Міністерство освіти і науки України
Дніпровський національний університет залізничного транспорту
імені академіка В. Лазаряна

Прийнято до спец. вченої ради ДФ08.820.009 Голова спец. ради Косолапов А.А.

Кваліфікаційна наукова
праця на правах рукопису

Жеваго Олександра Олександровича

УДК 004.4:004.9

ДИСЕРТАЦІЯ

МОДЕЛЮВАННЯ І АНАЛІЗ ПРОЦЕСІВ РОЗРОБКИ ТА НАЛАГОДЖЕННЯ КОМП'ЮТЕРНИХ ПРОГРАМ

122 – комп'ютерні науки
12 – інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

О. О. Жеваго

Науковий керівник Шинкаренко Віктор Іванович
доктор технічних наук, професор

Дніпро – 2021

АНОТАЦІЯ

Жеваго О. О. Моделювання і аналіз процесів розробки та налагодження комп'ютерних програм. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки». – Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна, Дніпро, 2021.

Зміст анотації

Дисертаційна робота присвячена дослідженню і вирішенню актуальної науково-практичної задачі аналізу процесів розробки та налагодження комп'ютерних програм, для підвищення якості навчання студентів програмуванню. Розуміння того, як студенти розробляють та налагоджують програмне забезпечення, і проблем, з якими вони стикаються, має великий потенціал для підвищення якості навчання програмування. А, як відомо, якість програмного забезпечення безпосередньо пов'язана з якістю відповідного процесу розробки. Стандартний підхід до оцінювання, який враховує тільки результат, не відповідає сучасним вимогам до навчання програмуванню.

В основі дослідження лежить припущення про те, що інструменти, які збирають дані про використання середовища розробки, дають більш детальне розуміння роботи розробників. Аналізуючи, як програмісти використовують середовище розробки, можна виявити закономірності поведінки та визначити проблеми, з якими вони стикаються.

У першому розділі виконано огляд та аналіз наявних підходів до отримання інформації щодо процесів розробки та налагодження програмного забезпечення. Показано, що багатогранні дослідження не ґрунтуються на моделях процесів і не дають можливості вивчення процесу конкретного розробника.

Встановлено на основі існуючих досліджень, що постійний моніторинг призводить до значного поліпшення успішності студентів.

Продемонстровано, що високий рівень практичної підготовки з програмування передбачає, що розробник володіє ефективним стилем розробки програм, що включає:

– застосування методу покрокової деталізації;

– навички декомпозиції;

– відповідність тексту програми стандартам програмування.

Дослідження в області навчання показали, що індивідуальний зворотний зв’язок допомагає студентам вчитися. Однак дати такий зворотний зв’язок студентам, які навчаються програмування, досить складно.

З’ясовано, що дослідження в області розробки програмного забезпечення зазвичай ґрунтуються на даних, отриманих з систем контролю версій, але не з середовища розробки. Система контролю версій не може оцінити внесок студента в кінцевий результат, оскільки не надає інформацію про процес розробки. Вони лише фіксують історію змін між комітами та не відображають реальну еволюцію коду.

На відміну від цього в дисертації інформація щодо процесів розробки та налагодження програм отримується безпосередньо з середовища розробки.

У другому розділі за допомогою інструментів конструктивно-продукційного моделювання формалізовано процеси розробки та налагодження програмного забезпечення.

Показано універсальність і перспективність використання конструктивно-продукційного моделювання для вирішення завдань різних предметних областей.

Розроблено конструктори для:

– формування історії розробки програмного забезпечення при використанні середовища розробки для подальшого автоматичного аналізу;

– класифікації історії розробки програми по типах змін і побудови файлу журналу подій з метою вивчення процесу розробки студентами комп’ютерних програм;

– формування журналу подій при налагодженні програм в середовищі розробки.

Визначено зв’язок конструктивно-продукційного підходу та використання методів Process Mining з об’єктно-орієнтованими моделями.

На основі конструктивних моделей створено доповнення до середовища розробки Visual Studio для реалізації можливості відслідковування роботи кожного студента як на лабораторних заняттях, так і під час самостійної роботи, в якому всі дії по розробці та налагодженню фіксуються в журналах подій.

Використано методи Process Mining для аналізу і побудови моделей процесів розробки та налагодження програм. Метод Discovery – для формування моделей процесів на основі файлів журналів подій. Метод Conformance Checking – для порівняння різних виконань процесів і виявлення поведінкових схожостей та відмінностей.

У третьому розділі приведені результати експериментальних досліджень. Перевірено та підтверджено можливість застосування сформованих конструктивно-продукційних моделей та розроблених на їх основі інструментальних засобів для відстеження процесів розробки та налагодження програмного забезпечення.

Виконано експеримент у вигляді олімпіади з налагодження, під час якого за допомогою розроблених інструментів відстежувалися та фіксувалися дії учасників в середовищі розробки.

Для експерименту розроблені програми з засіяними типовими логічними помилками, які визначені на основі досліджень про найбільш поширені помилки початківців. Кожного разу логічна помилка засіяна у одному рядку коду програми. Всі програми синтаксично коректні, але виконання кожної з них призводить до некоректного результату. Щоб допомогти виявити всі помилки, студентам надавався тест з вхідними та вихідними даними, що відповідають специфікації.

У експерименті взяв участь 41 студент з першого по п’ятий курси. Всі вони мали попередній досвід розробки на С-подібних мовах програмування і використання середовища розробки Visual Studio. Всього студентам треба було виконати п’ятнадцять завдань за 4 години.

Під час експерименту сформовано 487 файлів журналів подій для 41 учасника. Загалом журнали подій складаються з 2415 сесій налагодження та 16536 подій.

По результатам експерименту проведено класифікацію учасників на основі кількості правильно виконаних завдань як "High", "Middle", "Low". Студенти відносяться до класу "Low", якщо вони виконали менше половини завдань, таких – 9. Студенти з "Middle" класу виконали більше половини завдань, але не всі, таких – 23. Студенти з "High" класу виконали всі завдання, таких – 9.

Результати показали навички студентів по налагодженню програм з типовими логічними помилками. Вдалося виявити проблеми, які є більш складними для локалізації та виправлення.

Для розуміння поведінки учасників з кожного класу засобами Process Mining сформовано відповідні моделі процесів на основі журналів подій.

На основі отриманих моделей процесів налагодження та найбільш часто повторювальних сесій вдалося виявити 4 шаблони поведінки учасників експерименту.

За допомогою методу Conformance Checking засобів Process Mining проведено порівняння процесів різних учасників з метою виявлення особливостей поведінки.

У четвертому розділі продемонстровано інструменти для автоматичного моніторингу та візуалізації процесів розробки та налагодження програмного забезпечення.

Розроблені інструменти дозволяють:

- фіксувати взаємодію розробників із середовищем розробки;
- формувати журнали подій про процеси розробки та налагодження;
- візуалізувати процес розробки у вигляді 3D зображення з можливістю масштабування та обертання.

Для розширення середовища розробки Visual Studio можливістю відстежувати увесь процес розробки та налагодження програм розроблено VSPackage за допомогою Microsoft Visual Studio SDK.

На рівні процесу фіксуються події з базовою інформацією, такою як тип події (наприклад, зміна або збереження коду) та час. На контекстному рівні зберігаються конкретні дані залежно від типу події. Наприклад, події збереження зберігають ім’я файлу, який було збережено, та спосіб, за допомогою якого було введено дію збереження (наприклад, за допомогою гарячих клавіш чи вибору пункту меню).

Для збереження та аналізу даних, щодо процесів розробки та налагодження розроблено об’єктно-орієнтовану модель, яка відповідає терміналам з їх атрибутами і базується на моделі подій в середовищі розробки.

Наукова новизна отриманих результатів полягає в тому що в роботі вперше:

– виконано формалізацію процесів розробки та налагодження програмного забезпечення засобами конструктивно-продукційного моделювання, що на відміну від існуючих дозволяє розглядати ці процеси як послідовність елементарних дій за продукційними правилами, формалізувати процеси формування журналів подій та візуалізувати ці процеси;

– розроблені моделі процесів розробки та налагодження програм засобами Process Mining, що на відміну від існуючих дають можливість автоматизувати аналіз цих процесів;

– розроблені інструменти для збору даних щодо дій програміста у процесі розробки та налагодження програмного забезпечення з середовища розробки Visual Studio.

Дисертація є частиною науково-дослідної роботи «Конструктивно-продукційне моделювання в задачах розробки програмного забезпечення» (2021 р. № держреєстрації 0121U109167), яка виконана на кафедрі «Комп’ютерних інформаційних технологій» Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна.

Ключові слова: конструктивно-продукційне моделювання, конструктори, Process Mining, розробка програмного забезпечення, налагодження, огляд коду, навчання, інженерія програмного забезпечення.

ABSTRACT

Zhevaho O. O. Modeling and analysis of the process of developing and debugging computer programs. – Qualifying scientific work on the rights of the manuscript.

Thesis for obtaining the Ph. D. degree in specialty 122 – Computer Science. – Dnipro National University of Railway Transport named after Academician V. Lazaryan, Dnipro, 2021.

Abstract content

The dissertation is devoted to research and solving the urgent scientific and practical task of analyzing the processes of developing and debugging computer programs, in order to improve the quality of students' programming education. Understanding how students develop and debug programs and what problems they encounter has great potential to improve the quality of programming instruction. And, as we know, the quality of the software is directly related to the quality of the corresponding development process. The standard approach to assessment, which considers only the result, does not meet modern requirements for teaching programming.

The study is based on the assumption that tools that collect data on development environment usage provide a more detailed understanding of developers' work. By analyzing how programmers use the development environment, patterns of behavior can be identified and the problems they encounter can be identified.

The first chapter reviews and analyzes existing approaches to obtaining information about software development and debugging processes. It is shown that multifaceted studies are not based on process models and do not provide an opportunity to study the process of a particular developer.

Based on existing research, it has been found that continuous monitoring leads to significant improvements in student achievement.

It is shown that a high level of practical training in the field of programming implies that the developer has an effective style of program development, including:

– application of the step-by-step method;

– decomposition skills;

– compliance of the program text with programming standards.

Research in the field of learning has shown that individualized feedback helps students learn. However, giving such feedback to programming students is quite difficult.

It has been found that research in software development is usually based on data from version control systems but not from the development environment. Version control systems cannot assess the student's contribution to the final result because they do not provide information about the development process. They only record the history of changes between commits and do not reflect the actual evolution of the code.

In contrast, in this dissertation, information about the processes of writing and debugging programs is obtained directly from the integrated development environment.

The second chapter formalizes the processes of software development and debugging by using constructive-synthesizing modeling tools.

The versatility and perspectivity of the use of constructive-synthesizing modeling to solve problems in various subject areas is shown.

Constructors have been developed for:

– forming the history of software development using the development environment for further automatic analysis;

– classifying the history of writing program text by type of change and constructing a log file in order to study the process of computer program development by students;

– formation of a log of events during the debugging of programs in the development environment.

The connection between the constructive-synthesizing modeling and the use of Process Mining methods with object-oriented models is determined.

Based on constructive models, extension to the Visual Studio development environment were created to implement the ability to track each student's work both in

labs and during independent work, where all development and debugging activities are recorded in event logs.

Process Mining methods are used to analyze and build process models for software development and debugging. The Discovery method is used to build process models based on event log files. The Conformance Checking method is used for comparing different processes and identifying behavioral similarities and differences.

The third chapter presents the results of experimental research. The possibility of using the generated structural models and the tools developed on their basis for tracking the software development and customization processes is tested and confirmed.

An experiment in the form of a debugging olympiad was conducted in which participants' actions in the development environment were tracked and recorded using the developed tools.

For the experiment, programs were designed with seeded typical logical errors, which were determined based on a study of the most common beginner's errors. Each time a logical error is seeded in one line of program code. All of the programs are syntactically correct, but the execution of each leads to the wrong result. To help identify all errors, students were given a test with inputs and outputs that matched the specification.

The experiment involved 41 students from the first to the fifth year. All of them had experience in C-like programming languages and using Visual Studio development environment. In total, students had to complete fifteen tasks in four hours.

During the experiment, 487 event log files were generated for 41 participants. In total, the event logs consist of 2415 debugging sessions and 16536 events.

According to the results of the experiment, the participants were classified on the basis of the number of correctly performed tasks as "High", "Middle", "Low". Students belong to the class "Low" if they have completed less than half of the tasks, such – 9. Students from the "Middle" class have completed more than half of the tasks, but not all, such – 23. Students from the "High" class have completed all tasks, such – 9.

The results showed the skills of students to debug programs with typical logical errors. We were able to identify problems that are more difficult to localize and fix.

To understand the behavior of participants from each class by means of Process Mining, appropriate process models are formed on the basis of event logs.

Based on the obtained models of debugging processes and the most frequently repeated sessions, it was possible to identify 4 patterns of behavior of the participants of the experiment.

Using the Conformance Checking method of Process Mining, the processes of different participants were compared in order to identify the characteristics of behavior.

The fourth chapter demonstrates tools for automatic monitoring and visualization of software writing and debugging processes.

Developed tools allow:

– to record the interaction of developers with the development environment;

– to form event logs about development and debugging processes;

– visualize the development process in the form of a 3D image with the ability to scale and rotate.

To extend the Visual Studio development environment with the ability to track the entire process of program development and debugging, VSPackage was developed using the Microsoft Visual Studio SDK.

At the process level, events are logged with basic information such as event type (e.g., code change or save) and time. At the context level, specific data is stored depending on the type of event. For example, save events store the name of the file that was saved and the way in which the save action was entered (e.g., by using hotkeys or selecting a menu item).

To store and analyze data about the development and debugging processes, an object-oriented model was developed that corresponds to terminals with their attributes and is based on the development environment event model.

The scientific novelty of the results is that for the first time in the work:

– formalization of software development and debugging processes by means of constructive-synthesizing modeling, which, unlike the existing ones, allows to consider these processes as a sequence of elementary actions according to production rules, to formalize the processes of event logs formation and visualize these processes;

– developed models of processes of development and debugging of programs by means of Process Mining that unlike existing give the chance to automate the analysis of these processes;

– developed tools for collecting data on the actions of the programmer in the process of developing and debugging software from the Visual Studio development environment.

The dissertation is part of the research work «Modeling in software development tasks» (2021 year, registration number 0121U109167), which performed at the «Department of Computer Information Technologies», Dnipro National University of Railway Transport named after Academician V. Lazaryan.

Keywords: constructive-synthesizing modeling, constructors, Process Mining, software development process, debugging, code review, education, software engineering.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Праці включені до міждународних наукометричних баз (МНБД) Scopus та Web of Science:

1. Shynkarenko, V., Zhevaho, O. Development of a toolkit for analyzing software debugging processes using the constructive approach // Eastern-European Journal of Enterprise Technologies. – 2020. – pp. 29–38. https://doi.org/10.15587/1729-4061.2020.215090. *Стаття у фаховому журналі, що індексується у МНБД Scopus та має Q3 за даними Scimago Journal & Country Rank, зараховується як дві публікації.*

2. Shinkarenko V. I., Zhevago O. O. Generating university course timetable using constructive modeling // Radio Electronics, Computer Science, Control. – 2019. – pp. 152–162. https://doi.org/10.15588/1607-3274-2019-3-17. *Стаття у фаховому журналі, що індексується у МНБД Web of Science.*

3. Shynkarenko V., Zhevaho O. Constructive modeling of the software development process for modern code review // IEEE 15th International Conference on Computer Sciences and Information Technologies. – 2020. – pp. 392–395. https://doi.org/10.1109/CSIT49958.2020.9322002. *Матеріали конференції, індексується у МНБД Scopus.*

4. Shynkarenko, V., Zhevago, O. Visualization of program development process // IEEE 14th International Conference on Computer Sciences and Information Technologies. – 2019. – pp. 142–145. https://doi.org/10.1109/stc-csit.2019.8929774. *Матеріали конференції, індексується у МНБД Scopus.*

Праці у фахових виданнях затверджених МОН України:

5. Zhevaho O. O. An overview of tools for collecting data on software development and debugging processes from integrated development environments // Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту – 2021 – С. 24–37. https://doi.org/10.15802/stp2021/242042.

6. Шинкаренко В. І., Жеваго О. О. Експериментальні дослідження процесів налагодження комп'ютерних програм студентами з використанням Process Mining // Вісник Херсонського національного технічного університету – 2021 – С. 83–98. https://doi.org/10.35546/kntu2078-4481.2021.3.10.

Матеріали наукових конференцій:

7. Жеваго О. О., Шинкаренко В.І. Дослідження методів відстеження процесів розробки та відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XI Міжнародної науково-практичної конференції, м. Дніпро. – 2017. – С. 103.

8. Жеваго О. О., Шинкаренко В.І. Дослідження стилю відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIII Міжнародної науково-практичної конференції, м. Дніпро. – 2019. – С. 73.

9. Жеваго О. О., Шинкаренко В.І. Конструктивне моделювання та аналіз процесів розробки і відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIV Міжнародної науково-практичної конференції, м. Дніпро. – 2020. – С. 129.

10. Жеваго О. О., Шинкаренко В.І. Аналіз та вдосконалення навчального процесу з програмної інженерії з використанням методів Process Mining // Сучасні тенденції розвитку інформаційних систем і телекомунікаційних технологій: тези III Міжнародної науково-практичної конференції, м. Київ. – 2021. – С. 73–74.

11. Жеваго О. О., Шинкаренко В. І. Виявлення навичок відлагодження програм методом підсіву помилок // Інформаційні Технології в Металургії та Машинобудуванні: тези Міжнародної науково-технічної конференції ІТММ 2021, м. Дніпро. – 2021. – С. 339–340.

12. Жеваго О. О., Шинкаренко В. І. Составление расписания занятий на основе конструктивного моделирования // Проблеми математичного моделювання: Матеріали Всеукраїнської науково-методичної конференції, м. Кам’янське. – 2018. – С. 59–63.

ЗМІСТ

ВСТУП

Актуальність теми. Однією з актуальних проблем освітнього процесу в області інформаційних технологій є низький рівень практичних навичок студентів в розробці програмного забезпечення. Програмування вимагає багатьох компетенцій, і їх навчання є одним з основних завдань освіти в галузі комп'ютерних наук. Зазвичай при навчанні програмуванню викладач оцінює результати роботи та не має можливості відстеження та оцінювання процесів розробки та налагодження програм. Це зумовлює можливості отримання хибних навичок, що надалі сповільнює розвиток професійного рівня майбутнього фахівця. Така важлива частина, як процес розробки та налагодження програми, залишається без належної уваги.

Аналіз текстів програм викладачем викликає ряд проблем. Студенти не завжди дотримуються правил оформлення, приносять на перевірку неякісні тексти. На викладача лягає додаткове навантаження по верифікації коду програм. Крім того, перед викладачем стоїть завдання визначити особистий внесок студента в представлений результат. Цей результат може бути отриманий студентом як самостійно, так і шляхом запозичення. Особливо небезпечно повне запозичення, яке свідчить про відсутність особистого вкладу в роботу.

В процесі навчання основам програмування важливо виявляти проблеми та допомагати в їх усуненні, контролювати самостійність і якість виконання завдань, фіксувати труднощі в роботі кожного студента, виявляти приховані можливості в розробці програмного коду. Для цього необхідний постійний контроль за студентами з боку викладача. Однак під час групових занять, а особливо під час самостійної роботи, неможливо встежити за індивідуальними особливостями процесу програмування кожного студента. Досягнення максимальних результатів в навчанні й набуття навичок неможливо без механізму фіксації деталей процесу.

У зв'язку з цим актуальними є дослідження, спрямовані на формування нових підходів і розробку інструментів для підвищення якості навчання

студентів програмуванню. Розуміння того, як студенти розробляють та налагоджують програмне забезпечення, і проблем, з якими вони стикаються, має великий потенціал для підвищення якості навчання програмування. А, як відомо, якість програмного забезпечення безпосередньо пов’язана з якістю відповідного процесу розробки. Пропонується використовувати не лише код програми, а й аналіз процесу її розробки та налагодження для оцінки роботи студента.

Дослідження процесу розробки програмного забезпечення представлені в роботах N. G. Fonseca, M. Mader, V. Damevski, E. Amann, G. C. Murphy, P. Ardimento, J. Caldeira та ін.

В області процесу налагодження дослідження представлені в роботах B. S. Alqadi, J. I. Maletic, M. Perscheid, A. Zeller, M. Ahmadzadeh, C. Higgins, F. Petrillo, A. Yamashita, Y.-T. Lin та ін.

Дослідження в області розробки програмного забезпечення зазвичай ґрунтуються на даних, отриманих з систем контролю версій, але не з середовища розробки. Система контролю версій не може оцінити внесок студента в кінцевий результат, оскільки не надає інформацію про процес розробки. Вони лише фіксують історію змін між комітами та не відображають реальну еволюцію коду.

На відміну від цього в дисертації інформація щодо процесів розробки та налагодження програм отримується безпосередньо з середовища розробки.

Зв’язок роботи з науковими програмами, темами, планами. Дисертаційна робота виконана відповідно до:

– Закону України № 2623-III «Про пріоритетні напрями розвитку науки і техніки» редакція від 20.02.2021 р., зокрема за напрямом «Інформаційні та комунікаційні технології»;

– Закону України № 3715-VI «Про пріоритетні напрями інноваційної діяльності в Україні» редакція від 05.12.2012 р., зокрема за напрямом «Розвиток сучасних інформаційних, комунікаційних технологій, робототехніки».

Наукові дослідження, викладені в дисертації, виконані згідно з напрямом наукової роботи кафедри «Комп'ютерні інформаційні технології» Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна, а саме є частиною науково-дослідної роботи «Конструктивно-продукційне моделювання в задачах розробки програмного забезпечення» (2021 р. № держреєстрації 0121U109167), у якій дисертант приймає участь у якості виконавця.

Мета та задачі дослідження. Метою дисертації є вдосконалення процесу навчання студентів основам програмування, шляхом підвищення контролю за процесом виконання робіт та внаслідок підтримки індивідуального підходу до навчання.

