|+|change = Int32…|Program.cs|9|22-11-2020 10:41
8|+|Console.Write…|Program.cs|11|22-11-2020 10:42
9|+|Console.Read…|Program.cs|12|22-11-2020 10:43
10|+|int[] sortedArr…|Program.cs|-1|22-11-2020 10:44
11|+|for(int…|Program.cs|-2|22-11-2020 10:45
12|+|result +=…|Program.cs|-3|22-11-2020 10:46
13|+|change = change…|Program.cs|-4|22-11-2020 10:47
14|+|if (change…|Program.cs|-5|22-11-2020 10:48
15|+|public static…|Program.cs|17|22-11-2020 10:49
16|+|return 0;|Program.cs|25|22-11-2020 10:50
17|+|int result = 0;|Program.cs|20|22-11-2020 10:51
18|+|int[] sortedArr…|Program.cs|-6|22-11-2020 10:52
19|+|for(int i…|Program.cs|-7|22-11-2020 10:53
20|+|result +=…|Program.cs|-8|22-11-2020 10:54
21|+|change = change…|Program.cs|-9|22-11-2020 10:55
22|+|if (change ==…|Program.cs|-10|22-11-2020 10:56
23|*|return result;|Program.cs|25|22-11-2020 10:57
24|*|int change = 0;|Program.cs|4|22-11-2020 10:58
25|*|int[] cd =…|Program.cs|3|22-11-2020 10:59
26|-|int[] sortedArr…|Program.cs|-1|22-11-2020 11:00
27|-|for(int i = 0…|Program.cs|-2|22-11-2020 11:01
28|-|result += chan…|Program.cs|-3|22-11-2020 11:02
29|-|change = change…|Program.cs|-4|22-11-2020 11:03
30|-|if(change == 0)…|Program.cs|-5|22-11-2020 11:04
31|+|int result =…|Program.cs|10|22-11-2020 11:05
32|*|for(int i = 0…|Program.cs|6|22-11-2020 11:06
33|*|cd[i] = Int32…|Program.cs|7|22-11-2020 11:07
34|+|if(cd.Contains(…|Program.cs|18|22-11-2020 11:17
35|+|return 1;|Program.cs|19|22-11-2020 11:18
36|*|int result = ch…|Program.cs|20|22-11-2020 11:41
37|-|int[] sortedArr…|Program.cs|-6|22-11-2020 11:42
38|-|for(int i = 0;…|Program.cs|-7|22-11-2020 11:43
39|-|result += chan…|Program.cs|-8|22-11-2020 11:44
40|-|change = change…|Program.cs|-9|22-11-2020 11:45
41|-|if(change ==…|Program.cs|-10|22-11-2020 11:46
42|+|foreach(var co…|Program.cs|21|22-11-2020 12:06
43|+|int count = 1…|Program.cs|22|22-11-2020 12:08
44|+|if(count < res…|Program.cs|23|22-11-2020 12:09
45|+|result = count;|Program.cs|24|22-11-2020 12:10
46|+|///<summary>…|Program.cs|13|22-11-2020 12:40
47|+|///<param name…|Program.cs|14|22-11-2020 12:43
48|+|///<param name…|Program.cs|15|22-11-2020 12:44
49|+|///<returns>…|Program.cs|16|22-11-2020 12:45
```

REFERENCES

[1] Almeida, F. "Framework for Software Code Reviews and Inspections in a Classroom Environment", International Journal of Modern Education and Computer Science, 10 (10), pp. 31–39, 2018. https://doi.org/10.5815/ijmecs.2018.10.04

[2] Bacchelli, A., Bird, C. "Expectations, outcomes, and challenges of modern code review", In: 35th International Conference on Software Engineering, San Francisco, USA, 2013, pp. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[3] Sun, Q., Wu, J., Rong, W., Liu, W. "Formative assessment of programming language learning based on peer code review: Implementation and experience report", Tsinghua Science and Technology, 24(4), pp. 423–434, 2019. https://doi.org/10.26599/TST.2018.9010109

[4] De Andrade Gomes, P. H., Garcia, R. E., Spadon, G., Eler, D. M., Olivete, C., Correia, R. C. M. "Teaching software quality via source code inspection tool", In: IEEE Frontiers in Education Conference, Indianapolis, USA, 2017, pp. 1–8. https://doi.org/10.1109/FIE/.2017.8190658

[5] Shynkarenko, V., Zhevago, O. "Visualization of program development process", In: IEEE 14th International Scientific and Technical Conference on Computer Sciences and Information Technologies, Lviv, Ukraine, 2019, pp. 142–145. https://doi.org/10.1109/STC-CSIT.2019.8929774

[6] Shynkarenko, V., Zhevaho, O. "Constructive Modeling of the Software Development Process for Modern Code Review", In: IEEE 15th International Scientific and Technical Conference on Computer Sciences and Information Technologies, Zbarazh, Ukraine, 2020, pp. 392–395. https://doi.org/10.1109/CSIT49958.2020.9322002

[7] Shynkarenko, V., Zhevaho, O. "Development of a Toolkit for Analyzing Software Debugging Processes Using the Constructive Approach", Eastern-European Journal of Enterprise Technologies, 5(2–107), pp. 29–38, 2020. https://doi.org/10.15587/1729-4061.2020.215090