Для досягнення зазначеної мети були поставлені та вирішені наступні задачі:

– виконано формалізацію процесів збору даних про використання середовища розробки засобами конструктивно-продукційного моделювання;

– розроблено засоби для фіксації подій з середовища розробки;

– сформовано моделі процесів розробки та налагодження за допомогою методів Process Mining;

– розроблено засоби відстеження, моделювання та аналізу процесів розробки та налагодження програм.

Об'єктом дослідження є процес розробки та налагодження програм.

Предметом дослідження моделі та методи розробки та налагодження програм.

Методи дослідження. Для вирішення поставлених задач було використано:

– конструктивно-продукційне моделювання;

– методи та засоби Process Mining;

– технологія об'єктно-орієнтованого проєктування, при розробці програмної реалізації отриманих моделей;

– метод підсіву помилок для виявлення навичок налагодження.

Наукова новизна отриманих результатів. В роботі **вперше**:

– виконано формалізацію процесів розробки та налагодження програмного забезпечення засобами конструктивно-продукційного моделювання, що на відміну від існуючих дозволяє розглядати ці процеси як послідовність елементарних дій за продукційними правилами, формалізувати процеси формування журналів подій та візуалізувати ці процеси;

– розроблені моделі процесів розробки та налагодження програм засобами Process Mining, що на відміну від існуючих дають можливість автоматизувати аналіз цих процесів;

– розроблені інструменти для збору даних щодо дій програміста у процесі розробки та налагодження програмного забезпечення з середовища розробки Visual Studio.

Удосконалено:

– підхід до оцінювання робіт з програмування, який враховує не тільки результат, а й процес його досягнення. Для цього представлено інноваційний метод використання огляду коду і процесу розробки та налагодження програмного забезпечення при навчанні навичкам програмування.

Отримали подальший розвиток:

– засоби конструктивно-продукційного моделювання, зокрема зв’язок конструктивно-продукційного підходу та використання методів Process Mining з об’єктно-орієнтованими моделями.

Практичне значення отриманих результатів. Розроблений інструментарій, на основі конструктивних моделей, які дають формалізоване представлення процесів розробки та налагодження, дозволяє використовувати запропоновані моделі та методи в технології навчання студентів основам програмування, особливо на початкових курсах.

Тому, практична частина використання досягнутого наукового результату полягає у можливості вдосконалення викладачами планів з навчання основам програмування, а студентами вдосконалення свого стилю роботи.

Практичне значення результатів підтверджується використанням при проведені та аналізі результатів олімпіади по налагодженню в Дніпровському національному університеті залізничного транспорту імені академіка В. Лазаряна.

Особистий внесок здобувача. Результати дисертаційної роботи опубліковано в роботах у співавторстві:

[106] – розробка засобів візуалізації процесу розробки програмного забезпечення;

[107] – дослідження методів відстеження процесів розробки та налагодження програм;

[108] – дослідження стилю налагодження програм;

[109] – виконано формалізацію процесу розробки програми засобами конструктивно-продукційного моделювання;

[110] – розробка засобів для аналізу процессу налагодження програмного забепечення;

[111] – проведення експерименту з дослідження процесів налагодження;

[112] – виявлення навичок налагодження програм методом підсіву помилок;

[113] – використання методів Process Mining для вдосконалення навчального процесу з програмної інженерії;

[122] – формалізація процессу формування розкладу занять засобами конструктивно-продукційного моделювання;

[123] – розробка програмних засобів на основі конструктивних моделей для формування розкладу занять;

[129] – розробка конструктивних моделей для формалізації процесів розробки і налагодження програм.

В одноосібній статті [114] – огляд та аналіз існуючих засобів для збору даних про процеси розробки та налагодження програмного забезпечення.

Апробація результатів дисертації. Результати дисертаційної роботи представлено на всеукраїнських та міжнародних науково-практичних конференціях: «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті» (Дніпро, 2017, 2019, 2020), «Інформаційні технології в металургії та машинобудуванні» (Дніпро, 2021), «Сучасні тенденції розвитку інформаційних систем і телекомунікаційних технологій» (Київ, 2021), «Computer Sciences and Information Technologies» (Львів, Збараж, 2019, 2020).

Публікації. Результати дисертаційної роботи опубліковано в 12 наукових працях.

У фахових та рекомендованих Міністерством освіти і науки (МОН) України для публікації результатів дисертацій – 2.

У фаховому та рекомендованому МОН України для публікації результатів дисертацій «Eastern-European Journal of Enterprise Technologies», що індексується МНМБ Scopus – 1. Має Q3 за даними Scimago Journal & Country Rank, зараховується як дві публікації.

У фаховому та рекомендованому МОН України для публікації результатів дисертацій «Radio Electronics, Computer Science, Control», що індексується МНМБ Web of Science – 1.

Матеріали міжнародних конференцій, що індексуються МНМБ Scopus – 2.

У тезах доповідей міжнародних та всеукраїнських конференцій – 6.

Структура та обсяг дисертації. Дисертаційна робота складається із вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. Загальний обсяг дисертації становить 124 сторінки, в тому числі 93 сторінки основної частини, 20 рисунків, 6 таблиць, 2 додатки на 3 сторінках та список використаних джерел із 133 найменувань на 14 сторінках.

РОЗДІЛ 1
АНАЛІЗ ПІДХОДІВ ДО ОТРИМАННЯ ІНФОРМАЦІЇ ПРО ПРОЦЕСИ РОЗРОБКИ ТА НАЛАГОДЖЕННЯ ПРОГРАМ

1.1 Процес розробки програмного забезпечення

Мета всіх організацій залучених в розробку програмного забезпечення підвищити якість коду розробників. Розуміння того як розробники пишуть програми, їх налагоджують і на які труднощі натрапляють має дати корисну інформацію для покращення роботи розробників і скоротити розрив у продуктивності [1].

Якість програмного забезпечення безпосередньо пов’язана з якістю відповідного процесу розробки. Одним з найбільш вивчених методів підвищення якості програми є огляд коду [2]. Як правило, такий підхід реалізується в компаніях розробниках програмного забезпечення. Однак їх досвід може бути ефективний в освіті в області комп’ютерних наук.

За останні кілька років були проведені дослідження з ефективного використання огляду коду в навчанні програмуванню [3, 4]. Результати показують, що огляд коду може допомогти студентам поліпшити свої навички розробки програмного забезпечення.

В [5, 6] автори дослідили якість студентських програм. Результати дослідження виявили безліч недоліків якості та відсутність очікуваного підвищення якості коду студентів першого і другого курсів.

Кілька опублікованих досліджень показали, що постійний моніторинг призводить до значного поліпшення успішності студентів [4, 7, 8].

Високий рівень практичної підготовки з програмування передбачає, що розробник володіє ефективним стилем розробки програм, що включає:

- застосування методу покрокової деталізації;
- навички декомпозиції;
- відповідність тексту програми стандартам програмування.

Покрокова деталізація – парадигма розвитку складної програми з простої, поступово додаючи деталі [9, 10]. Принцип покрокового уточнення або деталізації як ключового для систематичного створення програм ввів в практику Ніклаус Вірт, який опублікував докладну статтю з прикладом використання цього принципу в журналі Communications of ACM в 1971 році [11]. Відхилення від методу покрокової деталізації виражається хаотичністю розробки програми, внесенням змін до вже написаних фрагментів тексту, несвоєчасністю додавання коментарів.

Застосування навичок декомпозиції характеризується порядком створення нових функцій, співвідношенням розмірів функцій і наявністю дублювання в тексті програми [12].

Однією з причин спостереження за процесом розробки тексту програми є контроль самостійності виконання роботи. В [13] пропонується вести відеозапис процесу розробки в поєднанні з поясненням всіх виконуваних дій. Але такий підхід досить трудомісткий як для студента, так і для викладача.

На сьогодні єдиний спосіб проаналізувати роботу студента – це безпосередньо стежити за процесом розробки. Однак викладач, не в змозі простежити таким чином за кожним студентом під час роботи, а особливо під час самостійної роботи.

Навчання сам на сам з експертом з програмування – це ефективний спосіб дати студенту індивідуальний зворотний зв’язок. Експерт може приділити час, щоб повністю зрозуміти код студента і вислухати, як студент пояснює хід своїх думок. Потім експерт може розглянути кожну з його помилок і дати зворотний зв’язок з приводу імен змінних, моделей проєктування. Однак індивідуальне навчання не представляється можливим в університетах, де викладачів набагато менше, ніж студентів. У викладачів немає часу, щоб детально вивчити код студента і дати йому детальний зворотний зв’язок.

Дослідження в області навчання показали, що індивідуальний зворотний зв’язок допомагає студентам вчитися [14–16]. Однак дати такий зворотний зв’язок студентам, які навчаються програмування, досить складно.

Дослідження в області розробки програмного забезпечення найчастіше ґрунтуються на даних, отриманих з систем контролю версій [17, 18].

Аналіз змін коду, зафіксованих в системах контролю версій, є найбільш поширеним способом аналізу даних про розробку програмного забезпечення. В останні роки з'явилося кілька повідомлень про успішне використання систем контролю версій на заняттях з програмування [19, 20]. Однак системи контролю версій не можуть оцінити внесок студента в кінцевий результат, оскільки вони не надають інформацію про процес розробки. Системи контролю версій фіксують тільки історію змін між комітами. В [21] показано, що історія з систем контролю версій не показує реальну еволюцію коду.

В [22] надали практичний посібник з використання середовища розробки. Показано, що інструменти для збору даних про використання середовища розробки дозволяють отримати більш детальне уявлення про роботу розробників. Для збору інформації про процес розробки програми в середовищі розробки зазвичай використовуються дані взаємодії [23–26]. Однак більшість цих інструментів відстежують тільки виклики команд під час сеансів кодування, без більш докладних контекстних даних.

Набір даних, що містить понад 600 годин взаємодії програміста з середовищем розробки, з яких понад 26 годин супроводжуються відеозаписами екрану комп'ютера та усними коментарями розробників, був представлений в роботі [27]. Недоліком даного підходу є те, що з отриманих відеозаписів не видобувається ніякої інформації, тому їх аналіз може бути тільки ручним.

В [28] реалізовані інструменти для відстеження взаємодії з середовищем розробки та об'єднання цих даних з більш докладною контекстною інформацією.

Blackbox – це проєкт зі збору даних [29], в рамках якого збираються дані компіляції Java користувачами в середовищі програмування BlueJ, розробленої для програмістів початківців. Blackbox залишається одним з найбільших репозиторіїв даних про програмування, доступних сьогодні дослідникам в області комп'ютерних наук. Він використовувався в численних дослідженнях [30], в тому числі для вивчення діяльності початківців [31], помилок компіляції та

повідомлень про помилки [32–34], а також проблем якості коду в програмах, написаних студентами [35]. Науковці часто використовують дані з цієї системи для розробки кількісних показників для різних аспектів програмування. Більшість використовують дані для вимірювання схильності новачків до синтаксичних або семантичних помилок, а також для вимірювання вміння їх виправляти. В [36] використовували дані про компіляцію користувачів BlueJ, щоб отримати поведінку програмістів початківців. З якими помилками вони зазвичай стикаються, час, який вони зазвичай витрачають на програмування перед компіляцією, і навички налагодження, а також як вони реагують на повідомлення про помилки від середовища розробки.

В [37] розробили розширення до середовища програмування NetBeans для того, щоб фіксувати, як студенти розробляють програмне забезпечення. Система записує події, коли студент зберігає, запускає або тестує код. Розширення використовували для вивчення загальних шляхів розв'язання проблем, які використовують початківці Java програмісти [38].

Hackystat [39] – це проєкт з відкритим кодом Гавайського університету, який забезпечує збір та аналіз даних з процесів програмування в освіті та промисловості. Система відстежує дії розробників і відправляє дані про процес на централізований вебсервер. Розробники можуть увійти на вебсайт, щоб переглянути зібрані дані та запустити аналіз. Конфігурація під назвою Hackystat-UH використовується для збору та аналізу даних з програмної інженерії студентів.

Marmoset [40] – автоматизована система оцінювання, розроблена в Університеті Меріленду. Це розширення до середовища розробки Eclipse для збору студентського коду і зберігання його в репозиторії системи контролю версій Concurrent Versioning System при кожному збереженні файлу. В [41] використовували Marmoset для аналізу поведінки студентів в тестуванні програмного забезпечення, щоб краще зрозуміти проблеми, пов'язані з навчанням розробці на основі тестування.

У ряді досліджень обговорюються способи перехоплення та аналізу взаємодії між розробниками та середовищем розробки. В [42] наведені деякі

статистичні дані про те, як Java розробники використовують середовище розробки Eclipse. В [43] представлений інструмент під назвою DFlow. Він виявляє найбільш трудомісткі та найскладніші дії, аналізуючи взаємодію розробників з середовищем розробки.

Система, яка автоматично відстежує, оцінює процес кодування, дає зворотний зв'язок і оцінку навичок програмування, представлена в [44]. Система розроблена на основі компонент з відкритим кодом, які забезпечують онлайн середовище програмування, яке може відстежувати дії студентів під час виконання завдання з програмування.

Візуалізація процесу розробки, яка дозволяє викладачам швидко та ефективно переглядати всю історію створення коду студента для індивідуального зворотного зв'язку, представлена в [45]. Система візуалізує код студента і спеціально розроблена для більш ефективного зворотного зв'язку, який дають викладачі після коротких навчальних вправ з програмування. Попереднє дослідження користувачів показало, що ця візуалізація дозволяє викладачам розуміти кроки, які зробив студент, перш ніж прийшов до остаточного варіанту коду, і допомагає давати глибший зворотний зв'язок. Але представлена система розроблена лише для невеликих і достатньо простих програм.

Налагодження та програмування йдуть рука об руку, оскільки малоймовірно, що будь-який програміст напише ідеальну програму з першого разу. Проблеми, з якими новачок стикається вперше, швидше за все, будуть виникати знову і знову, в різних середовищах і навіть на різних мовах програмування. Дуже важливо, щоб студент навчився досліджувати помилки.

1.2 Процес налагодження програмного забезпечення

Налагодження – одне з найважливіших, складних і трудомістких завдань при розробці програмного забезпечення. Розробники знають, що налагодження може забрати довгі години роботи, іноді тільки для зміни одного рядка коду.

Стандартний глосарій термінології програмної інженерії IEEE визначає налагодження як операцію з виявлення, локалізації та виправлення помилок в комп'ютерних програмах [46].

Розробники зазвичай витрачають не менше 30% свого часу на налагодження і використовують для цього середовище розробки [47].

Коли відбувається збій, розробники повинні виконати три основні дії.

По-перше, локалізувати несправність, яка полягає у визначенні операторів, відповідальних за збій. По-друге, зрозуміти причини збою. По-третє, виправити помилку – це визначення того, як модифікувати код для її усунення.

Розуміння причини збою зазвичай охоплює складні завдання, такі як перегляд залежностей програми та виконання частини програмного забезпечення кілька разів з різними вхідними даними.

Успішне налагодження зазвичай залежить від правильного розуміння синтаксису і семантики програми. Рівень розуміння програми та попередній досвід роботи з такими помилками – два ключові аспекти, які відрізняють новачка від досвідченого налагоджувача [48, 49]. Налагодження може значно поліпшити розуміння програми, оскільки воно вимагає, щоб розробники могли читати та розуміти код.

В роботі [50] розробники були опитані, щоб зрозуміти, як вони досліджують програми. Результат показав, що програмісти витрачають велику частину свого часу на читання і розуміння вихідного коду. Найбільш ефективним інструментом для розуміння коду, на їхню думку, є багаторазовий запуск програми за допомогою налагоджувача. Це підтверджує гіпотезу про те, що налагодження використовується не тільки для локалізації помилок, але і для розуміння коду.

Розвиток навичок ефективного налагодження особливо важливо для новачків: вони все ще вивчають синтаксис і семантику мови програмування, більш схильні до створення неправильного коду, мають обмежені навички розуміння програм і ефективного налагодження, що часто призводить до труднощів в осмисленні та усуненні помилок [51, 52].

Розроблена лекція для допомоги студентам у налагодженні за допомогою вправ [53]. Недоліком є те, що виконання запропонованих вправ вимагає досить високого рівня навичок, і не всі учасники можуть вивчити процес налагодження. Крім того, навчання на основі трьох вправ не може показати реальні навички. Експеримент по оцінці ефективності системи навчання налагодження не проводився.

У [54] вивчені вісім найбільш поширених помилок початківців. Написано вісім коротких програм, кожна з яких представляла один тип помилки, а учасники експерименту повинні були зіставити програму з типом виявленої помилки. В експерименті брали участь 59 студентів другого курсу Кентського державного університету. Експеримент був розроблений таким чином, щоб відстежувати порядок, в якому учасник виправляв помилки, а також час, витрачений на кожну з помилок. Отримані результати дозволяють виявити складні для налагодження помилки. Однак вони не дають можливості оцінити рівень навичок студентів і не визначають, які методи та інструменти налагодження були використані.

В [55] автори дослідження зібрали та проаналізували 450 помилок, які студенти не можуть вирішити та за якими вони звертаються за допомогою до викладачів. Для збору даних було розроблено вебдодаток, в якому студенти дають короткий опис завдання і проблем, з якими вони зіткнулися. Після цього викладач переглядає код разом зі студентом і допомагає йому розібратися в проблемі. Найбільш поширені типи помилок обговорюються на заняттях. Головний результат дослідження полягає в тому, що приблизно 22% проблем пов’язані з навичками вирішення завдань, а решта – з комбінацією логічних і синтаксичних помилок. Розуміння помилок, з якими стикаються студенти, дозволяє навчити їх необхідним навичкам налагодження. Викладачі використовують ці дані для постійного вдосконалення навчальної програми. Інформація про ефективність такого підходу не наводиться. Недоліком цього підходу є те, що студенти повинні самостійно відвідати сайт і описати проблеми, з якими вони стикаються, замість того, щоб збирати цю інформацію безпосередньо з середовища розробки.

Щоб скоротити час, що витрачається на пошук і усунення дефекту, що викликає збій програми, може бути корисним систематичний підхід до налагодження. Приклад такого підходу представлений в [56]. Він складається з семи кроків, які охоплюють всі дії, необхідні в процесі налагодження, з моменту виявлення проблеми до моменту усунення дефекту, що викликав збій. Сім кроків, що входять в цей підхід, такі:

1) відстежити проблему;

2) відтворити;

3) спростити тестовий приклад;

4) знайти можливі причини;

5) сфокусуватися на найбільш ймовірних причинах;

6) ізолювати ланцюжок який призводить до помилки;

7) виправити дефект.

Кроки з 4 по 6 підходу складають цикл «Знайти–Сфокусуватися–Ізолювати». Цикл, який є найважчим, оскільки ці кроки часто повинні застосовуватися ітераційно, щоб знайти корінь проблеми. Один з методів, який може бути використаний в циклі «Знайти–Сфокусуватися–Ізолювати», називається дельта-налагодженням, який може бути використаний для систематичного звуження кола можливих джерел помилок шляхом порівняння двох запусків програми, невдалого і вдалого [57]. У літературі також описано кілька варіантів удосконалення дельта-налагодження [56, 58–60].

Інший підхід [61], полягає в тому, щоб зосередитися на аномаліях, тобто порівняти поведінку виконання з нормальною поведінкою програми. Приклади методик, пов’язаних з цим підходом, можна знайти в [62].

Наступний тип методів заснований на пошуку динамічних графів викликів. Граф динамічних викликів – це представлення виконання програми, що відбиває структуру викликів методів [63]. Методи, засновані на пошуку таких графів можна знайти в [63, 64].

Список типів методів, згаданих вище, далеко не вичерпний. Інші типи методів представлені в [65–69], також існує комбінації методів [70–75]. У [76] розглянуті сучасні тенденції в методах налагодження програмного забезпечення.

В [56] ввели поняття наукового методу налагодження. Отримавши звіт про проблему, розробник намагається відтворити помилку, спостерігаючи за збоєм. Потім, ґрунтуючись на спостереженнях, розробник створює гіпотези про причини збою. Після цього розробники перевіряють свої гіпотези. Для цього розробники можуть вивести значення в певному місці, щоб визначити, відкинути гіпотезу чи ні. Часто вони можуть змінити вихідний код і оцінити, чи працює програма так, як очікувалося. Якщо програма дає несподіваний результат, значить, гіпотеза відкинута. Якщо гіпотеза відкинута, розробнику необхідно висунути нову гіпотезу про причини збою. Якщо гіпотеза підтверджується і помилка виправлена, програмісти припиняють процес налагодження. Якщо гіпотеза підтверджується, але помилка не виправлена, розробники уточнюють гіпотезу і перевіряють її знову.

У дослідженні [77] була запропонована систематична процедура налагодження. Дослідження показало, що лише деякі студенти використовували структурний підхід, але швидко повернулися до неструктурних. Студентів просили знайти дефекти в коді та виправити їх. Крім того, студентів попросили задокументувати свій підхід. В результаті, щоб поліпшити навички налагодження у студентів, був розроблений системний підхід. Недоліком даного підходу є те, що висновок про навички ґрунтується на підході, задокументованому студентом самостійно і вручну.

Крім використання описаних вище методів, в процесі налагодження розробники можуть покладатися на символьні налагоджувачі, такі як GDB [78]. Такі налагоджувачі дозволяють розробникам вказувати точки в програмі, в яких виконання повинно бути припинено. Типовий символьний налагоджувач підтримує різні типи точок зупину та опції для подальшого уточнення моменту, коли програма повинна бути припинена, наприклад, шляхом зазначення умови яке вказує, чи слід призупинити виконання при досягненні точки зупину. Коли

виконання програми припинено, розробники можуть використовувати налагоджувачі, щоб, наприклад, досліджувати значення змінних, стек викликів, проходу по коду та оцінки виразів [78]. На основі символьних налагоджувачів, було розроблено кілька графічних символьних налагоджувачів, наприклад, DDD [79]. Більшість функцій символьного налагодження нині інтегровані в середовища розробки. Дослідження присвячене таким налагоджувачам в середовищах розробки та тому, як розробники використовують їх.