[8] Janke, M., Mader, P. "Mining Code Change Patterns from Version Control Commits", In: IEEE Transactions on Software Engineering, 2020. https://doi.org/10.1109/TSE.2020.3004892

[9] Gousios, G., Spinellis, D. "Mining software engineering data from GitHub", In: IEEE/ACM 39th International Conference on Software Engineering Companion, Buenos Aires, Argentina, 2017, pp. 501–502. https://doi.org/10.1109/ICSE-C.2017.164

[10] Guerrero-Higueras, Á. M., DeCastro-García, N., Conde, M., Matellán, V. "Predictive models of academic success: A case study with version control systems", In: Proceedings of the 6th International Conference on Technological Ecosystems for Enhancing Multiculturality, New York, USA, 2018, pp. 306–312. https://doi.org/10.1145/3284179.3284235

[11] Krusche, S., Seitz, A. "ArTEMiS: An automatic assessment management system for interactive learning", In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, New York, USA, 2018, pp. 284–289. https://doi.org/10.1145/3159450.3159602

[12] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., Dig, D. "Is it dangerous to use version control histories to study source code evolution?", In: European Conference on Object-Oriented Programming, Lecture Notes in

Computer Science, Springer, Berlin, Heidelberg, Germany, 2012, pp. 79–103. https://doi.org/10.1007/978-3-642-31057-7_5
[13] Breuker, D. M., Derriks, J., Brunekreef, J. "Measuring static quality of student code", In: Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science, New York, USA, 2011, pp. 13–17. https://doi.org/10.1145/1999747.1999754
[14] Keuning, H., Heeren, B., Jeuring, J. "Code quality issues in student programs", In: Annual Conference on Innovation and Technology in Computer Science Education, New York, USA, 2017, pp. 110–115. https://doi.org/10.1145/3059009.3059061
[15] Fonseca, N. G., Macedo, L., Mendes, A. J. "CodeInsights: Monitoring programming students' progress", In: Proceedings of the 17th International Conference on Computer Systems and Technologies, New York, USA, 2016, pp. 375–382. https://doi.org/10.1145/2983468.2983492
[16] Snipes, W., Murphy-Hill, E., Fritz, T., Vakilian, M., Damevski, K., Nair, A. R., Shepherd, D. "A Practical Guide to Analyzing IDE Usage Data", In: The Art and Science of Analyzing Software Data, Morgan Kaufmann, Boston, USA, 2015, pp. 85–138. https://doi.org/10.1016/B978-0-12-411519-4.00005-7
[17] Damevski, K., Shepherd, D. C., Schneider, J., Pollock, L. "Mining Sequences of Developer Interactions in Visual Studio for Usage Smells", IEEE Transactions on Software Engineering, 43(4), pp. 359–371, 2017. https://doi.org/10.1109/TSE.2016.2592905
[18] Snipes, W., Augustine, V., Nair, A. R., Murphy-Hill, E. "Towards recognizing and rewarding efficient developer work patterns", In: 35th International Conference on Software Engineering, San Francisco. USA, 2013, pp. 1277–1280. https://doi.org/10.1109/ICSE.2013.6606697
[19] Snipes, W., Nair, A. R., Murphy-Hill, E. "Experiences gamifying developer adoption of practices and tools", In: 36th International Conference on Software Engineering, New York, USA, 2014, pp. 105–114. https://doi.org/10.1145/2591062.2591171
[20] Amann, S., Proksch, S., Nadi, S., Mezini, M. "A Study of Visual Studio Usage in Practice", In: 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, Japan, 2016, pp. 124–134. https://doi.org/10.1109/saner.2016.39
[21] Amann, S., Proksch, S., Nadi, S.: FeedBaG "An interaction tracker for Visual Studio", In: IEEE 24th International Conference on Program Comprehension, Austin, USA, 2016, pp. 1–3. https://doi.org/10.1109/ICPC.2016.7503741
[22] Wang, Y., Li, H., Feng, Y., Jiang, Y., Liu, Y. "Assessment of programming language learning based on peer code review model: Implementation and experience report", Computers and Education, 59(2), pp. 412–422, 2012. https://doi.org/10.1016/j.compedu.2012.01.007
[23] Pon-Barry, H., Packard, B. W. L., St. John, A. "Expanding capacity and promoting inclusion in introductory computer science: a focus on near-peer mentor preparation and code review", Computer Science Education, 27(1), pp. 54–77, 2017. https://doi.org/10.1080/08993408.2017.1333270