Щоб дізнатися, якою мірою описані вище методи та техніки використовуються в [80] вивчали практику розробників налагодження професійного програмного забезпечення. Вони стежили за вісьмома розробниками програмного забезпечення в чотирьох різних компаніях протягом дня, спостерігаючи за їх методами, даючи їм подумати вголос і відповісти на кілька запитань. Результати показали, що жоден з розробників не мав спеціальної освіти або підготовки в області налагодження. Більш того, всі розробники використовують спрощений науковий метод, який полягає у формулюванні та перевірки гіпотез, як описано в [56]. Всі учасники вміють користуватися символьними налагоджувачами, а також вважають за краще їх використання ніж оператори виведення на екран. Нарешті, вони виявили, що жоден з розробників не знав про більш функціональні можливості, такі як умовні точки зупину. Ґрунтуючись на відповідях 303 респондентів, вони виявили, що навчання налагодженню все ще рідкість, але останнім часом все більше курсів стали включати його у свої плани. Разом з тим майже немає кореляції між використовуваною мовою програмування та інструментами налагодження.

В [81, 82] вивчили, як програмісти шукають інформацію і надали систему рекомендацій для навігації під час сесій налагодження [83]. Вони виявили, що розробники шукають інформацію абсолютно по-різному, коли їх просять виправити помилку і коли їм потрібно зібрати інформацію, щоб виправити помилку. Вони також виявили, що розробники витрачають половину свого часу налагодження на пошук інформації.

В [84] провели експерименти, щоб визначити відмінності між експертами та новачками в налагодженні шляхом проведення двох експериментів, в яких порівнювалося, як ці дві групи підходять до вирішення завдань. Новачками в цьому дослідженні були студенти, які нещодавно закінчили перший або другий курс. Експертами були аспіранти кафедри факультету комп'ютерних наук. У першому експерименті кожному учаснику було надано 30 хвилин на виконання кожної програми. Сесії записувалися на відео, і учасників просили розмірковувати вголос для подальшого аналізу. У наданих дефектних програмах були приведені три різних типи помилок: некоректний аргумент функції, неправильний вивід на екран, помилки з областю дії змінної. Експерти в цьому експерименті знаходили дефекти швидше, ніж новачки, і перевіряли менше гіпотез – тобто, експерти були більш ефективні у визначенні правильної причини для даного дефекту програми. Новачки змогли визначити правильну причину дефекту тільки у 21% випадків [84].

У другому експерименті були ті ж самі учасники. На цей раз давали одну програму для налагодження. Додаткові матеріали включали друкований опис того, що програма повинна робити, як повинен виглядати правильний результат, папір для записів і калькулятор. Результати, отримані при аналізі того, як різні групи вирішували завдання, виявилися дещо несподіваними. Автори очікували, що програміст відразу ж приступить до розв'язання проблеми, редагуючи програму, не намагаючись спочатку зрозуміти її. Однак, це виявилося не так. Основні відмінності між групами новачків і експертів в даному дослідженні полягали в тому, що новачкам потрібно більше часу для визначення правильної причини помилки, і в цілому вони були менш успішними, ніж група експертів, в усуненні дефектів. Дослідження також показало, що новачки схильні додавати дефекти при спробі усунути первинну проблему. Експерти, що брали участь в дослідженні, робили це рідко [84].

В [85] проаналізували помилки компіляції студентів протягом семестру, що включає 15 вправ і 2 іспити. В ході аналізу виявили 226 різних смислових помилок. Їх результати були розділені на 6 класів помилок по 7 темам

інформатики. Теми охоплювали умовні вирази, цикли, методи, масиви, класи, файли та рядки. Класи помилок включали: поле не знайдено, використання нестатичної змінної всередині статичного методу, невідповідність типів, використання неініціалізованої змінної, виклик методу з неправильними аргументами та ім’я методу не знайдено [85]. На другому етапі дослідження порівнювалися «хороші» та «слабкі» студенти налагоджувачі. Учасників попросили виправити помилкову програму, яка містить компіляційні та логічні помилки протягом 2 годин. Помилки, включені в код, відповідали помилкам які найчастіше допускали студенти, які брали участь в першій фазі дослідження. Отримані результати статистично показали, що хороший програміст не обов’язково є хорошим налагоджувачем, але хороший налагоджувач зазвичай є хорошим програмістом. Автори пояснюють ці результати наступним чином: хороший програміст, який не обов’язково є хорошим налагоджувачем, здатний писати відносно прості програми, які найчастіше компілюються і працюють правильно тому їм не потрібно налагоджувати програми так часто і вони уникають розвиток навичок, якими вони ще не володіють. З іншого боку, хороші налагоджувачі, як правило, є і хорошими програмістами, тому що знання системи та мови, яке робить їх хорошими налагоджувачами підвищує їх компетентність як програмістів. Автори також вважають, що завдання з програмування не оцінюються за належними критеріями, що дозволяє студентам з меншим розумінням теми отримувати вищі оцінки [85].

Інструменти для обчислювального мислення, одним з аспектів якого є налагодження, розроблені в [86]. Для пояснення їх структури використовується навчальна гра Light-Bot. Недоліком Light-Bot є те, що вона використовується виключно для навчання навичкам програмування і налагодження, а не для їх аналізу. Крім того, вона має дуже обмежений набір інструментів налагодження, і при переході на професійне середовище розробки доведеться проводити процес навчання повторно.

В роботі [87] на основі вивчення набору даних BlueJ-Blackbox представлено помилки, з якими зазвичай стикаються початківці Java фахівці. Результати

дослідження показують, що семантичні помилки зустрічаються частіше, ніж синтаксичні, особливо серед досвідчених програмістів. Недоліком є те, що аналізуються тільки помилки компіляції та тільки з середовища BlueJ IDE, яка використовується для навчання програмуванню і радикально відрізняється від професійного середовища.

У дослідженні [88] було організовано вивчення того, як програмісти встановлюють точки зупину. Аналізуючи оператори, за допомогою яких розробники встановлюють точки зупину, було виявлено, що 53% точок зупину були встановлені в операторах виклику і тільки 1% в циклах. Ці дані також підтверджуються дослідженням [89]. Висновок даного дослідження полягає в тому, що установка точок зупину і покрокове виконання коду є найбільш часто використовуваними методами налагодження.

Те, як користувачі проводять час за роботою в Microsoft Visual Studio, розглядалося в [90]. Дані про процеси розробки та налагодження програмного забезпечення витягуються з середовища розробки. Однак інформація про налагодження обмежується точками зупинки. Крім того, добута інформація ніяк не аналізується.

Приховані моделі Маркова також використовуються як засіб пошуку поведінки розробника з даних про взаємодію з середовищем розробки [91]. Відповідно до цього підходу, вивчена серія сесій налагодження за участю близько 200 професійних розробників з ABB Inc. Розроблена модель налагодження фіксує поведінку розробників під час установки точок зупину, початку налагодження і покрокового виконання коду. Недоліком запропонованого підходу є те, що він збирає дані лише для кількох інструментів налагодження і є напівавтоматичним. Процес налагодження будується вручну експертом. Крім того, з середовища розробки витягуються тільки дані про налагодження без урахування процесу розробки.

В [92] реалізована інфраструктура Swarm Debug Infrastructure (SDI), що надає інструменти для збору та обміну налагоджувальних дій. Програмісти можуть використовувати загальний досвід попередніх сеансів налагодження.

SDI була оцінена в експерименті за участю 10 інженерів. SDI призначений тільки для збору та обміну інформацією про встановлені точки зупинки та шляхи налагодження. У ньому не реалізований аналіз процесу налагодження і навичок розробників.

В роботі [93] представлений онлайн інструмент Ladebug, призначений для підтримки навчання навичкам налагодження. У цьому інструменті студенти використовують систематичний процес налагодження для виявлення і виправлення помилок в заздалегідь заданих вправах. Представлений інструмент використовується тільки для навчання основам налагодження і не аналізує навички студентів. Крім того, процес налагодження відбувається всередині інструменту, який використовується тільки для навчання з дуже обмеженим набором інструментів. При переході до професійного середовища розробки студентам доведеться проходити процес навчання, використовуючи реальні, а не навчальні інструменти.

У статті [94] вивчалися когнітивні процеси студентів під час налагодження програм, з використанням відстеження руху очей. Рухи очей студентів записувалися, щоб побачити, як поводяться студенти різних рівнів навичок під час налагодження. Дослідження показало, що програмісти початківці при налагодженні комп'ютерних програм дотримуються лінійного підходу, в той час, як студенти з досвідом програмування дотримуються більш логічного і стратегічного підходу. Недоліком такого підходу є те, що неможливо проаналізувати процес налагодження під час індивідуальної роботи.

Розуміння діяльності по налагодженню може допомогти практикам і дослідникам розробити нове сімейство інструментів налагодження, більш ефективних і більш пристосованих до специфіки кожного завдання і, таким чином, підвищить продуктивність розробників.

1.3 Використання методів Process Mining в аналізі процесів розробки та налагодження програмного забезпечення

За останнє десятиліття застосування Process Mining було ефективним в аналізі процесів на основі даних про події.

Метою Process Mining є виявлення, моніторинг і поліпшення процесів за допомогою даних з журналу подій інформаційної системи. Process Mining застосовує спеціалізовані алгоритми інтелектуального аналізу даних до журналу подій [95].

Цільова група IEEE випустила маніфест Process Mining [96]. Маніфест підтримали 53 організації та 77 експертів з Process Mining. Він спрямований на просування Process Mining. Крім того, визначаючи набір правил і перераховуючи критичні питання, цей маніфест повинен служити керівництвом для розробників програмного забезпечення, вчених і кінцевих користувачів.

Process Mining може бути в рівній мірі застосований до програмного забезпечення [97–99]. Використовуючи методи Process Mining, в роботі [100] вивчали, як програмісти взаємодіють з репозиторіями програмного забезпечення.

У дослідженні [101] вивчалися записи в системах відстеження проблем. Загальний недолік цих робіт полягає в тому, що дані витягуються з систем, які не дозволяють оцінити внесок студента в результат, оскільки не надають інформацію про процес розробки і налагодження програмного забезпечення.

У статті [98] для перевірки поведінки розробників використовувалася метод перевірки на відповідність. Оцінювалися дії 40 розробників початківців, з кодування в п’яти сесіях розробки. Робота в основному зосереджена на використанні підходу на основі відповідності, порівнюючи виконання процесів, записаних в журналах подій, з деякими з очікуваних моделей поведінки, представлених у вигляді моделі процесу. Подібний підхід був представлений в дослідженні [102], але там була представлена тільки базова концепція, залишивши її реалізацію і перевірку для подальшої роботи.

В [103] журнали подій використовуються для підвищення якості системи. В [104], Process Mining використовується для ідентифікації патернів програмних систем. В [105] автори пропонують метод для ідентифікації архітектурної моделі програмного забезпечення.

У той час як перераховані вище роботи в основному спрямовані на розуміння програмних систем, дане дослідження полягає в застосуванні методів Process Mining для ідентифікації та розуміння поведінки розробників під час розробки та налагодження коду.

Висновки до першого розділу

Систематизація результатів досліджень свідчить про нестачу знань про те, як програмісти усувають проблеми в процесі розробки та налагодження програмного забезпечення.

Існує ряд програмних інструментів для аналізу якості текстів програм, але вони аналізують тільки готові тексти, а для аналізу процесів програмування та налагодження таких інструментів немає.

Як ми бачимо, багатогранні дослідження процесів розробки та налагодження не засновані на моделях процесів і не дозволяють вивчити процес розробки та налагодження кожного конкретного розробника програмного забезпечення.

З огляду на вище сказане, основною задачею дослідження є розробка методів та інструментів для відстеження, моделювання та аналізу процесів розробки та налагодження програмного забезпечення.

За матеріалами розділу опубліковано роботи [106–114].

РОЗДІЛ 2
КОНСТРУКТИВНО-ПРОДУКЦІЙНЕ МОДЕЛЮВАННЯ ПРОЦЕСІВ РОЗРОБКИ ТА НАЛАГОДЖЕННЯ ПРОГРАМ

За останні роки було запропоновано кілька мов для моделювання процесу розробки [115]. Серед них:

- Directly Follows Graphs [98];
- методи, засновані на онтологіях [120];
- моделі Маркова [121].

Для опису процесів розробки та налагодження програмного забезпечення застосована методологія конструктивно-продукційного моделювання. Дослідження на тему конструктивно-продукційного моделювання представлені в роботах [116–119].

Конструктивно-продукційне моделювання можна використовувати для моделювання і формалізації будь-яких структур і конструктивних процесів.

Інструменти конструктивно-продукційного моделювання використовувалися для вирішення таких завдань, як:

- складання розкладу занять університету [122, 123];
- моделювання спалахів блискавки в грозовому фронті [124];
- моделювання адаптації алгоритмів [125];
- формалізація та автоматизація процесу порівняння документів для виявлення текстових запозичень [126] і багато інших.

Широкий спектр цих завдань демонструє універсальність і перспективність використання конструктивно-продукційного моделювання для вирішення завдань різних предметних областей.

Зокрема, методологія конструктивно-продукційного моделювання була застосована для формалізації процесів складання розкладу занять [122, 123]. Формування розкладу занять виконувалося двома конструкторами. В результаті реалізації першого конструктора формувався коректний, але не оптимальний розклад занять університету, який є початковою популяцією для генетичного

алгоритму. Після формування розкладу слідував етап перетворення за допомогою генетичного алгоритму. Для формалізації цього етапу розроблений другий конструктор. В результаті реалізації якого формується розклад занять університету.

Для дослідження практичного застосування запропонованого методу він був програмно реалізований і експериментально перевірений при вирішенні завдання формування розкладу занять Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна. Експеримент включав два етапи: на першому етапі відбувався збір вхідної інформації, на другому завантаження вхідних даних і автоматизоване формування розкладу. Проведений експеримент підтвердив працездатність запропонованих методів і програмних засобів, які їх реалізують.

Основи онтології узагальненого конструктора наведені в роботах [127, 128]. В роботі наведені лише ті складові, які необхідні для подальшого викладу матеріалу.

Розробка конструктора передбачає визначення елементів носія, що можуть бути розширені, сигнатури відношень і операцій, а також інформаційного забезпечення.

Сигнатура $\Sigma_L = \langle \Xi, \Theta, \Phi, \{\rightarrow, \hookleftarrow\}, \Psi \rangle$ складається з кінцевої множини операцій і відповідних відношень, де $\Xi \supset \{\bullet, :\}$ – відношення та операції зв'язування і перетворення елементів носія, $\Theta = \{\Rightarrow, |\Rightarrow, \|\Rightarrow\}$ – операції підстановки та виведення, Φ – відношення та операції над атрибутами, $(\rightarrow)$ – відношення підстановки, $(\hookleftarrow)$ – відношення атрибутивності. $\Psi = \{\psi_i : \langle s_i, g_i \rangle\}$ – кінцева множина правил підстановки, s_i – послідовність відношень підстановки, g_i – кінцева множина операцій над атрибутами. Якщо операції над атрибутами не виконуються, то правило підстановки буде виглядати $\langle s_i, \varepsilon \rangle$, де ε – порожній символ.

Інформаційне забезпечення конструювання включає: онтологію, мету, правила, обмеження, початкові умови, умови завершення конструювання.

Найскладнішою частиною є створення правил підстановки, що визначають процес виведення відповідних конструкцій.

Для формування конструкцій необхідно виконати ряд перетворень узагальненого конструктора, а саме:

- спеціалізацію;
- інтерпретацію;
- конкретизацію;
- реалізацію.

Спеціалізація визначає семантичну природу носія, призначення конструктора, кінцевий набір операцій, їх семантику та атрибути, порядок виконання та обмеження [116, 118].

Інтерпретація – це зв’язування алгоритмів, що реалізують певну алгоритмічну структуру (конструктор алгоритмів), з операціями сигнатури. У процесі інтерпретації відбувається зв’язування моделей конструктора і внутрішнього виконавця. В результаті виходить конструктивна система, що володіє інструментами конструювання [116, 118].

Під конкретизацією конструктора мається на увазі визначення конкретних правил підстановки, обмежень, початкових умов і умов завершення побудови, основних елементів носія з їх властивостями та значеннями властивостей. Після операцій інтерпретації та конкретизації, виконуваних зовнішнім виконавцем, конструктивна система має все необхідне для створення конструкцій [116–119].

Реалізація, здійснювана внутрішнім виконавцем системи, полягає у формуванні структури носіїв-елементів шляхом виконання алгоритмів, пов’язаних з операціями підстановки. Реалізовано може бути тільки конструктор, який був попередньо спеціалізований, інтерпретований і конкретизований.

2.1 Конструктивно-продукційний підхід до моделювання процесу розробки програмного забезпечення

2.1.1 Конструктор історії розробки програми

Мета конструктора – сформувати історію процесу розробки програми.

Початкові умови – нетермінал σ, з якого починається виведення.

Умови завершення – розробку програми завершено і всі події оброблені.

Спеціалізація узагальненого конструктора на основі конструктивно-продукційного підходу до формування історії процесу розробки програмного забезпечення:

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_S\!\mapsto C_{CH} = \langle M_{CH}, \Sigma_{CH}, \Lambda_{CH} \rangle, \qquad (2.1)$$

де ${}_S\!\mapsto$ операція спеціалізації (виконує зовнішній виконавець), M_{CH} неоднорідний розширюваний носій, який містить множини терміналів і нетерміналів, Σ_{CH} сигнатура відношень і відповідних операцій, Λ_{CH} інформаційне забезпечення конструювання.

Термінали з їх атрибутами:

– ${}_{context,time,project,developer}\,textChangedEvent$ – подія від зовнішнього виконавця (середовища розробки) про те, що код змінився. Його атрибути: *context* – контекст, який містить всю необхідну інформацію для обробки даної події, заповнюється зовнішнім виконавцем; *time* – час, коли відбулася подія; *project* – інформація про проєкт; *developer* – інформація про розробника;

– ${}_{id,name}\,developer$ – інформація про розробника. Його атрибути: *id* – ідентифікатор; *name* – ім’я розробника;

– ${}_{id,name}\,project$ – інформація про проєкт. Його атрибути: *id* – ідентифікатор; *name* – назва проєкту;

– ${}_{newSnapshot,oldSnapshot,changeType,countChangedLines,lineNumber}\,context$ – контекст *textChangedEvent* події. Його атрибути: *newSnapshot* – поточний код в

текстовому редакторі; *oldSnapshot* – попередній код в текстовому редакторі; *changeType* – тип зміни, може бути одним з *Insert, Edit, Delete*; *countChangedLines* – кількість рядків, які були змінені; *lineNumber* – номер рядка, в якому відбулися зміни;

– $_{time,\,project,\,developer}$*closedEvent* – подія від зовнішнього виконавця (середовища розробки) про те, що середовище розробки буде закрито. Його атрибути: *time* – час, коли відбулася подія; *project* – інформація про проєкт; *developer* – інформація про розробника;

– $_{value,\,info,\,snapshotContext,\,time,\,project,\,developer}$*snapshot* – моментальний знімок коду в середовищі розробки. Його атрибути: *value* – поточний код в текстовому редакторі; *info* – інформація про відмінність поточного коду від попереднього; *snapshotContext* – контекст; *time* – час, коли відбулася подія; *project* – інформація про проєкт; *developer* – інформація про розробника;

– $_{changeType,\,changeMode,\,lineNumber}$*info* – інформація про відмінність поточного коду від попереднього. Його атрибути: *changeType* – тип зміни, може бути одним з *Insert, Edit, Delete*; *changeMode* – режим зміни, може бути одним з *single,multi*; *lineNumber* – номер рядка, в якому відбулися зміни;

– $_{snapshot}^{index,\,length}$*snapshots* – масив моментальних знімків коду. Його атрибути: *index* – індекс моментального знімка *snapshot* в масиві; *length* – розмір масиву;

– $_{chain}^{index,\,length}$*chains* – масив ланцюжків. Його атрибути: *index* – індекс ланцюжка *chain* в масиві; *length* – розмір масиву;

– $_{number,\,lineNumber,\,text,\,context,\,time,\,project,\,developer}$*chain* – ланцюжок з інформацією про історію розробки програми. Його атрибути: *number*– номер; *lineNumber* – номер рядка в останньому моментальному знімку коду; *text* – вміст ланцюжка;

context – контекст; *time* – час, коли відбулася подія; *project* – інформація про проєкт; *developer* – інформація про розробника;

– $_{value}previousChangedLineNumber$ – номер попереднього рядка, в якому були зроблені зміни. Має атрибут *value*;

– $_{value}i$ – лічильник. Має атрибут *value*;

– $_{value}r_j$ – тимчасова змінна. Має атрибут *value*.

Вводяться наступні операції над атрибутами:

– $==(leftOperand, rightOperand, result)$ – операція порівняння на рівність значень *leftOperand* і *rightOperand*, та запис результату в *result*;

– $>(leftOperand, rightOperand, result)$ – операція порівняння, при якому значення лівого операнда *leftOperand* більше, ніж значення правого операнда *rightOperand*, та запис результату в *result*;

– *&(leftOperand,rightOperand,result)* – операція бінарної кон'юнкції, результат *result* дорівнює найменшому значенню операндів *leftOperand* і *rightOperand*;

– $\lhd(attribute, condition, valueIfTrue, valueIfFalse)$ – операція присвоєння значення *valueIfTrue* до атрибуту *attribute*, якщо умова *condition* виконується, інакше присвоєння значення *valueIfFalse*;

– $\rhd(condition, operation_1, operation_2)$ – операція виконання дії $operation_1$, якщо умова *condition* виконується, інакше виконання дії $operation_2$;

– $\succ(array, attributeName, attributeValue, record, operation)$ – операція пошуку в масиві *array* елементу, атрибут *attributeName* якого має значення *attributeValue* і заміни цього елементу на *record*, якщо елемент не знайдено, виконання дії *operation*;

– $\prec(array, newRecord)$ – операція додавання нового запису *newRecord* в масив *array*;

– $\equiv(array, index, result)$ – операція пошуку в масиві *array*, по індексу *index* і присвоєння результату в *result*;

– $\approx(array, index_1, index_2, attributeName, result)$ – операція пошуку в масиві *array* елементів по індексам $index_1, index_2$, порівняння значень атрибута *attributeName* зі значеннями знайдених елементів і присвоєнням результату в *result*;

– $\circ(terminal)$ – операція запису значень атрибутів терміналу *terminal* зовнішнім виконавцем.

Щоб інтерпретувати C_{CH} необхідно уточнити базову алгоритмічну структуру [116, 118].

Нехай є наступна базова алгоритмічна структура:

$$C_{A,CH} = \langle \mathrm{M}_{A,CH}, V_{A,CH}, \Sigma_{A,CH}, \Lambda_{A,CH} \rangle, \tag{2.2}$$

де $V_{A,CH} = \{A_i^0 |_{X_i}^{Y_i}\}$ – множина основних алгоритмів внутрішнього виконавця конструювання.

Нижче представлені алгоритми, що реалізують операції над атрибутами:

$$\begin{aligned}
&A_{1,2,3} |_{leftOperand, rightOperand}^{result}, \; A_4 |_{attribute, condition, valueIfTrue, valueIfFalse}^{attribute}, \\
&A_5 |_{condition, operation_1, operation_2}^{operation_1, operation_2}, A_6 |_{array, attributeName, attributeValue, record, operation}^{array}, \\
&A_7 |_{array, newRecord}^{array}, \; A_8 |_{array, index}^{result}, \\
&A_9 |_{array, index_1, index_2, attributeName}^{result}, \; A_{10} |_{terminal}^{terminal},
\end{aligned} \tag{2.3}$$

де A_i ідентифікатор алгоритму, X_i, Y_i множини визначень і значень алгоритму $A_i^0 |_{X_i}^{Y_i}$.

Інтерпретуємо конструктор:

$$
\begin{aligned}
&\langle C_{CH} = \langle M_{CH}, \Sigma_{CH}, \Lambda_{CH} \rangle, \\
&C_{A,CH} = \langle M_{A,CH}, V_{A,CH}, \Sigma_{A,CH}, \Lambda_{A,CH} \rangle \rangle_I \mapsto C_{I,CH} = \langle M_{I,CH}, \Sigma_{I,CH}, \Lambda_{I,CH} \rangle, \\
&\Lambda_{I,CH} = \Lambda_{CH} \cup \{ \\
&(A_1 \mid {}^{result}_{leftOperand, rightOperand} \hookleftarrow ==), \\
&(A_2 \mid {}^{result}_{leftOperand, rightOperand} \hookleftarrow >), \\
&(A_3 \mid {}^{result}_{leftOperand, rightOperand} \hookleftarrow \&), \\
&(A_4 \mid {}^{attribute}_{attribute, condition, valueIfTrue, valueIfFalse} \hookleftarrow \triangleleft), \\
&(A_5 \mid {}^{operation_1, operation_2}_{condition, operation_1, operation_2} \hookleftarrow \triangleright), \\
&(A_6 \mid {}^{array}_{array, attributeName, attributeValue, record, operation} \hookleftarrow \succ), \\
&(A_7 \mid {}^{array}_{array, newRecord} \hookleftarrow \prec), \\
&(A_8 \mid {}^{result}_{array, index} \hookleftarrow \equiv), \\
&(A_9 \mid {}^{result}_{array, index_1, index_2, attributeName} \hookleftarrow \approx), \\
&(A_{10} \mid {}^{terminal}_{terminal} \hookleftarrow \circ) \},
\end{aligned}
\tag{2.4}
$$

де $\mapsto_I$ – операція інтерпретації.

Проведемо конкретизацію конструктора C_{CH} призначеного для створення історії розробки коду:

$$C_{I,CH} = \langle M_{I,CH}, \Sigma_{I,CH}, \Lambda_{I,CH} \rangle_K \mapsto C_{K,CH} = \langle M_{K,CH}, \Sigma_{K,CH}, \Lambda_{K,CH} \rangle, \tag{2.5}$$

де $\mapsto_K$ – операція конкретизації.

Правила підстановки наведені нижче (2.6) – (2.10).

В результаті виконання правила s_1, накопичуються події *textChangedEvent*. Дані з контексту події перетворюються в моментальний знімок коду зі збереженням інформації про зміну. Разом з тим, щоб не збирати надлишкові дані, знімки перезаписуються, якщо зміна відбувається на тому самому рядку, що і до цього

$$
\begin{aligned}
&s_1 = \langle \sigma \to \sigma \circ (textChangedEvent) \rangle, \\
&g_1 = \langle value\text{↵}snapshot = \\
&newSnapshot\text{↵}context\text{↵}textChangedEvent, \\
&changeType\text{↵}info\text{↵}snapshot = \\
&changeType\text{↵}context\text{↵}textChangedEvent, \\
&lineNumber\text{↵}info\text{↵}snapshot = \\
&lineNumber\text{↵}context\text{↵}textChangedEvent, \\
&context\text{↵}snapshot = context\text{↵}textChangedEvent, \\
&time\text{↵}snapshot = time\text{↵}textChangedEvent, \\
&developer\text{↵}snapshot = developer\text{↵}textChangedEvent, \\
&project\text{↵}snapshot = project\text{↵}textChangedEvent, \\
&> (countChangedLines\text{↵}context\text{↵}textChangedEvent, 1, value\text{↵}r_1), \\
&\lhd (changeMode\text{↵}info\text{↵}snapshot, value\text{↵}r_1, 'multi', 'single'), \\
&== (value\text{↵}previousChangedLineNumber, \\
&lineNumber\text{↵}context\text{↵}textChangedEvent, value\text{↵}r_1), \\
&== (countChangedLines\text{↵}context\text{↵}textChangedEvent, 1, value\text{↵}r_2), \\
&\& (value\text{↵}r_1, value\text{↵}r_2, value\text{↵}r_3), \\
&\rhd (value\text{↵}r_3, \succ (snapshots, 'value', \\
&oldSnapshot\text{↵}context\text{↵}textChangedEvent, snapshot, \\
&\prec (snapshots, snapshot)), \\
&\prec (snapshots, snapshot)), \\
&value\text{↵}previousChangedLineNumber = \\
&lineNumber\text{↵}context\text{↵}textChangedEvent \rangle.
\end{aligned}
\tag{2.6}
$$

Правило s_2 застосовується, якщо подія *closedEvent* отримана від зовнішнього виконавця

$$s_2 = \langle \sigma \to chains\,\alpha \circ (closedEvent) \rangle. \tag{2.7}$$

Потім буде виконано правило s_3. В результаті виконання правила s_3, в масив з результатами буде додано перший ланцюжок, значенням якого є код першого моментального знімка

$$
\begin{aligned}
&s_3 = \langle \alpha\, textChangedEvent_{d_3} \to \alpha \rangle, \\
&\tau_1 \lrcorner \tau\, g_{3.1} = \langle \\
&== (value \lrcorner i, 0, value \lrcorner r_1), \\
&d_3 = \langle value \lrcorner r_1 \rangle\rangle, \\
&\tau_0 \lrcorner \tau\, g_{3.2} = \langle \\
&\equiv (snapshots, value \lrcorner i, snapshot), \\
&value \lrcorner i = value \lrcorner i + 1, \\
&number \lrcorner chain = value \lrcorner i, \\
&lineNumber \lrcorner chain = \\
&lineNumber \lrcorner info \lrcorner snapshot, \\
&context \lrcorner chain = context \lrcorner snapshot, \\
&time \lrcorner chain = time \lrcorner snapshot, \\
&developer \lrcorner chain = developer \lrcorner snapshot, \\
&project \lrcorner chain = project \lrcorner snapshot, \\
&text \lrcorner chain = value \lrcorner snapshot, \\
&\prec (chains, chain) \rangle.
\end{aligned}
\tag{2.8}
$$

Коли правило s_4 буде виконано n разів в результаті послідовного порівняння моментальних знімків коду, ланцюжки будуть заповнені необхідними значеннями та додані в масив

$$
\begin{aligned}
&s_4 = \langle \alpha\, textChangedEvent_{d_4} \to \alpha \rangle, \\
&\tau_1 \lrcorner \tau\, g_{4.1} = \langle \\
&> (value \lrcorner i, 0, value \lrcorner r_1), d_4 = \langle value \lrcorner r_1 \rangle\rangle, \\
&\tau_0 \lrcorner \tau\, g_{4.2} = \langle \\
&\approx (snapshots, value \lrcorner i, value \lrcorner i - 1, 'value', text \lrcorner chain), \\
&\equiv (snapshots, value \lrcorner i, snapshot), \\
&value \lrcorner i = value \lrcorner i + 1, \\
&number \lrcorner chain = value \lrcorner i, \\
&lineNumber \lrcorner chain = lineNumber \lrcorner info \lrcorner snapshot, \\
&\prec (chains, chain) \rangle
\end{aligned}
\tag{2.9}
$$

$$s_5 = \langle \alpha \to \varepsilon \rangle. \qquad (2.10)$$

В результаті формується масив ланцюжків, що зображає історію розробки програми.

2.1.2 Конструктор перетворювач історії розробки програми у файл журналу подій

Мета конструктора – класифікувати історію розробки програми по дрібномасштабним змінам та створити файл журналу подій. Типи дрібномасштабних змін наведені в таблиці 2.1.

Початкові умови – нетермінал σ, з якого починається виведення.

Умови завершення – всі зміни коду з історії розробки програми класифіковані, і файл журналу подій створено.

Спеціалізація узагальненого конструктора на основі конструктивно-продукційного підходу до формування файлу журналу подій:

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_S\!\mapsto C_{EL} = \langle M_{EL}, \Sigma_{EL}, \Lambda_{EL} \rangle, \qquad (2.11)$$

де ${}_S\!\mapsto$ операція спеціалізації (виконує зовнішній виконавець), M_{EL} неоднорідний розширюваний носій, який містить множини терміналів і нетерміналів, Σ_{EL} сигнатура відношень і відповідних операцій, Λ_{EL} інформаційне забезпечення конструювання.

Таблиця 2.1 – Типи дрібномасштабних змін коду

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

Термінали з їх атрибутами:

– $_{traces}log$ – файл журналу подій. Має атрибут *traces* – масив ланцюжків дій;

– $_{trace}^{index,length}traces$ – масив ланцюжків дій. Його атрибути: *index* – індекс ланцюжка *trace* в масиві; *length* – розмір масиву;

– $_{id,startTime,finishTime,project,developer,events}trace$ – масив подій, створених в результаті одного виконання процесу. Його атрибути: *id* – ідентифікатор; *startTime* – час початку; *finishTime* – час закінчення; *project* – інформація про проєкт; *developer* – інформація про розробника; *events* – масив подій, що сталися під час сеансу розробки;

– $_{id,name}developer$ – інформація про розробника. Його атрибути: *id* – ідентифікатор; *name* – ім’я розробника;

– $_{id,name}project$ – інформація про проєкт. Його атрибути: *id* – ідентифікатор; *name* – назва проєкту;

– $_{event}^{index,length}events$ – масив подій, що сталися під час сеансу розробки. Його атрибути: *index* – індекс події *event* в масиві; *length* – розмір масиву;

– $_{name,time,context}event$ – подія під час сеансу розробки. Його атрибути: *name* – ім’я, *time* – час події, *context* – контекст, об’єкт динамічної структури з інформацією про оточення події.

Вводяться наступні операції над атрибутами:

– $\prec(array, newRecord)$ – операція додавання нового запису *newRecord* в масив *array*;

– $\circ(terminal)$ – операція запису значень атрибутів терміналу *terminal* зовнішнім виконавцем;

– $\propto(chains, i, chain)$ – операція отримання елемента за індексом i з масиву *chains* і запис результату в *chain*;

– $\cong(chains, i, category)$ – операція отримання елемента за індексом i з масиву $chains$, класифікація його значення та запис у $category$;

– $\pm(events, traces)$ – операція групування $events$ по $traces$;

– $\mp(traces, \log)$ – операція збереження $traces$ в файл журналу подій $\log$.

Щоб інтерпретувати C_{EL} необхідно уточнити базову алгоритмічну структуру [117, 119].

$$C_{A,EL} = \langle \mathrm{M}_{A,EL}, V_{A,EL}, \Sigma_{A,EL}, \Lambda_{A,EL} \rangle, \tag{2.12}$$

де $V_{A,EL} = \{A_i^0 |_{X_i}^{Y_i}\}$ – множина основних алгоритмів внутрішнього виконавця конструювання.

Нижче представлені алгоритми, що реалізують операції над атрибутами:

$$\begin{aligned} &A_1 |_{array,newRecord}^{array}, A_2 |_{terminal}^{terminal}, \\ &A_3 |_{chains,i}^{chain}, A_4 |_{chains,i}^{category}, \\ &A_5 |_{events}^{traces}, A_6 |_{traces}^{log}, \end{aligned} \tag{2.13}$$

де A_i ідентифікатор алгоритму, X_i, Y_i множини визначень і значень алгоритму $A_i^0 |_{X_i}^{Y_i}$.

Інтерпретуємо конструктор:

$$\begin{aligned} &\langle C_{EL} = \langle M_{EL}, \Sigma_{EL}, \Lambda_{EL} \rangle, \\ &C_{A,EL} = \langle M_{A,EL}, V_{A,EL}, \Sigma_{A,EL}, \Lambda_{A,EL} \rangle \rangle_I \mapsto C_{I,EL} = \langle M_{I,EL}, \Sigma_{I,EL}, \Lambda_{I,EL} \rangle, \\ &\Lambda_{I,EL} = \Lambda_{EL} \bigcup \{ \\ &(A_1 |_{array,newRecord}^{array} \hookleftarrow \prec), \\ &(A_2 |_{terminal}^{terminal} \hookleftarrow \circ), \\ &(A_3 |_{chains,i}^{chain} \hookleftarrow \propto), \\ &(A_4 |_{chains,i}^{category} \hookleftarrow \cong), \\ &(A_5 |_{events}^{traces} \hookleftarrow \pm), \\ &(A_6 |_{traces}^{\log} \hookleftarrow \mp) \}, \end{aligned} \tag{2.14}$$

де $_I\mapsto$ – операція інтерпретації.

Проведемо конкретизацію конструктора C_{EL}:

$$C_{I,EL} = \langle M_{I,EL}, \Sigma_{I,EL}, \Lambda_{I,EL} \rangle_K \mapsto C_{K,EL} = \langle M_{K,EL}, \Sigma_{K,EL}, \Lambda_{K,EL} \rangle, \qquad (2.15)$$

де $_K\mapsto$ – операція конкретизації.

Послідовне виконання правил будемо позначати як $\prod_{i=1}^{n} s_i$.

Правила підстановки наведені нижче (2.16) – (2.17).

Правило s_1 отримує історію розробки програми *chains* від зовнішнього виконавця і послідовно виконує правило s_2 для кожного елементу. Правило s_2 класифікує зміни та записує до файлу журналу подій

$$\begin{aligned} s_1 &= \langle \sigma \to chains \bullet \prod_{i=1}^{length\_chains} \alpha_i \rangle, \\ g_1 &= \langle \circ(chains) \rangle \end{aligned} \qquad (2.16)$$

$$\begin{aligned} s_2 &= \langle \log \alpha_i \to \varepsilon \rangle, \\ g_2 &= \langle \propto (chains, i, chain), \\ &\cong (chains, i, category), \\ &name\_event = category, \\ &time\_event = time\_chain, \\ &context\_event = context\_chain, \\ &\prec (events, event), \\ &\pm (events, traces), \mp (traces, \log) \rangle. \end{aligned} \qquad (2.17)$$

В результаті формується файл журналу подій з класифікованими змінами історії розробки програми.

2.2 Конструктивно-продукційний підхід до моделювання процесу налагодження програмного забезпечення

Конструктивно-продукційне моделювання застосовано для формалізації процесу збору даних про використання середовища розробки під час налагодження.

Мета конструктора – створити журнал подій, в якому показується процес налагодження.

Початкові умови – нетермінал σ, з якого починається виведення.

Умови завершення – всі події оброблено, і файл журналу подій створено.

Спеціалізація узагальненого конструктора на основі конструктивно-продукційного підходу до формування файлу журналу подій:

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_S\mapsto C_L = \langle M_L, \Sigma_L, \Lambda_L \rangle, \tag{2.18}$$

де ${}_S\mapsto$ операція спеціалізації (виконує зовнішній виконавець), M_L неоднорідний розширюваний носій, який містить множини терміналів і нетерміналів, Σ_L сигнатура відношень і відповідних операцій, Λ_L інформаційне забезпечення конструювання.

Термінали з їх атрибутами:

– ${}_{traces}log$ – файл журналу подій. Має атрибут *traces* – масив ланцюжків дій;

– ${}_{trace}{}^{index,length}traces$ – масив ланцюжків дій. Його атрибути: *index* – індекс ланцюжка *trace* в масиві; *length* – розмір масиву;

– ${}_{id,startTime,finishTime,project,developer,events}trace$ – масив подій, створених в результаті одного виконання процесу. Його атрибути: *id* – ідентифікатор; *startTime* – час початку; *finishTime* – час закінчення; *project* – інформація про проєкт; *developer* – інформація про розробника; *events* – масив подій, що сталися під час сеансу розробки;

– ${}_{id,name}developer$ – інформація про розробника. Його атрибути: *id* – ідентифікатор; *name* – ім’я розробника;

– ${}_{id,name}project$ – інформація про проєкт. Його атрибути: *id* – ідентифікатор; *name* – назва проєкту;

– $_{context}startDebuggingSessionEvent$ – подія про початок сесії налагодження від зовнішнього виконавця (середовища розробки). Має атрибут $context$ – контекст події;

– $_{context}debuggingEvent$ – подія під час сесії налагодження від зовнішнього виконавця (середовища розробки). Має атрибут $context$ – контекст події;

– $_{context}endDebuggingSessionEvent$ – подія про завершення сесії налагодження від зовнішнього виконавця (середовища розробки). Має атрибут $context$ – контекст події;

– $_{event}^{index,length}events$ – масив подій, що сталися під час сеансу налагодження. Його атрибути: $index$ – індекс події $event$ в масиві; $length$ – розмір масиву;

– $_{id,type,filePath,time,lineNumber,context}event$ – подія під час сеансу налагодження. Його атрибути: id – ідентифікатор; $type$ – тип; $filePath$ – шлях до файлу, який налагоджують; $lineNumber$ – номер рядка який налагоджують; $time$ – час події, $context$ – контекст події, об’єкт динамічної структури з інформацією про оточення події (середовище розробки, проєкт, розробник, файли), для різних подій структура об’єкта може бути різною;

– $_{id,startTime,finishTime,snapshots,events,developer,project}session$ – сеанс налагодження. Його атрибути: id – ідентифікатор; $startTime$ – час початку; $finishTime$ – час закінчення; $project$ – проєкт; $developer$ – розробник; $events$ – масив подій, що сталися під час сеансу налагодження; $snapshots$ – масив моментальних знімків коду (знімок – файл коду програми в певний момент часу), зроблених перед початком сеансу налагодження.

Вводяться наступні операції над атрибутами:

– $\prec(startDebuggingSessionEvent, session)$ – створення сеансу налагодження $session$ коли відбулася подія $startDebuggingSessionEvent$;

– $\succ(session, debuggingEvent)$ – операція додавання події $debuggingEvent$ до сеансу налагодження $session$;

– $\cong(session, endDebuggingSessionEvent)$ – завершення сеансу налагодження $session$ коли відбулася подія $endDebuggingSessionEvent$;

– $\circ(terminal)$ – операція запису значень атрибутів терміналу $terminal$ зовнішнім виконавцем;

– $\lhd(developerId, projectId, \log)$ – операція створення файлу журналу подій $\log$ для проєкту $projectId$ та розробника $developerId$;

– $\rhd(debuggingEvent, session, \log)$ – операція переміщення події $debuggingEvent$ з сеансу налагодження $session$ до файлу журналу подій $\log$;

– $\approx(\log)$ – операція збереження файлу журналу подій $\log$.

Щоб інтерпретувати C_L необхідно уточнити базову алгоритмічну структуру [116, 118].

$$C_{A,L} = \langle \mathrm{M}_{A,L}, V_{A,L}, \Sigma_{A,L}, \Lambda_{A,L} \rangle, \tag{2.19}$$

де $V_{A,L} = \{A_i^0 |_{X_i}^{Y_i}\}$ – множина основних алгоритмів внутрішнього виконавця конструювання.

Нижче представлені алгоритми, що реалізують операції над атрибутами:

$$\begin{aligned}
&A_1 |_{terminal}^{terminal},\\
&A_2 |_{session, startDebuggingSessionEvent}^{session},\\
&A_3 |_{session, debuggingEvent}^{session},\\
&A_4 |_{session, endDebuggingSessionEvent}^{session},\\
&A_5 |_{developerId, projectId, log}^{log},\\
&A_6 |_{debuggingEvent, session, log}^{log},\\
&A_7 |_{log}^{log},
\end{aligned} \tag{2.20}$$

де A_i ідентифікатор алгоритму, X_i, Y_i множини визначень і значень алгоритму $A_i^0|_{X_i}^{Y_i}$.