[24] García-Borgoñón, L., Barcelona, M. A., García-García, J. A., Alba, M., Escalona, M. J. "Software process modeling languages: A systematic literature review", Information and Software Technology, 56(2), pp. 103–116, 2014. https://doi.org/10.1016/j.infsof.2013.10.001
[25] Shynkarenko, V. I., Ilman, V. M. "Constructive-Synthesizing Structures and Their Grammatical Interpretations. i. Generalized Formal Constructive-Synthesizing Structure", Cybernetics and Systems Analysis, 50(5), pp. 655–662, 2014. https://doi.org/10.1007/s10559-014-9655-z
[26] Shynkarenko, V. I., Ilman, V. M. "Constructive-Synthesizing Structures and Their Grammatical Interpretations. II. Refining Transformations", Cybernetics and Systems Analysis, 50(6), pp. 829–841, 2014. https://doi.org/10.1007/s10559- 014-9674-9
[27] Shynkarenko, V. I., Ilman, V. M., Skalozub, V. V. "Structural models of algorithms in problems of applied programming. I. Formal algorithmic structures", Cybernetics and Systems Analysis, 45(3), pp. 329–339, 2009. https://doi.org/10.1007/s10559-009-9118-0
[28] Shynkarenko, V. I., Ilman, V. M., Skalozub, V. V. "Structural models of algorithms in problems of applied programming. II. Structural-algorithmic approach to software simulation", Cybernetics and Systems Analysis, 45(4), pp. 544–550, 2009. https://doi.org/10.1007/s10559-009-9122-4
[29] Van Der Aalst, W. "Process mining: Overview and opportunities", ACM Transactions on Management Information Systems, 3(2), pp. 1–17, 2012. https://doi.org/10.1145/2229156.2229157
[30] Van Der Aalst, W. et.al. "Process mining manifesto", In: Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg, Germany, pp. 169–194, 2012. https://doi.org/10.1007/978-3-642-28108-2_19
[31] Rubin, V. A., Mitsyuk, A. A., Lomazova, I. A., Van Der Aalst, W. M. P. "Process mining can be applied to software too!" In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, New York, USA, 2014, pp. 1–8. https://doi.org/10.1145/2652524.2652583
[32] Sebu, M. L., Ciocarlie, H. "Applied process mining in software development", In: 9th IEEE International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 2014, pp. 55–60. https://doi.org/10.1109/SACI.2014.6840098
[33] Ardimento, P., Bernardi, M. L., Cimitile, M., Maggi, F. M. "Evaluating coding behavior in software development processes: A process mining approach", In: IEEE/ACM International Conference on Software and System Processes, Montreal, Canada, 2019, pp. 84–93. https://doi.org/10.1109/ICSSP.2019.00020
[34] Skalozub, V., Ilman, V., Shynkarenko, V. "Development of ontological support of constructive-synthesizing modeling of information systems", Eastern-European Journal of Enterprise Technologies, 6(4–90), pp. 58–69, 2017. https://doi.org/10.15587/1729-4061.2017.119497
[35] Verbeek, H. M. W., Buijs, J. C. A. M., Van Dongen, B. F., Van Der Aalst, W. M. P. "XES, XESame, and ProM 6", In: Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg, Germany, pp. 60–75, 2011. https://doi.org/10.1007/978-3-642-17722-4_5
[36] Van der Aalst, W. "Process mining: Data science in action", Springer, Berlin, Heidelberg, Germany, pp. 1–467, 2016. https://doi.org/10.1007/978-3-662-49851-4
[37] Leemans, S. J. J., Poppe, E., Wynn, M. T. "Directly Follows-Based Process Mining: Exploration & a Case Study", In: International Conference on Process Mining, Aachen, Germany, 2019, pp. 25–32. https://doi.org/10.1109/ICPM.2019.00015

Viktor Shynkarenko is Full Professor in Computer and Information Technologies Department of Dnipro National University of Railway Transport named after academician V. Lazaryan, Dnipro, Ukraine. His current research interests are constructive-synthesizing modeling, software engineering education.

Oleksandr Zhevaho is Ph. D. student in Computer and Information Technologies Department of Dnipro National University of Railway Transport named after academician V. Lazaryan, Dnipro, Ukraine. His current research interests are analysis of software development process and Process Mining.