Інтерпретуємо конструктор:

$$\begin{aligned}
&\langle C_L = \langle M_L, \Sigma_L, \Lambda_L \rangle, \\
&C_{A,L} = \langle M_{A,L}, V_{A,L}, \Sigma_{A,L}, \Lambda_{A,L} \rangle \rangle_I \mapsto C_{I,L} = \langle M_{I,L}, \Sigma_{I,L}, \Lambda_{I,L} \rangle, \\
&\Lambda_{I,L} = \Lambda_L \bigcup \{ \\
&(A_1|_{terminal}^{terminal} \hookleftarrow \circ), \\
&(A_2|_{session,startDebuggingSessionEvent}^{session} \hookleftarrow \prec), \\
&(A_3|_{session,debuggingEvent}^{session} \hookleftarrow \succ), \\
&(A_4|_{session,endDebuggingSessionEvent}^{session} \hookleftarrow \cong), \\
&(A_5|_{developerId,projectId,log}^{log} \hookleftarrow \lhd), \\
&(A_6|_{debuggingEvent,session,log}^{log} \hookleftarrow \rhd), \\
&(A_7|_{log}^{log} \hookleftarrow \approx) \},
\end{aligned} \tag{2.21}$$

де $\mapsto_I$ – операція інтерпретації.

Проведемо конкретизацію конструктора C_L:

$$C_{I,L} = \langle M_{I,L}, \Sigma_{I,L}, \Lambda_{I,L} \rangle_K \mapsto C_L = \langle M_{K,L}, \Sigma_{K,L}, \Lambda_{K,L} \rangle, \tag{2.22}$$

де $\mapsto_K$ – операція конкретизації.

Правила підстановки наведені нижче (2.23) – (2.26).

Правило s_1 застосовується, коли подія *startDebuggingSessionEvent* отримана від зовнішнього виконавця. В результаті виконання правила створюється сеанс налагодження *session*

$$\begin{aligned}
&s_1 = \langle \sigma \to startDebuggingSessionEvent \bullet \alpha) \rangle, \\
&g_1 = \langle \circ(startDebuggingSessionEvent), \\
&\prec(startDebuggingSessionEvent, session) \rangle.
\end{aligned} \tag{2.23}$$

Правило s_2 застосовується, коли відбулася подія *debuggingEvent*. В результаті подія *debuggingEvent* додається до сесії налагодження *session*

$$
\begin{aligned}
&s_2 = \langle \alpha \rightarrow debuggingEvent \bullet \alpha \rangle, \\
&g_2 = \langle \circ(debuggingEvent), \\
&\succ (session, debuggingEvent) \rangle.
\end{aligned} \tag{2.24}
$$

Правило s_3 застосовується, коли відбулася подія $endDebuggingSessionEvent$. В результаті завершується сеанс налагодження $session$

$$
\begin{aligned}
&s_3 = \langle \alpha \rightarrow endDebuggingSessionEvent \bullet \beta \rangle, \\
&g_3 = \langle \circ(endDebuggingSessionEvent), \\
&\cong (session, endDebuggingSessionEvent) \rangle.
\end{aligned} \tag{2.25}
$$

В результаті виконання правила s_4, створюється файл журналу подій log. Файл буде заповнено подіями з минулих сеансів налагодження розробника для активного проєкту. Якщо файл порожній тоді буде формуватися з подій поточного сеансу налагодження

$$
\begin{aligned}
&s_4 = \langle \beta \rightarrow \log \rangle, \\
&g_4 = \langle \lhd (id{\hookleftarrow}developer{\hookleftarrow}session, id{\hookleftarrow}project{\hookleftarrow}session, \log), \\
&\rhd (debugEvent, session, \log) \rangle.
\end{aligned} \tag{2.26}
$$

В результаті формується файл журналу подій налагодження.

Висновки до другого розділу

За допомогою інструментів конструктивно-продукційного моделювання формалізовано процеси розробки та налагодження програмного забезпечення.

Розроблено три конструктори. Перший, створює модель процесу розробки програмного забезпечення під час використання середовища розробки для автоматичного аналізу. Другий, класифікує історію розробки програми за дрібними змінами та створює файл журналів подій з метою вивчення процесу розробки студентів. Третій, формує журнал налагоджувальних дій.

Процес налагодження тексту програми розглядається як послідовність дій при роботі з інструментами в середовищі розробки.

Для аналізу сформованих журналів подій використовуються методи Process Mining. Інформація, витягнута з процесу розробки програми, може бути використана для автоматичного надання студенту рекомендацій щодо вдосконалення коду та відстеження тенденції його зміни. Аналіз історії розробки програми та налагодження показує, скільки часу зайняла розробка програми та його налагодження, які частини викликали найбільші труднощі.

На основі конструктивних моделей розроблено розширення до середовища розробки Visual Studio для автоматичного моніторингу та візуалізації процесів розробки та налагодження програмного забезпечення.

За матеріалами розділу опубліковано роботи [109–112, 122, 123, 129].

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ПРОЦЕСІВ РОЗРОБКИ ТА НАЛАГОДЖЕННЯ ПРОГРАМ

3.1 Мета та задача експериментального дослідження

Метою експериментального дослідження є:

– перевірка можливості застосування розроблених конструктивно-продукційних моделей;

– перевірка можливості застосування розроблених інструментальних засобів для відстеження процесів розробки та налагодження програмного забезпечення;

– виявлення особливостей поведінки під час розробки та налагодження програм в середовищі розробки;

– виявлення найбільш складних помилок для виправлення.

Задачею дослідження є перевірка коректності відстеження та аналізу дій під час розробки програми та налагодження в середовищі розробки.

План експериментального дослідження включає:

– розробку програм із заздалегідь визначеними помилками на основі досліджень про найбільш поширені помилки початківців [54, 55, 85];

– проведення олімпіади з налагодження, під час якої за допомогою розроблених інструментів відстежувати та фіксувати дії учасників в середовищі розробки;

– формування за допомогою методів Process Mining моделей процесів налагодження на основі сформованих журналів подій;

– порівняння процесів за допомогою методів та інструментів Process Mining;

– оцінку моделей процесів налагодження.

3.2 Розробка завдань для виявлення навичок налагодження

Наша мета експериментально вивчити процес налагодження, студентами поширених логічних помилок. Логічна помилка виникає, коли код синтаксично правильний, але програма виконується неправильно. Цей тип помилок, як правило, досить складний і студентам важче знайти та виправити його.

Систематичне знайомство студентів з різними типами помилок може значно поліпшити навички налагодження. Крім того, ізольоване виявлення окремих помилок спрощує пошук і усунення проблеми. Студентам пропонується визначити помилку та усунути її в установлений термін. Аналізуючи стратегії усунення помилок, ми можемо виявити шаблони, які впливають або пояснюють поведінку при налагодженні і які необхідно враховувати при навчанні.

Для збору та аналізу поведінки під час налагодження використовується розширення до Microsoft Visual Studio IDE, яке відстежує всі дії при роботі з середовищем розробки.

Мова C++ викладається на нашій кафедрі як перша мова програмування, тому саме вона була обрана.

Для експерименту ми вивчили п’ятнадцять різних типів помилок, наведених у таблиці 3.1.

Типи обрані тому, що вони часто зустрічаються серед студентів та є особливо важкими для виявлення програмістами початківцями на основі попередніх досліджень [54, 55, 85].

Таблиця 3.1 – Типи логічних помилок використаних в експерименті

Номер типу логічної помилки	Короткий опис логічної помилки
1	Спроба встановити значення змінній більше за максимально допустиме значення для відповідного типу
2	Ділення цілого числа на ціле число, тому результат округляється
3	Інкремент не значення, а вказівника
4	Неправильне використання функції sizeof
5	Помилка в написанні
6	Помилка при перестановці значень
7	Помилка паралелізму
8	Неправильне використання оператора switch, відсутність оператора break
9	Зайва крапка з комою
10	Неправильна послідовність арифметичних операцій
11	Відсутність фігурних дужок
12	Нерозуміння області дії змінних
13	Відсутність ініціалізації
14	Використання «=», коли мається на увазі «==»
15	Помилка з граничними умовами

Перший тип помилки виникає коли програма намагається записати до змінної значення, яке більше за максимальне значення визначеного типу. Приклад такої помилки наведено у лістингу 3.1.

Лістинг 3.1 – Приклад програми з логічною помилкою першого типу, відповідно до таблиці 3.1:

```
int main()
{
    int n = 20, factorial = 1, i = 1;

    while (i <= n)
    {
        factorial = factorial * i;
        i++;
    }

    return 0;
}
```

Другий тип помилки виникає коли обидва операнда є цілими числами та результат ділення дає дробовий результат, але без оператора приведення до типу з плаваючою комою. Тому дробова частина буде відкинута. Приклад такої помилки наведено у лістингу 3.2.

Лістинг 3.2 – Приклад програми з логічною помилкою другого типу, відповідно до таблиці 3.1:

```
int main()
{
    double half = 1/2; // 0, а не 0.5

    return 0;
}
```

Третій тип помилки виникає коли програма намагається збільшити змінну, на яку посилається вказівник (**counter++;*), так як пріоритет операції «++» вище, ніж в операції «*» (розіменування вказівника). Фактично це просто інкремент вказівника. Щоб зробити код коректним, необхідно додати круглі дужки: (*(* counter) ++;*). Приклад такої помилки наведено у лістингу 3.3.

Лістинг 3.3 – Приклад програми з логічною помилкою третього типу, відповідно до таблиці 3.1:

```
int increment(int *counter)
{
  *counter++;

  return *counter;
}

int main()
{
    int counter = 1;

    counter = increment(&counter);

    return 0;
}
```

Четвертий тип помилки виникає коли намагаються використовувати *sizeof* для масиву переданого у якості параметра функції, адже в такому випадку отримуємо розмір вказівника, а не кількість елементів масиву. Приклад такої помилки наведено у лістингу 3.4.

Лістинг 3.4 – Приклад програми з логічною помилкою четвертого типу, відповідно до таблиці 3.1:

```
void print (int arr[])
{
    int length = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < length; i++)
    {
        cout << arr[i] << " ";
    }
}

int main()
{
    int arr[3] = { 0, 1, 2 };
    print(arr);

    return 0;
}
```

П’ятий тип помилки виникає коли помиляються в написанні операторів, наприклад «=+» замість «+=». Приклад такої помилки наведено у лістингу 3.5.

Лістинг 3.5 – Приклад програми з логічною помилкою п’ятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int counter = 1;

    counter =+ 1; // =+ замість +=

    return 0;
}
```

Шостий тип помилки виникає коли пропускають використання тимчасової змінної для заміни значень двох змінних, або роблять це неправильно. Приклад такої помилки наведено у лістингу 3.6.

Лістинг 3.6 – Приклад програми з логічною помилкою шостого типу, відповідно до таблиці 3.1:

```
int main()
{
    int a = 1, b = 2, temp = 0;

    temp = b; // b замість a
    a = b;
    b = temp;

    return 0;
}
```

Сьомий тип помилки виникає коли міняють місцями твердження програми. Деякі студенти не до кінця розуміють послідовний характер кроків в програмі. Приклад такої помилки наведено у лістингу 3.7. У цій помилці ми поміняли місцями два твердження програми. Перший оператор викликав сортування, а другий друкував відсортований масив. Очевидно, що в результаті на екран виводиться невідсортований масив.

Лістинг 3.7 – Приклад програми з логічною помилкою сьомого типу, відповідно до таблиці 3.1:

```
int main()
{
    const int n = 5;
    int arr[n] = { 4, 3, 2, 1, 0 };

    Output(arr, n);
    Sort(n, arr);

    return 0;
}
```

Восьмий тип помилки виникає коли забувають оператор *break* в операторі *switch*, що призводить до некоректного виконання програми. Приклад такої помилки наведено у лістингу 3.8.

Лістинг 3.8 – Приклад програми з логічною помилкою восьмого типу, відповідно до таблиці 3.1:

```
int main()
{
    char operation = '*';
    float a = 1.0, b = 3.0, result = 0;

    switch (operation)
    {
        case '+':
            result = a + b;
            break;
        case '*':
            result = a * b; // не вистачає оператора break;
        case '/':
            result = a / b;
            break;
    }

    return 0;
}
```

Дев’ятий тип помилки виникає коли додають зайву крапку з комою. Варіантів цієї помилки може бути дуже багато. Приклад такої помилки наведено у

лістингу 3.9. В даному прикладі цикл не має тіла і викликає неправильне функціонування програми.

Лістинг 3.9 – Приклад програми з логічною помилкою дев'ятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int arr[3] = { 0, 1, 2 };
    int length = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < length; i++); // зайва крапка з комою
    {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Десятий тип помилки виникає коли програміст не розуміє пріоритети операцій і в результаті отримує помилкові значення обчислень. Приклад такої помилки наведено у лістингу 3.10.

Лістинг 3.10 – Приклад програми з логічною помилкою десятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int a = 55, sum = 0;
    int mod = a % 10;

    sum *= 10 + mod;

    return 0;
}
```

Одинадцятий тип помилки виникає коли забувають вказати фігурні дужки. В такому випадку лише перший рядок буде тілом умови або циклу. То ж, гарним стилем буде, завжди вказувати фігурні дужки, навіть якщо тіло складається лише з одного рядка. Приклад такої помилки наведено у лістингу 3.11.

Лістинг 3.11 – Приклад програми з логічною помилкою одинадцятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int n = 5, factorial = 1, i = 1;

    while (i <= n)
        factorial = factorial * i;
        i++; // не в межах циклу

    return 0;
}
```

Дванадцятий тип помилки виникає коли створюють декілька змінних з однією назвою в рамках однієї функції. Треба розуміти, що локальні змінні мають локальну область видимості, тобто вони входять в зону видимості з точки оголошення і виходять в самому кінці блоку, в якому визначені. Приклад такої помилки наведено у лістингу 3.12.

Лістинг 3.12 – Приклад програми з логічною помилкою дванадцятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int i = 0;

    for (int i = 0; i < 5; i++)
    {
        ;
    }

    cout << i; // 0

    return 0;
}
```

Тринадцятий тип помилки виникає коли забувають ініціалізувати змінну перед тим як її використовувати, що призводить до виключних ситуацій під час виконання. Приклад такої помилки наведено у лістингу 3.13.

Лістинг 3.13 – Приклад програми з логічною помилкою тринадцятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int counter;

    while (counter < 10)
    {
        ;
    }

    return 0;
}
```

Чотирнадцятий тип помилки виникає при використанні одинарного знака рівності «=» для перевірки рівності в умовах замість подвійного знака рівності «==». Таким чином, програма присвоїть значення правої частини виразу змінній в лівій частині, і результатом цього твердження завжди буде істинним. Приклад такої помилки наведено у лістингу 3.14.

Лістинг 3.14 – Приклад програми з логічною помилкою чотирнадцятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int counter;

    while (counter < 10)
    {
        ;
    }

    return 0;
}
```

П’ятнадцятий тип помилки виникає коли програміст виконує ітерацію циклу більшу або меншу кількість разів, ніж очікується. Таким чином, програма читає або записує дані за межею виділеної пам’яті, що може привести до пошкодження даних або аварійного завершення роботи програми. Приклад такої помилки наведено у лістингу 3.15.

Лістинг 3.15 – Приклад програми з логічною помилкою п’ятнадцятого типу, відповідно до таблиці 3.1:

```
int main()
{
    int counter = 1;

    if (counter = 10) // ‘=’ замість ‘==’
    {
        ;
    }

    return 0;
}
```

Для проведення олімпіади з налагодження було розроблено тридцять коротких програм, кожна з яких включає один тип з перерахованих в таблиці 3.1 помилок. Для кожної програми логічна помилка проявляється одним рядком коду. Тобто помилка може бути виправлена шляхом зміни тільки одного рядка коду. Всі програми синтаксично коректні, але виконання кожної з них призводить до некоректного результату. Щоб допомогти виявити всі помилки, студентам надається зразок вхідних та вихідних даних. Всього студентам треба було виконати п’ятнадцять завдань за 4 години.

3.3 Формування моделей процесів розробки та налагодження

Поточне дослідження застосовує методи Process Mining в області програмної інженерії для виявлення навичок розробки та налагодження програмного забезпечення. Ґрунтуючись на методах Process Mining, аналізуючи дані про використання середовища розробки, ми прагнемо отримати шаблони поведінки програмістів.

3.3.1 Застосування методів Process Mining до процесу розробки програмного забезпечення

Журнал подій є відправною точкою для Process Mining. Кожна подія в такому журналі належить діям, які можуть бути виконані на ресурсі в певний час і для певного випадку. Журнал подій структурований як набір записів, де кожна із записів є ланцюжком дій, створених в результаті одного виконання процесу (випадку). Як мінімум, запис події включає ідентифікатор процесу, до якого відноситься подія, час і безліч додаткових атрибутів. Опис кожного атрибута, що зберігається в журналі подій для процесу розробки програмного забезпечення, наведено в таблиці 3.2.

Таблиця 3.2 – Опис атрибутів, що зберігаються у файлі журналу подій

Атрибут	Рівень	Опис
Name	Trace	Ідентифікатор сеансу
StartedDateTime	Trace	Час початку сеансу
FinishedDateTime	Trace	Час завершення сеансу
Project	Trace	Ідентифікатор проєкту
Developer	Trace	Ідентифікатор розробника
Activity	Event	Тип дрібномасштабної зміни
Timestamp	Event	Час коли відбулася подія
Context	Event	Контекст події

Файл журналу подій зберігається у форматі eXtensible Event Stream [130], який є стандартним форматом для Process Mining, розробленим IEEE Task Force для реєстрації подій. Файл містить класифіковані дрібномасштабні зміни коду відповідно до типів з таблиці 2.1.

Методи виявлення (discovery) Process Mining використовується для формування моделі процесу розробки програми. Формування моделі процесу дозволяє отримати спосіб і порядок виконання процесу. Для виявлення моделі процесу розробки з журналу подій використовується ProM. ProM – це широко

поширений фреймворк з відкритим кодом, який є де-факто стандартом, для реалізації Process Mining [131].

Для виявлення моделі процесу розробки в ProM використовується Inductive Visual Miner [132], який дозволяє використовувати графіки прямого слідування. Графіки прямого слідування представляють дії у вигляді прямокутників і пов'язують дві дії разом, якщо одна з них безпосередньо слідує за іншою. Крім того, кожне ребро має значення, яке вказує на кількість записів в журналі подій.

3.3.1.1 Ілюстративний приклад

Завдання полягає в тому, щоб написати програму для розрахунку мінімальної кількості монет, необхідних для видачі здачі покупцеві. У якості вхідних даних програма приймає масив номіналів монет і необхідну кількість здачі. Результат роботи показаний в лістингу 3.16.

Лістинг 3.16 – Програма студента для вирішення завдання

```
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; } } }
```

Оцінка результатів навчання студента відіграє важливу роль в освіті та зазвичай здійснюється шляхом оцінки підсумкових результатів студента. Тепер, на основі цієї програми, викладач повинен зрозуміти та оцінити навички та вміння програміста. Це досить складно зробити, оцінюючи тільки кінцевий результат. Тому пропонується використовувати для оцінки роботи студента не тільки код програми, а й процес його розробки.

У попередньому розділі описано конструктор для формування історії процесу розробки програми. В результаті формується масив ланцюжків, який повністю показує історію процесу кодування. Історія процесу розробки програми з лістингу 3.16 представлена в лістингу 3.17.

Історія процесу розробки програми представлена у вигляді послідовності ланцюжків, кожен з яких містить наступну інформацію:

– порядковий номер;

– тип зміни («+» – додати, «-» – видалити, «*» – редагувати);

– код;

– ім’я файлу;

– номер рядка в остаточній версії програми або негативний ідентифікатор;

– час коли були зроблені зміни.

Лістинг 3.17 – Історія процесу розробки тексту програми

```
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
```

Конструктор, представлений в попередньому розділі, класифікує історію процесу кодування по дрібномасштабним змінам та створює файл журналу подій, призначений для вивчення процесу розробки програми студентами. Файл містить послідовність подій відповідно до типів змін наведених в таблиці 2.1.

Фрагмент журналу подій у форматі eXtensible Event Stream наведено в рисунку 3.1.

```
<trace>
    <string key="developer" value="John Doe"/>
    <string key="project" value="IllustrativeApp"/>
    <date key="startedDateTime" value="2020-11-22T10:35:00.000+02:00"/>
    <date key="finishedDateTime" value="2020-11-22T12:47:59.000+02:00"/>
    <string key="concept:name" value="Session1"/>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:35:00.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Start"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:35:21.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Add variable"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:35:21.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Add variable assignment"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:36:21.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Add variable"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:36:21.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Add variable assignment"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-11-22T10:36:21.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Add variable"/>
    </event>
```

Рисунок 3.1 – Фрагмент журналу подій у форматі eXtensible Event Stream

Модель процесу розробки програми з лістингу 3.16 наведено на рисунку 3.2.

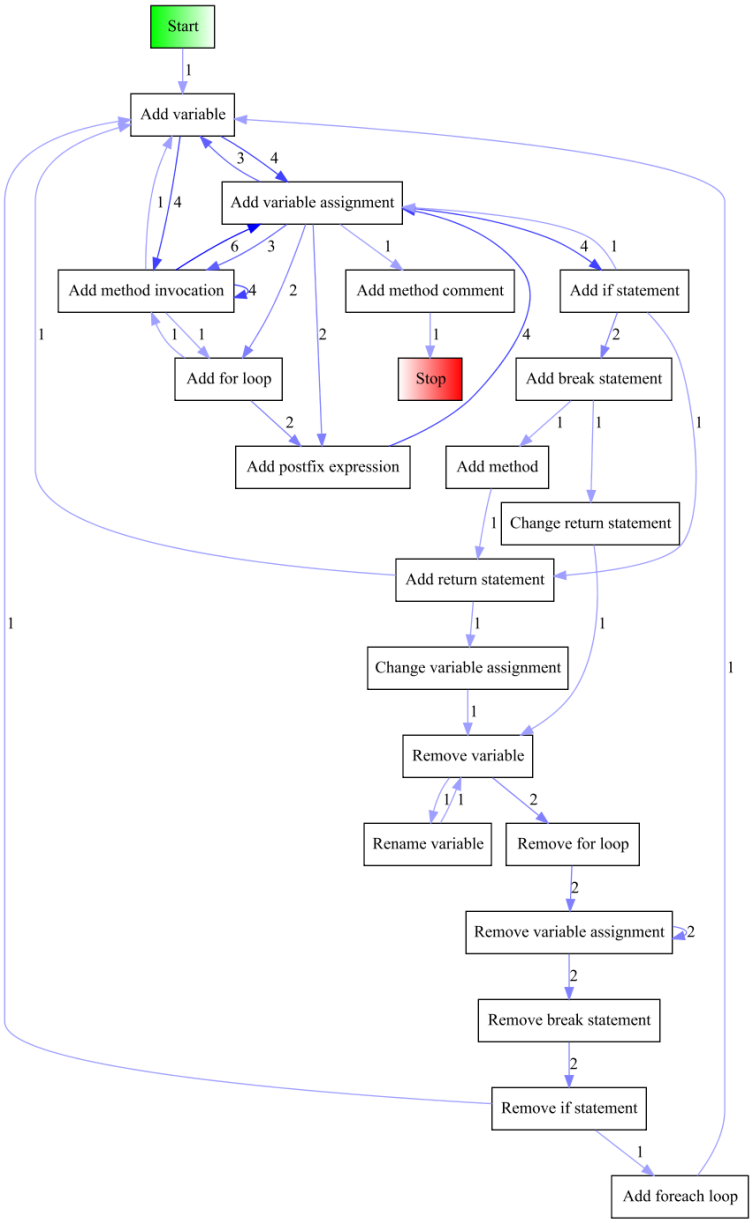


Рисунок 3.2 – Модель процесу розробки програми, сформована за допомогою Inductive Visual Miner

Аналіз історії розробки програми дає інформацію про те, скільки часу знадобилося студенту, які частини викликали найбільші труднощі.

Аналіз програми (лістинг 3.16) і історії її розробки (лістинг 3.17) показує, що в процесі виконання у програміста спочатку було інше рішення (ланцюжки 1–33), ніж те, яке представлено в результаті. В процесі тестування, переконавшись, що обране рішення недостатньо добре підходить, алгоритм був перероблений (ланцюжки 34–45). Також можна помітити, що спочатку весь код програми був всередині функції *main*, і тільки потім, за допомогою рефакторінга *Extract Method,*

логіка знаходження мінімальної кількості монет була винесена в окремий метод. Такий стиль порушує принципи методу покрокової деталізації, на що необхідно вказати студенту в результаті перевірки. Крім того, коментарі до методу були написані в самому кінці, що також є ознакою "поганого" стилю. Можна виділити й позитивні моменти, наприклад, стиль іменування змінних і методів. Імена були придумані відразу, а не в кінці, коли програма вже майже завершена.

3.3.2 Застосування методів Process Mining до процесу налагодження програмного забезпечення

Відправною точкою Process Mining є журнал подій. Опис атрибутів журналу подій для вивчення процесу налагодження програмного забезпечення наведено в таблиці 3.3.

Таблиця 3.3 – Опис атрибутів, що зберігаються у файлі журналу подій

Атрибут	Рівень	Опис
Name	Trace	Ідентифікатор сеансу налагодження
StartedDateTime	Trace	Час початку сеансу
FinishedDateTime	Trace	Час завершення сеансу
Project	Trace	Ідентифікатор проєкту
Developer	Trace	Ідентифікатор розробника
Activity	Event	Тип події
Timestamp	Event	Час коли відбулася подія
Context	Event	Контекст події
Resource	Event	Шлях до файлу, який налагоджується
LineNumber	Event	Номер рядка, з яким пов’язана подія

Перший метод Process Mining – це виявлення (discovery). Метод виявлення на основі журналу подій, що складається із запису всіх дій, що відбуваються під час процесу налагодження програмного забезпечення, формує модель, яка

представляє, як і в якому порядку виконувався процес. ProM використовується для виявлення моделі процесу налагодження з журналу подій.

Фрагмент журналу подій для виявлення моделі процесу налагодження наведено на рисунку 3.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
 <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext" />
 <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.xesext" />
 <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext" />
 <classifier name="Activity" keys="concept:name" />
 <string key="concept:name" value="I Doe Task002.xes" />
 <trace>
  <string key="developer" value="I Doe" />
  <string key="project" value="Task002" />
  <date key="finishedDateTime" value="2021-05-21T09:02:52.5218810+03:00" />
  <date key="startedDateTime" value="2021-05-21T09:02:13.0078042+03:00" />
  <string key="concept:name" value="I Doe Task002 Session1" />
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:13.0078042+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="Start" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:13.0078042+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="DocumentOpened" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:40.4171154+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="BuildBegin" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:44.7372113+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="BuildDone" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:47.8993203+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="DebugStart" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:52.5218810+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="DebugEnd" />
  </event>
  <event>
   <date key="time:timestamp" value="2021-05-21T09:02:52.5218810+03:00" />
   <string key="lifecycle:transition" value="complete" />
   <string key="concept:name" value="Finish" />
  </event>
 </trace>
</log>
```

Рисунок 3.3 – Фрагмент журналу подій у форматі eXtensible Event Stream

Журнал подій п’яти сеансів налагодження в форматі eXtensible Event Stream, які були відстежені з використанням розширення до середовища розробки в процесі налагодження програмного забезпечення імпортовані в ProM, щоб детально показати, як відбувався процес налагодження. На рис. 3.4 представлені послідовності дій, де кожна послідовність є одним сеансом налагодження.

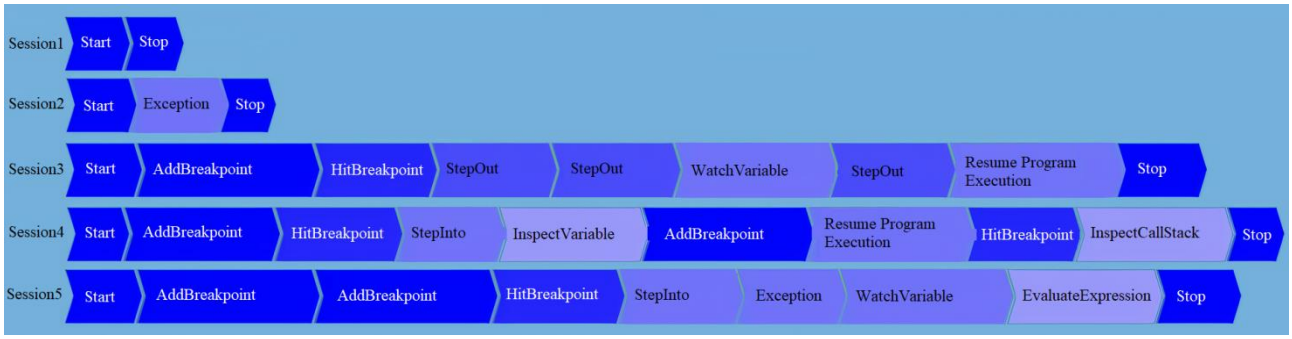


Рисунок 3.4 – Сеанси налагодження

Нижче представлені конструкції представлення процесу налагодження, що забезпечують однозначну відповідність представлення та процесу для перших двох сеансів (формули 3.1–3.2).

$$\begin{aligned} &\sigma \overset{s_1}{\Rightarrow} startDebuggingSessionEvent \bullet \alpha \overset{s_3}{\Rightarrow} \\ &startDebuggingSessionEvent \bullet endDebuggingSessionEvent \bullet \beta \overset{s_4}{\Rightarrow} \\ &startDebuggingSessionEvent \bullet endDebuggingSessionEvent \bullet \log, \end{aligned} \tag{3.1}$$

$$\begin{aligned} &\sigma \overset{s_1}{\Rightarrow} startDebuggingSessionEvent \bullet \alpha \overset{s_2}{\Rightarrow} \\ &startDebuggingSessionEvent \bullet debuggingEvent \bullet \alpha \overset{s_3}{\Rightarrow} \\ &startDebuggingSessionEvent \bullet debuggingEvent \bullet \\ &endDebuggingSessionEvent \bullet \beta \overset{s_4}{\Rightarrow} \\ &startDebuggingSessionEvent \bullet debuggingEvent \bullet \\ &endDebuggingSessionEvent \bullet \log. \end{aligned} \tag{3.2}$$

Для виявлення моделі процесу налагодження в ProM використовується Inductive Visual Miner [132]. Модель процесу налагодження наведено на рисунку 3.5.

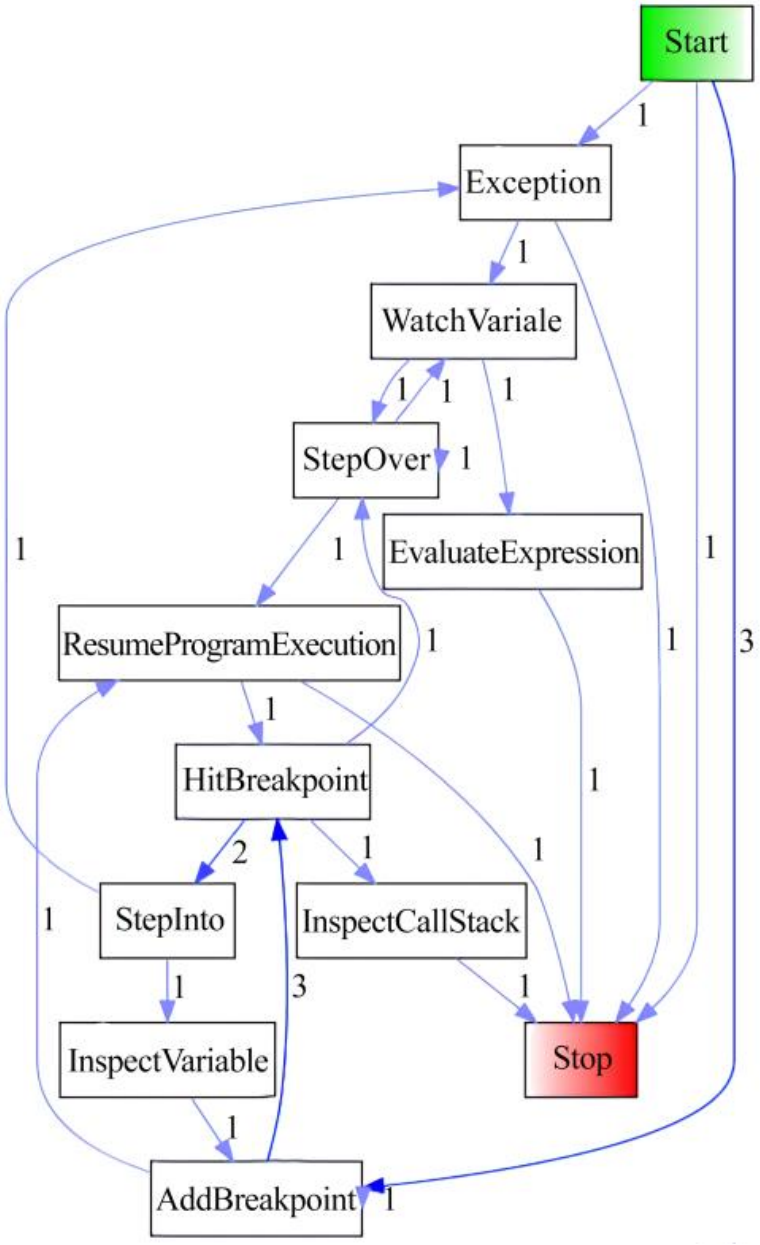


Рисунок 3.5 – Модель процесу налагодження

Другим методом Process Mining є перевірка відповідності (Conformance Checking).

Перевірка відповідності дозволяє порівняти різні реалізації процесів і виявити поведінкові схожості та відмінності. У дослідженні ми проводимо перевірку відповідності двічі: коли виявлена модель відтворюється в журналі подій і при вимірі відхилення між визначеною моделлю і виявленою моделлю.

Для перевірки відповідності використовується ознака придатності [96]. Модель має відмінну придатність, якщо всі послідовності можуть бути відтворені

моделлю від початку до кінця. Придатність характеризується числом від 0 (дуже погано) до 1 (відмінно).

Визначена модель показує, які переходи між подіями можливі (рис. 3.6):

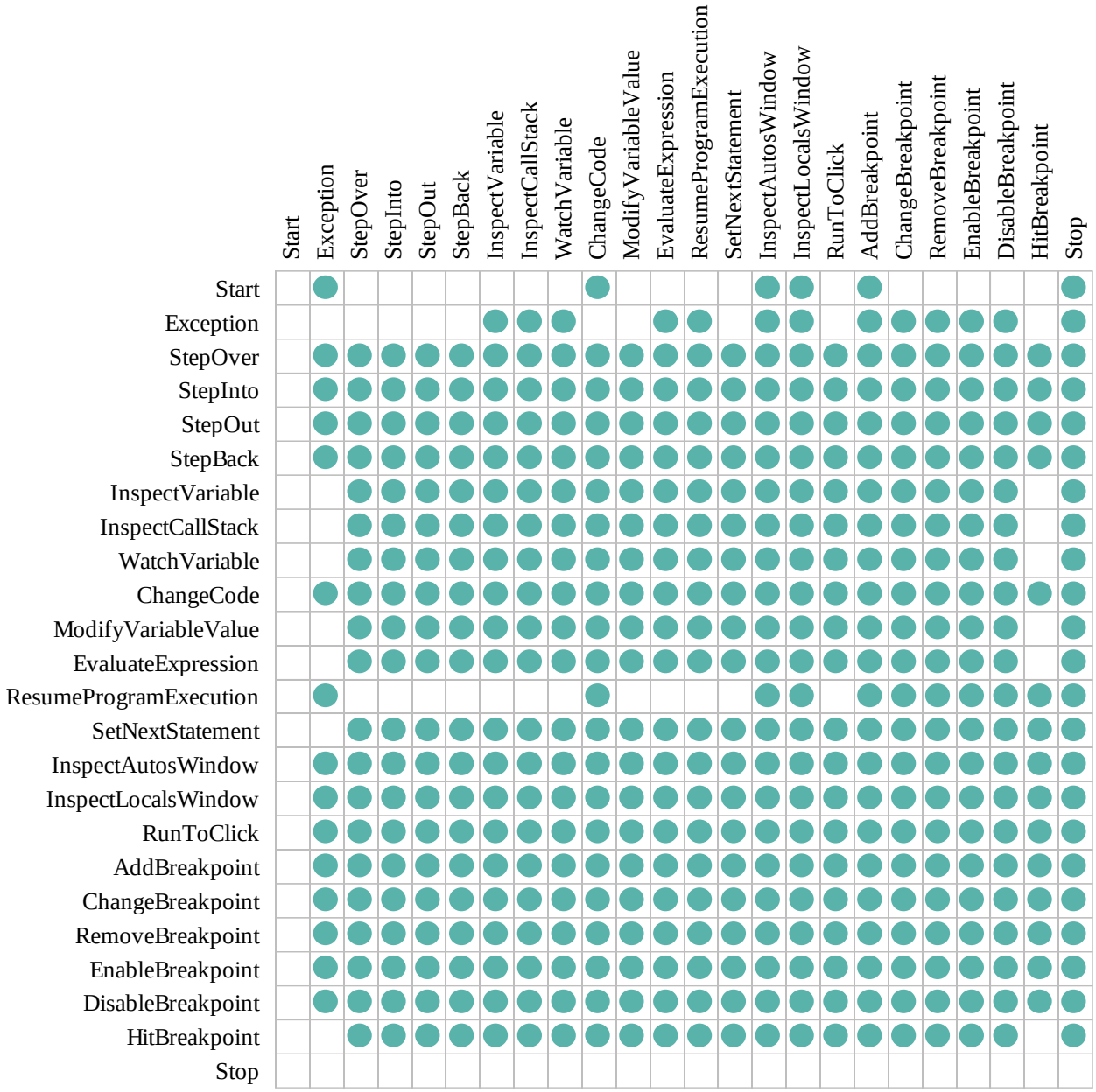


Рисунок 3.6 – Допустимі переходи між подіями

Інструментарій ProM містить плагіни, що дозволяють розраховувати придатність по моделі та по файлу журналу подій, а також придатність за двома моделями. Перевірка відповідності виконується за допомогою плагіна Visualize deviations ProM [133]. Придатність файлу журналу подій, сформованого за допомогою розширення до Visual Studio, стосовно результуючої моделі дорівнює одиниці, що підтверджує надійність і адекватність результуючої моделі.

3.4 Аналіз результатів експериментів

У експерименті взяв участь 41 студент з першого по п’ятий курси. Всі вони мали попередній досвід розробки на C подібних мовах програмування і використання середовища розробки Visual Studio. Всього студентам треба було виконати п’ятнадцять завдань за 4 години.

На основі кількості правильно виконаних завдань учасники були класифіковані як "High", "Middle", "Low". Студенти відносяться до класу "Low", якщо вони виконали менше половини завдань, таких – 9. Студенти з "Middle" класу виконали більше половини завдань, але не всі, таких – 23. Студенти з "High" класу виконали всі завдання, таких – 9.

Тридцять два (~78 %) студента змогли вирішити половину або більше завдань. Дев’ять студентів виправили всі помилки. В середньому їм на це знадобилося трохи більше 2 годин. Жодна з категорій помилок не є повністю нерозв’язаною усіма студентами. Помилки з категорій 2, 8 та 14 вирішили усі студенти. Немає жодної категорії, яка б містила більше неправильних відповідей, ніж правильних.

Щодо того, які помилки є найбільш складними для студентів, результати показують, що більшість учасників здатні правильно відповісти на половину або більше завдань. Виходячи з кількості правильних відповідей, більшість помилок потрапляли в діапазон від 75 % правильних. Результати свідчать про те, що студенти мають помірні труднощі з категоріями помилок 4 (неправильне використання функції sizeof) та 12 (нерозуміння області дії змінних). 25 % завдань з цих категорій були вирішені неправильно.

В табл. 3.4 наведено перелік найбільш часто використовуваних команд середовища розробки Visual Studio учасниками експерименту.

Таблиця 3.4 – Частота використання команд під час експерименту

Номер	Команда	Частота	Відсоток використання серед учасників		
			High	Middle	Low
1	DebugStart	2415	15.368 %	14.426 %	14.059 %
2	DebugEnd	2415	15.368 %	14.426 %	14.059 %
3	BuildDone	2415	15.368 %	14.426 %	14.059 %
4	BuildBegin	1433	8.893 %	8.195 %	9.855 %
5	CodeChanged	1291	7.86 %	7.074 %	10.096 %
6	BreakpointHitted	323	0.821 %	2.433 %	2.068 %
7	DocumentOpened	248	1.431 %	1.718 %	0.896 %
8	Edit.Undo	230	0.845 %	1.12 %	3.067 %
9	BreakpointAdd	168	0.375 %	1.472 %	0.482 %
10	BreakpointRemove	160	0.634 %	1.344 %	0.241 %
11	Edit.Paste	114	0.446 %	0.896 %	0.379 %
12	StepOver	101	0.211 %	0.704 %	0.896 %
13	ExceptionThrown	90	0.634 %	0.63 %	0.138 %
14	StepInto	85	0 %	0.587 %	1.034 %
15	Edit.Copy	74	0.352 %	0.566 %	0.207 %

Відносно невеликий набір команд зустрічається майже у всіх учасників, а саме дев'ять команд зустрічаються у 90 % або більше студентів. Це загальний набір команд: запуск і зупинка налагоджувача, зміна коду, виконання програми, відкриття файлів з текстом програми, а також копіювання та вставка. Більшість інших команд зустрічаються у меншого кола розробників. До них відносяться команди такі як установка, видалення і зупинка на точках зупинки, покрокове виконання, відміна зроблених змін, а також події виключних ситуацій. Більшість з цих команд мають широке застосування і повинні бути рекомендовані іншим розробникам.

У табл. 3.5 наведено статистику практик налагодження, які використовувалися студентами під час експерименту.

Таблиця 3.5 – Статистика використання практик налагодження

Практика	Відсоток сесій в яких застосовано практику		
	High	Middle	Low
Зміна коду	43.82 %	37.13 %	57.84 %
Покрокове виконання	1.37 %	8.58 %	12.75 %
Точки зупину	5.34 %	17.31 %	14.71 %
Покрокове виконання серед сесій з точками зупину	25.71 %	49.57 %	86.67 %
Виведення на екран	4.83 %	7.33 %	17.41 %

Як можна побачити учасники з класу "Low" роблять набагато більше змін коду ніж інші при цьому у них значно менше сесій налагодження. Ще одна виразна особливість учасників класу "Low" це частіше використання практики виведення на екран значень змінних аніж встановлення точок зупину чи покрокове виконання.

Також в табл. 3.5 можна помітити два типи використання точок зупину. По-перше, для перевірки чи виконується частина коду, а по-друге, для покрокового виконання програми. При цьому лише учасники класу "Low" в переважній більшості використовують точки зупину для покрокового виконання (86.67 %). Учасники з класу "High" навпаки лише у 25.71 % сесій використовували таким чином. Учасники з класу "High" взагалі майже не використовували практики встановлення точок зупину та покрокового виконання. Це може вказувати на те, що їх навичок розробника достатньо щоб одразу робити правильні гіпотези, а іншим необхідно використовувати налагодження для вивчення поведінки програми.

Під час експерименту було сформовано 487 файлів журналів подій для 41 учасника. Загалом журнали подій складаються з 2415 сесій налагодження та 16536 подій.

Щоб зрозуміти, що сталося в кожному класі учасників, сформували відповідні моделі процесів на основі журналів подій (фаза виявлення процесу). Зокрема, Directly Follow Graph плагін фреймворку ProM (метод Process Mining) приймає на вхід журнали учасників з "High", "Middle", "Low" класів та створює відповідні моделі процесів, у вигляді DFG.

На рис. 3.7 зображено загальну модель процесу налагодження комп’ютерних програм учасниками експерименту.

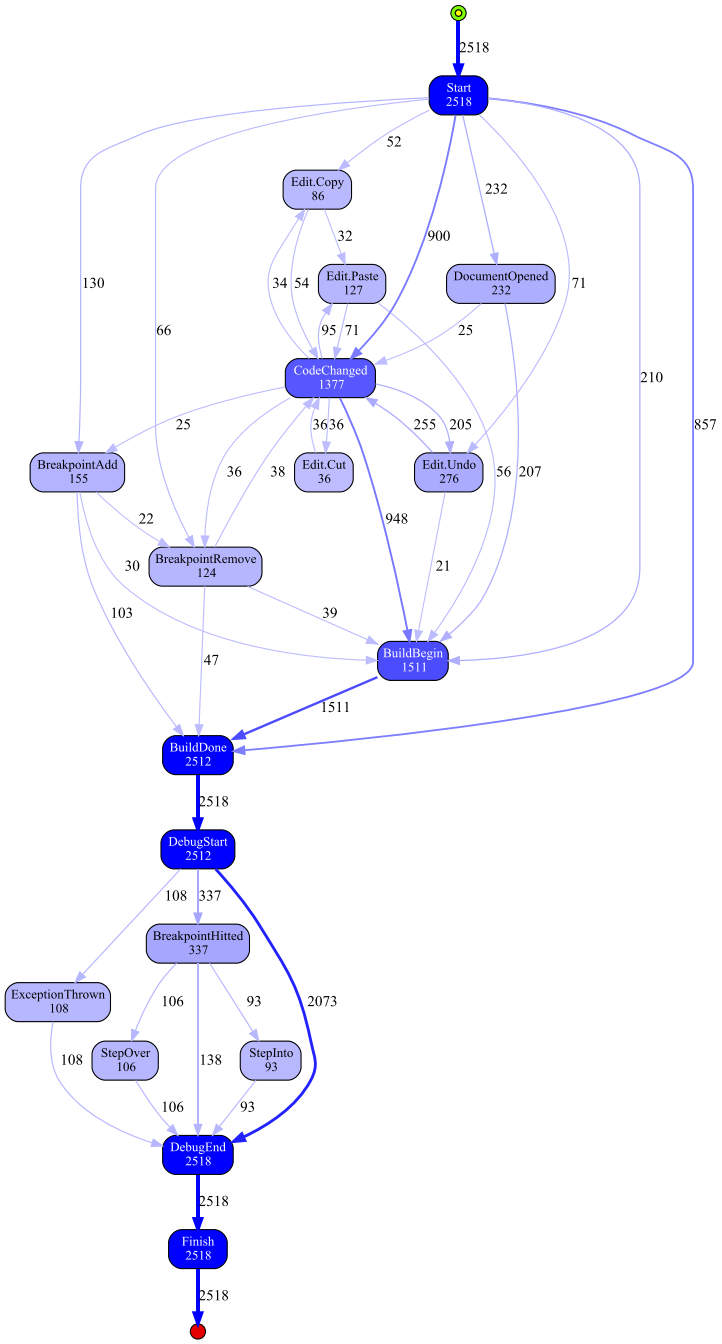


Рисунок 3.7 – Загальна модель процесу налагодження у вигляді Directly Follows Graph

Сформовано один загальний файл журналу подій на основі усіх сесій налагодження. Задля того щоб модель була інформативною було проведено фільтрацію засобами ProM. В результаті модель сформовано на основі файлу журналу подій, який складається з 90 % подій, які були відстежені з середовища розробки Visual Studio під час експерименту. Модель відображає команди, які були використані під час налагодження та зв'язки між ними.

На основі отриманих моделей процесів налагодження та найбільш часто повторювальних сесій вдалося виявити 4 шаблони поведінки учасників експерименту.

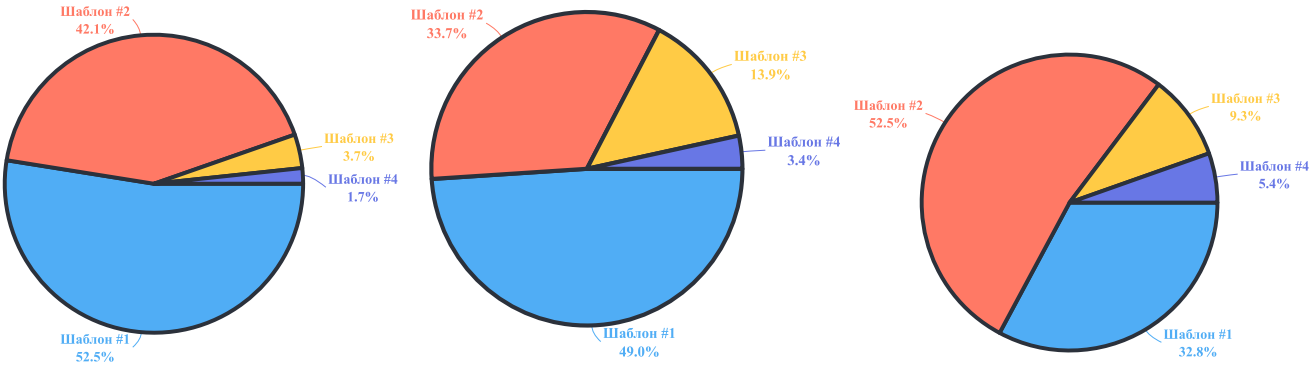
Шаблон #1 – виконання програми в режимі налагодження без будь-яких змін та налагоджувальних дій. Використовується, щоб перевірити поведінку програми.

Шаблон #2 – це зміна тексту програми й виконання в режимі налагодження без будь-яких змін та налагоджувальних дій. Цей шаблон відповідає застосуванню методу спроб та помилок.

Шаблон #3 – це виконання програми в режимі налагодження з встановленими точками зупину. Використовується, щоб дослідити поведінку програми, а саме які її частини виконуються. А також використовується для покрокового виконання.

Шаблон #4 – це зміна тексту програми, встановлення точок зупину й виконання в режимі налагодження. Цей шаблон відповідає застосуванню методу припущення про помилку. Коли робиться припущення про помилку, в цьому місці виконується зміна коду і встановлюються точки зупину для перевірки, що помилка виправлена.

На рис. 3.8 наведено частоту використання шаблонів різними класами учасників експерименту.



а *б* *в*

Рисунок 3.8 – Статистика використання шаблонів учасниками: а – клас "High"; б – клас "Middle"; в – клас "Low"

Як можна побачити на рис. 3.8 у всіх класах учасників переважає використання перших двох шаблонів, та дуже незначне використання шаблонів #3 та #4. Вагома різниця лише в тому, що в учасників з класу "Low" значно переважає використання методу спроб та помилок, а учасники класу "High" використовують шаблони #3 та #4 лише в 5.4 % сесій налагодження.

На рис. 3.9–3.12 наведено, як виявлені шаблони відображаються на отриманих засобами Process Mining моделях процесів налагодження.

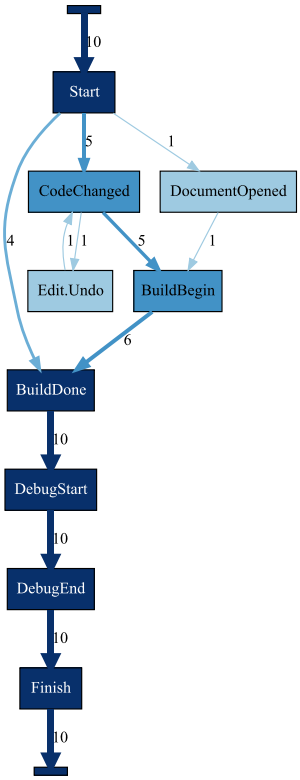


Рисунок 3.9 – Модель процесу налагодження із застосуванням методу спроб та помилок

Рис. 3.9 відображає модель застосування методу спроб та помилок, яке здійснюється через внесення змін (CodeChanged) та виконання програми без будь-яких налагоджувальних дій. Такий процес повторюється допоки не буде знайдено рішення або закінчаться ідеї. Такий метод застосовується усіма класами учасників і різниця лише в тому що учасникам класу "Low" треба більше спроб і вони мають гірший результат.

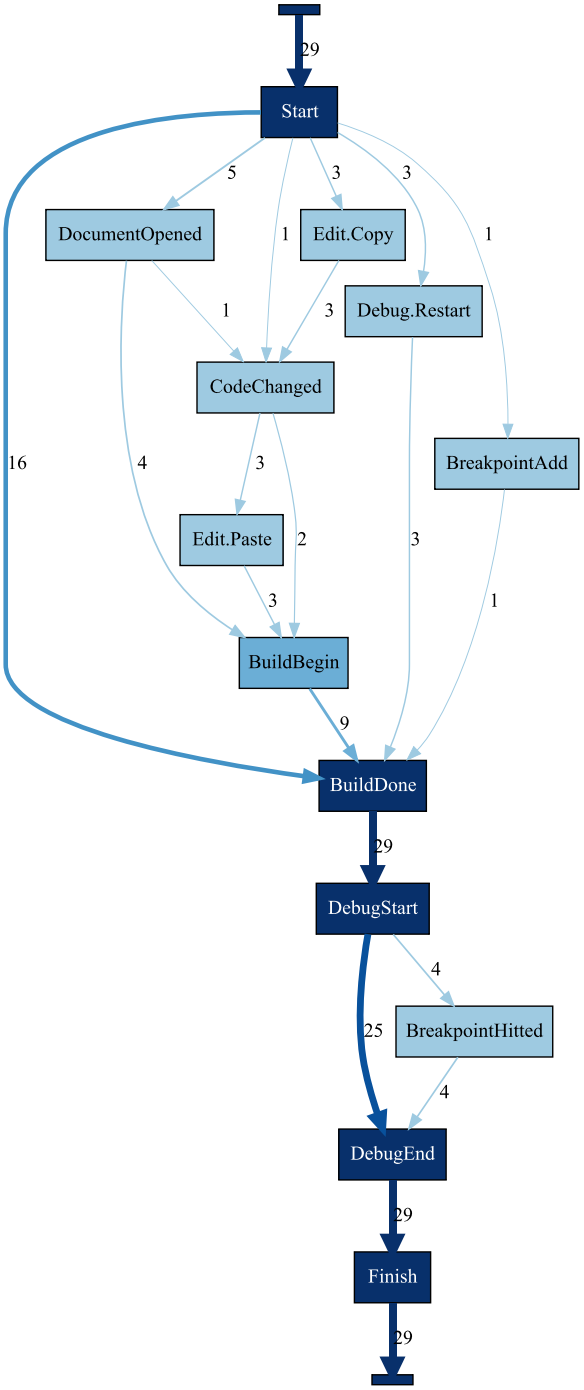


Рисунок 3.10 – Модель процесу налагодження із застосуванням точок зупину, але без покрокового виконання

Модель на рис. 3.10 відображає застосування шаблону #3 при налагодженні, а саме послідовність дій по виконанню програми без будь-яких змін з встановленими точками зупину. Коли точка зупину спрацює є декілька варіантів подальших дій. Перший, закінчити налагодження, в такому випадку мета була переконатися, що частина коду виконується. Другий, продовжити налагодження шляхом покрокового виконання. Як показали результати експерименту другий варіант використовується переважною більшістю учасників з класу "Low". Учасники з класу "Middle" однаково використовують обидва методи, а учасники класу "High" надають перевагу першому, адже вони здатні без покрокового виконання зрозуміти поведінку програми.

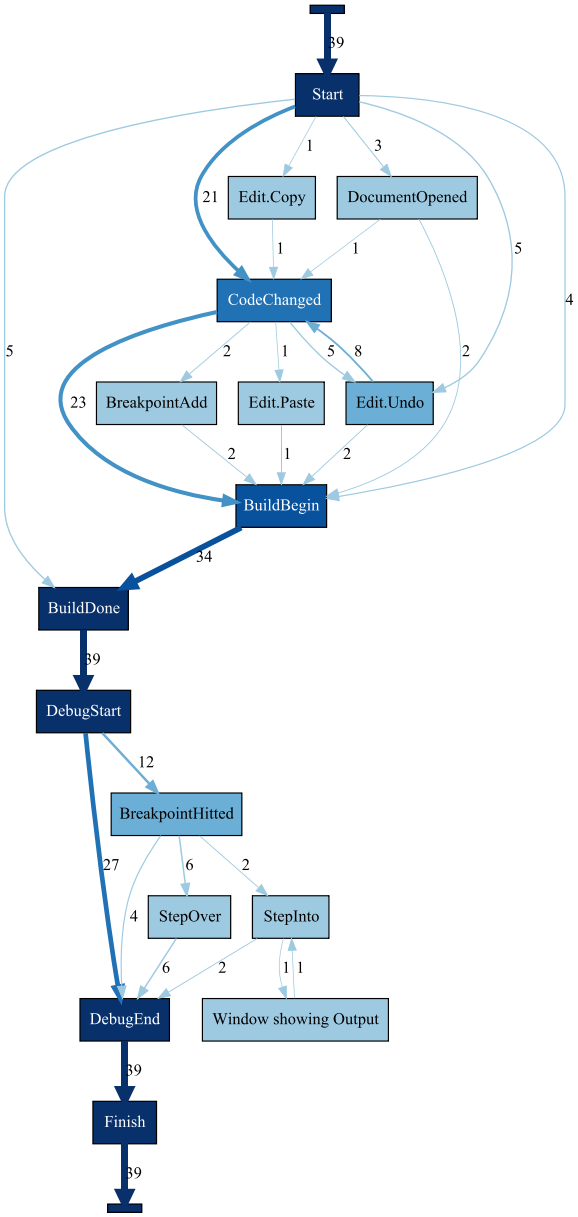


Рисунок 3.11 – Модель процесу налагодження застосовуючи шаблон #4

Модель на рис. 3.11 відображає застосування методу припущення про помилку. Коли програміст висуває гіпотезу, робить зміну тексту програми, щоб виправити помилку, після чого виконує програму з встановленими точками зупину і перевіряє шляхом покрокового виконання чи помилка виправлена.

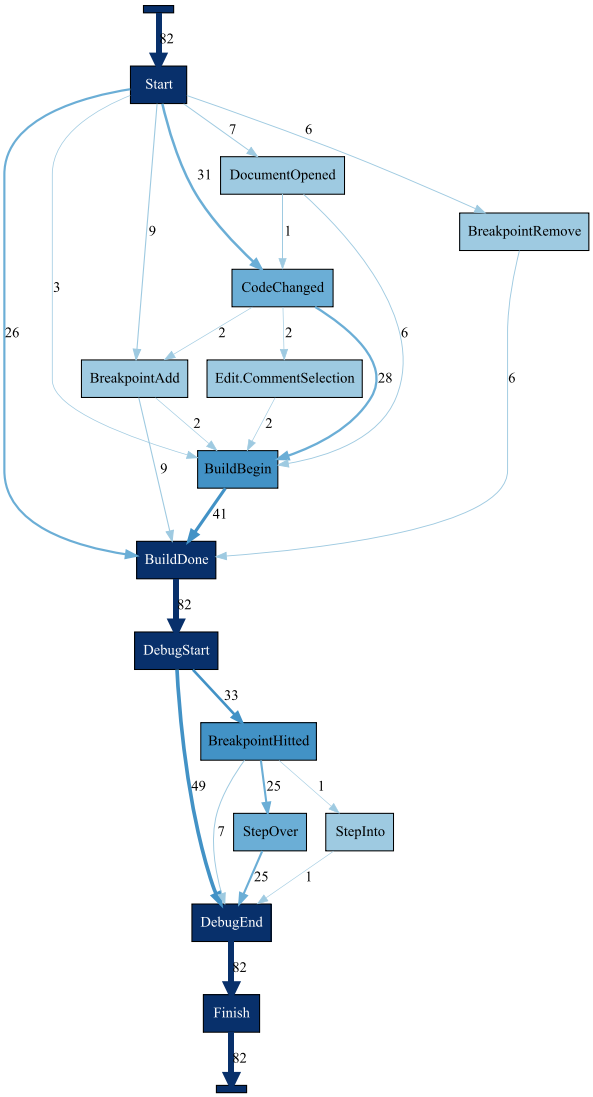


Рисунок 3.12 – Модель процесу налагодження із застосуванням одразу усіх шаблонів

На рис. 3.12 зображена модель одного з учасників експерименту, яка відображає застосування одразу усіх шаблонів під час вирішення завдання. Шаблон #1 – Start–BuildDone–DebugStart–DebugEnd–Finish. Шаблон #2 – Start–CodeChanged–BuildBegin–BuildDone–DebugStart–DebugEnd–Finish. Шаблон #3 – Start–BreakpointAdd–BuildDone–DebugStart–BreakpointHitted–DebugEnd–Finish. Шаблон #4 – Start–CodeChanged–BreakpointAdd–BuildBegin–BuildDone–DebugStart–BreakpointHitted– StepOver–DebugEnd–Finish.

Результати демонструють ефективність Process Mining для кращого розуміння того, як розробники налагоджують програми. Виходячи з результатів дослідження, адаптивні методи навчання можна застосовувати для студентів з різним ступенем успішності.

Використання методів Process Mining представляють новий підхід до вивчення процесів налагодження програм. В ході експерименту проведено оцінку розробників, щоб виділити тих, хто домігся найкращих результатів. Потім порівняти поведінку, отриману від цієї групи розробників, з поведінкою інших. Результати показують різні моделі поведінки розробників з різними навичками.

Установка точок зупину і проходження по коду виконувалося більшістю розробників за допомогою налагоджувача, інші функції налагодження використовуються набагато рідше і набагато меншою кількістю розробників. Лише 2 учасників використовували такі засоби як перегляд значень змінних. Учасники, які не використовували засоби налагодження натомість використовували метод спроб і помилок, а також використовували виведення значень на екран. Ці результати закликають посилити практичний досвід налагодження в навчальних програмах.

Хоча це не дивний результат, але експеримент наочно продемонстрував, що досвід є ключовим фактором навичок налагодження. Адже всі студенти 4 та 5 курсів виконали половину або більше завдань. А серед студентів першого курсу немає таких, які б виконали всі завдання.

Аналіз даних дозволив виявити студентів, які вирішували завдання не самостійно. Одного учасника було дискваліфіковано через те, що всі його завдання були вирішені шляхом вставки готового рішення. Окрім того ще 5 учасникам було не зараховано правильно виконані завдання через відсутність процесу досягнення результату, що викликає велику підозру в не самостійності. Також такі рішення співпадали з рішеннями інших учасників. Моделі процесів таких учасників складалися лише з декількох сесій налагодження, в кожній з яких було не більше 5 дій, одна з яких це команда Edit.Paste великого шматку коду.

Висновки до третього розділу

Перевірено та підтверджено в результаті експериментальних досліджень можливість застосування сформованих конструктивно-продукційних моделей та розроблених на їх основі інструментальних засобів для відстеження процесів розробки та налагодження програмного забезпечення.

Розроблено програми з типовими логічними помилками. Проведено олімпіаду з налагодження, під час якої за допомогою розроблених інструментів відстежувалися та фіксувалися дії учасників в середовищі розробки. Результати показали навички студентів по налагодженню програм з типовими логічними помилками. Вдалося виявити проблеми, які є більш складними для локалізації та виправлення.

Після застосування методів Process Mining до журналів подій сформовано моделі процесів розробки та налагодження програм. За допомогою інструментів та методів Process Mining проведено порівняння процесів різних учасників з метою виявлення особливостей поведінки.

Таким чином, розробивши інструменти та методи збору інформації про процеси розробки та налагодження й автоматизуючи формування, перевірку та аналіз моделей процесів, був розроблений повний технологічний цикл оцінки якості процесу.

Дане дослідження дозволяє як викладачеві, так і програмістам отримати додаткову інформацію про процес роботи та його якість, тим самим розширюючи можливості ефективного управління процесом і підвищення кваліфікації. Як відомо, чим більше інформації відомо про процес, тим ефективнішим може бути управління.

Результати дослідження показують, що методи Process Mining корисні для розуміння того, як програмісти виконують налагодження та перед якими труднощами постають.

Виходячи з результатів дослідження, адаптивні методи навчання можна застосовувати для студентів з різним ступенем успішності.

За матеріалами розділу опубліковано роботи [110, 111].

РОЗДІЛ 4
РОЗРОБКА ЗАСОБІВ ДЛЯ ВІДСТЕЖЕННЯ ПРОЦЕСІВ РОЗРОБКИ ТА НАЛАГОДЖЕННЯ ПРОГРАМ

4.1 Розробка інструментів для запису подій з середовища програмування

Одним з найпоширеніших середовищ розробки програмних засобів є Microsoft Visual Studio. Розробка інструментів відстеження процесів розробки та налагодження в Visual Studio, є дуже зручним, оскільки його широко застосовують як на виробництві, так і для навчання у вищих навчальних закладах.

Середовище розробки Visual Studio побудовано на принципах автоматизації та розширюваності, дозволяючи розробникам інтегрувати в себе практично будь-які нові елементи та взаємодіяти як зі стандартними, так і з новими компонентами. Для реалізації цих завдань користувачам Visual Studio надано кілька інструментів. Самим базовим і універсальним з яких, є об’єктна модель автоматизації Visual Studio.

Об’єктна модель Visual Studio складається з взаємопов’язаних груп об’єктів, які охоплюють усі аспекти роботи в середовищі розробки та надає можливості для їх розширення. Доступ до будь-якої з цих груп можливий через глобальний інтерфейс верхнього рівня *Development Tools Environment*.

Події об’єктів автоматизації визначені в кореневому елементі *Development Tools Environment – Events*. Даний інтерфейс містить посилання на наступні події:

- BuildEvents – перелік подій для побудови рішення;
- CommandEvents – події перед та після виконання команди;
- DebuggerEvents та DebuggerExpressionEvaluationEvents – події, що підтримуються налагоджувачем;
- DocumentEvents – перелік подій по роботі з документами;
- ProjectItemEvents – перелік подій по роботі з елементами проєкту;
- SolutionEvents – перелік подій по роботі з рішенням;
- TextEditorEvents – події про зміни, внесені до редактора коду;

– FindEvents – події для операцій пошуку у файлах;

– WindowEvents та WindowVisibilityEvents – події по роботі з вікнами.

Розширення – це надбудови, що дозволяють налаштовувати та покращувати взаємодію в Visual Studio шляхом додавання нових функцій або інтеграції наявних інструментів.

З точки зору задач розширення виділяють наступні:

– розширення графічного інтерфейсу створенням додаткової функціональності з відповідними елементами керування;

– розширення проєктів та рішень;

– розширення редактору коду шляхом доповнення функціональності вбудованого редактора коду, а також створення нових редакторів (наприклад, для нової мови програмування);

– розширення мовних служб повноцінним додаванням підтримки нової мови, в тому числі новий тип проєкту, редактор коду, специфічний налагоджувач;

– розширення налагоджувача або створення нового для нової мови програмування;

– розширення на основі створення наборів для Visual Studio (VSPackages).

З точки зору реалізації існує три методи розширення:

– створення макросів – запис дій користувача для їх автоматизованого повторення;

– створення додатків – інтегрованих до середовища COM-об’єктів з майже повним доступом до середовища та його API;

– створення інтегрованих наборів (VSPackage) – є рекомендованим засобом розробки розширень, дозволяє поєднувати в собі автоматизацію управління компонентами середовища розробки через об’єктну модель автоматизації.

Для розширення середовища розробки Visual Studio можливістю відстежувати увесь процес розробки та налагодження програм був розроблений відповідний VSPackage за допомогою Microsoft Visual Studio SDK.

Наша мета зафіксувати взаємодію розробників із середовищем розробки внаслідок не лише дій користувача, а й детальної контекстної інформації.

На рівні процесу ми фіксуємо події з базовою інформацією, такою як тип події (наприклад, зміна або збереження коду) та час. На контекстному рівні ми зберігаємо конкретні дані залежно від типу події. Наприклад, події збереження зберігають ім’я файлу, який було збережено, та спосіб, за допомогою якого було введено дію збереження (наприклад, за допомогою гарячих клавіш чи вибору пункту меню).

Для більшості подій контекст фіксує ціль, для якої було викликано подію, наприклад ім’я збереженого файлу, інформацію про побудовані проєкти або ідентифікатор команди, яку викликали. Для подій змін контекст також робить знімок коду файлу, який піддається редагуванню.

За допомогою підписки на події від середовища розробки можна отримувати всю необхідну інформацію про дії користувачів.

Реалізувавши інтерфейс *IWpfTextViewCreationListener* отримуємо можливість слухати події текстового редактора, в якому пишеться код. *IWpfTextViewCreationListener* має метод *TextViewCreated* який викликається, коли текстове представлення редактора створено. *TextViewCreated* приймає у якості параметра об’єкт інтерфейсу *IWpfTextView*. *IWpfTextView* – це створене текстове представлення.

В конструкторі підписуємося на подію зміни вмісту текстового редактора

```
this.view.TextBuffer.Changed +=
this.OnTextContentChanged;
```

в результаті будь-які зміни тексту програми будуть викликати обробник *OnTextContentChanged*

```
private void OnTextContentChanged(object sender,
        TextContentChangedEventArgs e)
```

TextContentChangedEventArgs – надає інформацію про транзакції редагування в *ITextBuffer*. Властивість *Changes* являє собою колекцію об’єктів *ITextChange*.

ITextChange – описує одну безперервну операцію зміни тексту в текстовому редакторі.

Об’єкт інтерфейсу *ITextChange* та об’єкт *TextContentChangedEventArgs* дає всю необхідну інформацію для ведення історії розробки.

Починаючи з версії Visual Studio 2010, з’явилась можливість істотно спростити розгортання VSPackage модулей за допомогою VSIX пакетів. Фактично VSIX файл – це повноцінний інсталятор, який дозволяє встановити пакет методом «одного кліку». Публікація VSIX пакету на офіційному сайті розширень Visual Studio Gallery дозволить користувачу встановлювати такий модуль через середовище (рис. 4.1).

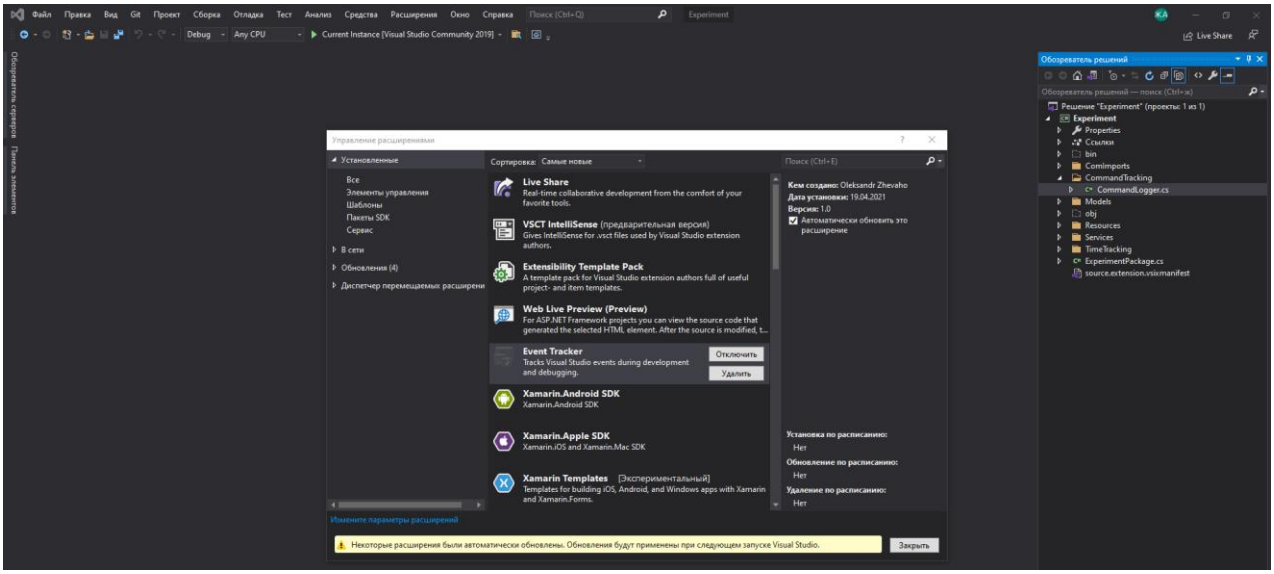


Рисунок 4.1 – Управління розширеннями в середовищі розробки

На основі конструктивних моделей розроблено розширення до Microsoft Visual Studio, в якому всі дії по написанню та налагодженню програмного забезпечення реєструються в журналах подій.

Для збереження та аналізу була створена модель даних, яка відповідає раніше представленим терміналам з їх атрибутами. На рисунку 4.2 зображено запропоновану модель у вигляді діаграми класів.

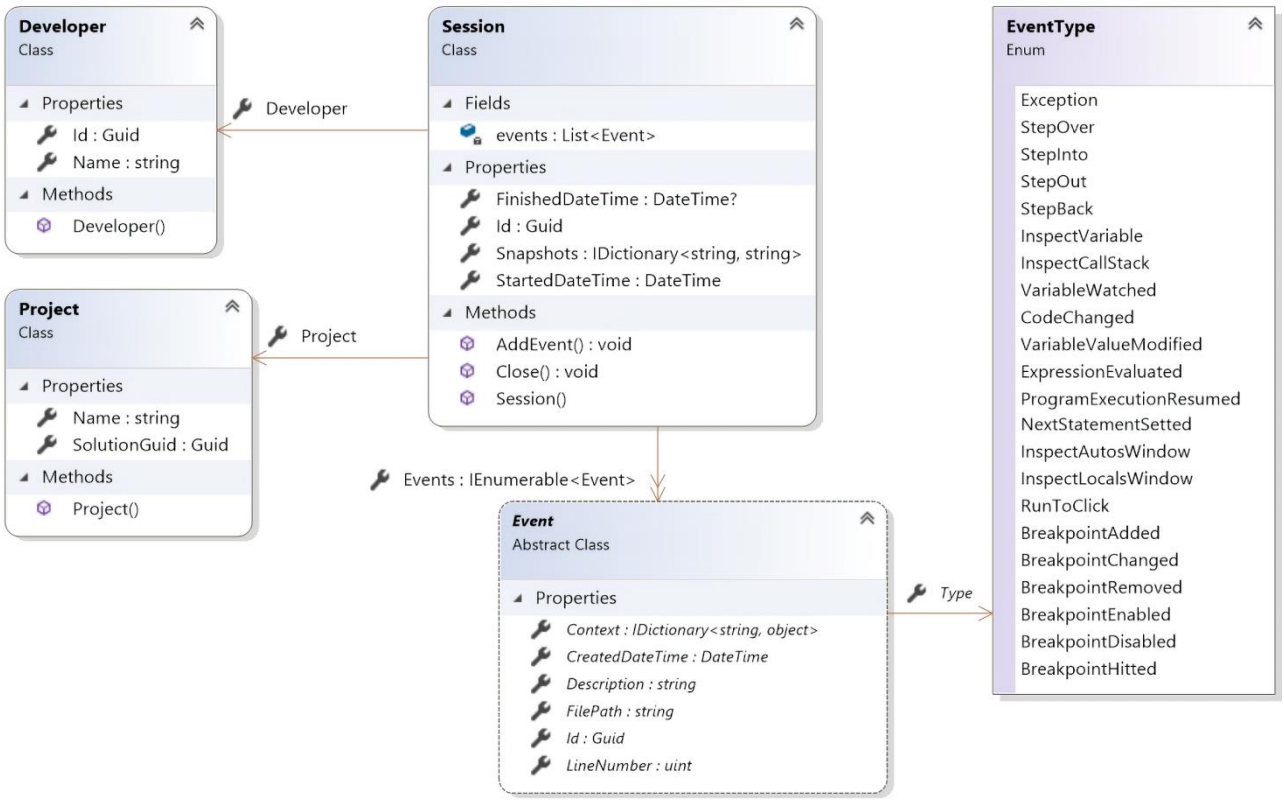


Рисунок 4.2 – Діаграма класів моделі даних для збереження та аналізу подій добутих з середовища розробки

Діаграма класів базується на моделі подій в середовищі розробки. Події виникають, коли програміст виконує будь-які дії у своєму середовищі розробки. Події містять тип, час, контекст і шлях до активного файлу з номером рядка, на якому відбулася подія, якщо це необхідно. Зберігається не тільки проста інформація про виконувані команди, але й контекстні дані, об’єкт динамічної структури з інформацією про середовище події. Структура об’єкта залежить від типу події.

Developer – це користувач, який працює в середовищі розробки.

Project – це набір компонентів з якими працює розробник.

Session – це сеанс, який містить інформацію про розробника, проєкт та події.

Кожна подія пов’язана з певним типом. Кожен тип події представлений певним класом, який є реалізацією абстрактного базового класу *Event*. Список *EventType* містить усі можливі типи подій, які відстежуються.

На рисунку 4.3 представлена архітектура верхнього рівня розробленого програмного інструменту.

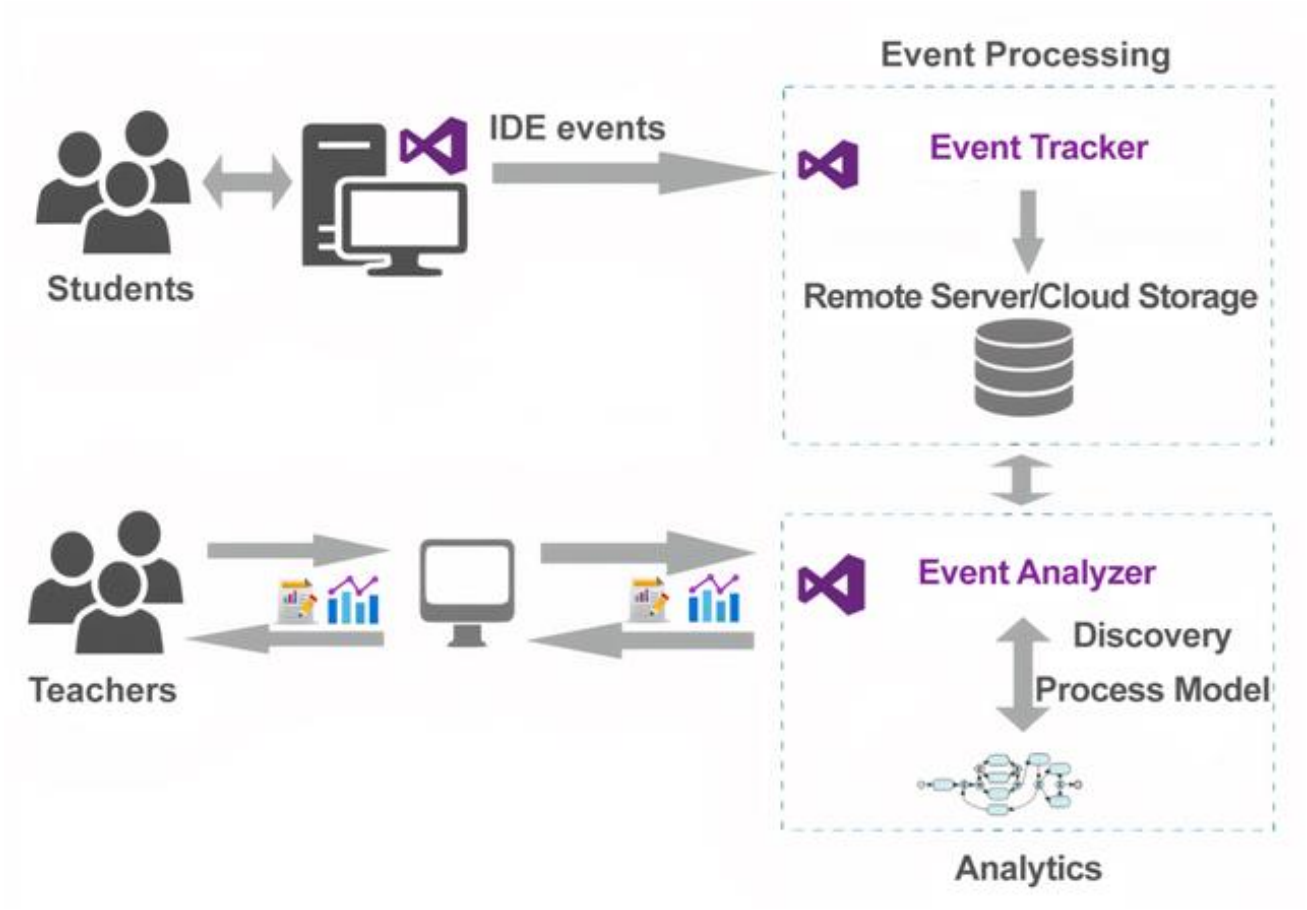


Рисунок 4.3 – Архітектура інструменту для моніторингу та аналізу процесів розробки та налагодження

На рисунку 4.3 показані два основні компоненти: Event Tracker та Event Analyzer.

Event Tracker – це розширення до Microsoft Visual Studio, написане мовою програмування C#. Компонент відповідає за збір даних із поточних сеансів роботи проєкту та надсилання їх на віддалений сервер. Розширення не тільки реєструє, які події відбуваються в середовищі розробки, але також зберігає відповідну контекстну інформацію про них.

Event Analyzer відповідає за аналіз журналів подій отриманих з середовища розробки за допомогою методів Process Mining.

4.2 Візуалізація процесу розробки

За допомогою використання бібліотеки Three.js реалізовано можливість візуалізації процесу розробки програмного забезпечення як набору 3D кубів.

Вхідними даними для візуалізації є словник. Ключ словника – це порядковий номер рядка, доданого до програми, що вивчається, значенням є сам рядок. Результат – послідовність тривимірних паралелепіпедів висотою n-i, де n – кількість рядків у словнику, i – ключ словника. Отже, рядок, написаний першим при розробці програми, матиме максимальну висоту.

4.2.1 Приклад застосування методу покрокової деталізації та візуалізації процесу розробки програми

Розглянемо приклад візуалізації процесу розробки програми.

Лістинг 4.1 – Приклад програми для візуалізації процесу її розробки

```
/// Searches the entire sorted array of numbers for an element.
/// <param name="numbers">Array of numbers</param>
/// <param name="item">The object to locate.</param>
/// <returns>The index of the element if item is found, otherwise a negative number</returns>
public static int BinarySearch(int[] numbers, int item)
{
    int low = 0;
    int high = numbers.Length - 1;
    while (low <= high)
    {
        // Find the middle of the array
        int mid = low + (high - low) / 2;
        if (item < numbers[mid])
        {
            // If the search item is less than the value in the middle, then the high limit will be the element to the middle.
            high = mid - 1;
        }
        else if (item > numbers[mid])
        {
            // If the search key is greater than the value in the middle, then the lower limit will be the element after the middle.
            low = mid + 1;
        }
        else
```

```
        {
            // Item index found.
            return mid;
        }
    }
    // Item index not found.
    return -1;
}
```

Історія розробки програми буде виглядати наступним чином:

Лістинг 4.2 – Історія розробки програми

```
#1 - public static int BinarySearch(int[] numbers, int item)
#2 - int low = 0;
#3 - int high = numbers.Length - 1;
#4 - return -1;
#5 - while (low <= high)
#6 - int mid = (high - low) / 2;
#7 - return mid;
#8 - if (item < numbers[mid])
#9 - high = mid - 1;
#10 - else if (item > numbers[mid])
#11 - low = mid + 1;
#12 - else
#13 - /// Searches the entire sorted array of numbers for
an element.
/// <param name="numbers">Array of numbers</param>
/// <param name="item">The object to locate.</param>
/// <returns>The index of the element if item is found,
otherwise a negative number</returns>
#14 - // Item index not found.
#15 - // If the search item is less than the value in the
middle, then the high limit will be the element to the
middle.
#16 - // If the search key is greater than the value in the
middle, then the lower limit will be the element after the
middle.
#17 - low +
#18 - // Find the middle of the array
#19 - // Item index found.
```

Ітераційний процес розробки представлений на рисунку 4.4.

Будь-яке складне завдання деталізується на наступній ітерації на наступному рівні P_i.

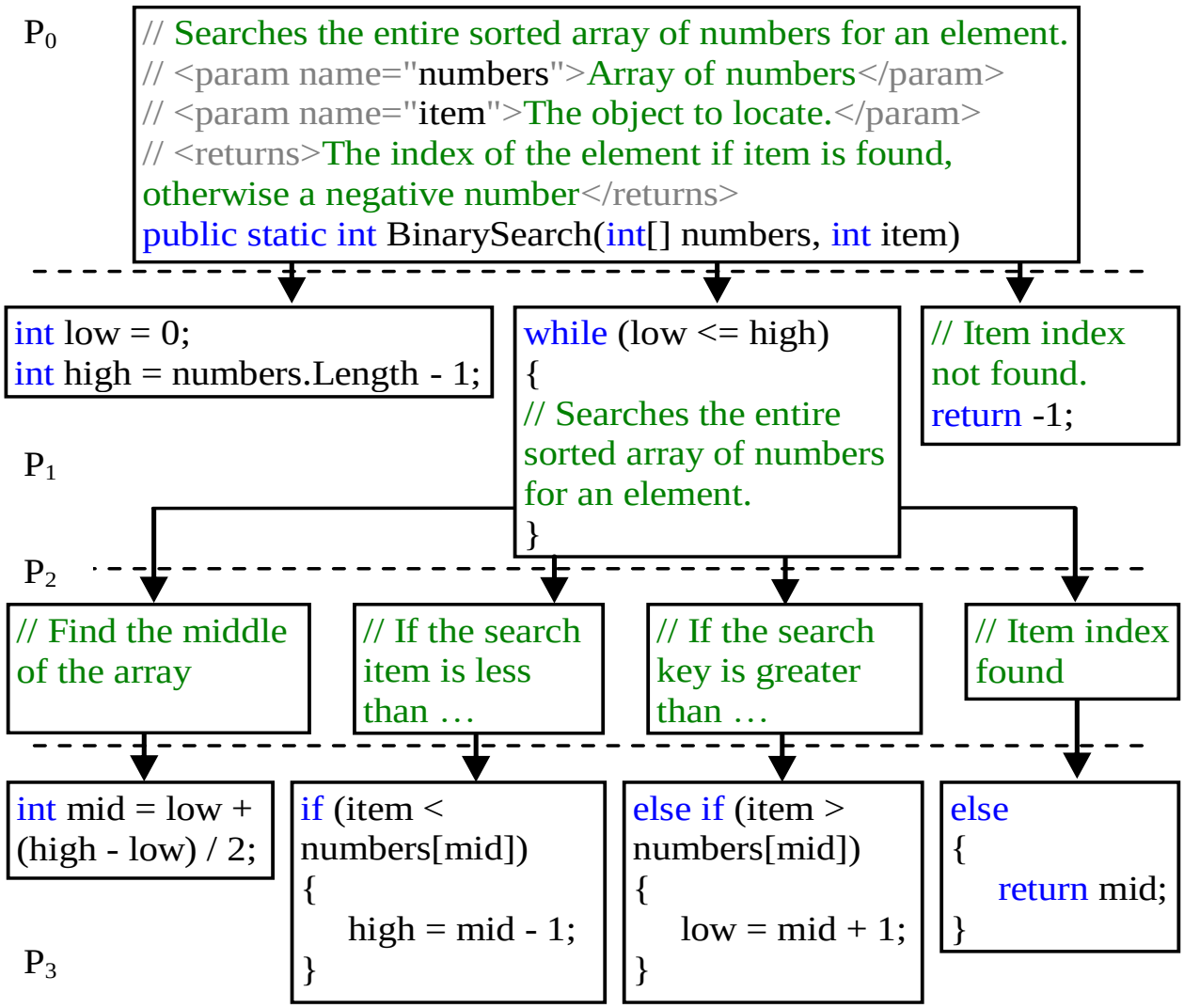


Рисунок 4.4 – Графічне представлення процесу розробки програми відповідно до методу покрокової деталізації

Приклад типового відхилення від методу покрокової деталізації процесу розробки програми представлений на рисунку 4.5.

Це відхилення виражається в хаотичному написанні тексту програми, внесенні змін у вже написаний фрагмент тексту, а також додаванні коментарів в кінці процесу розробки.

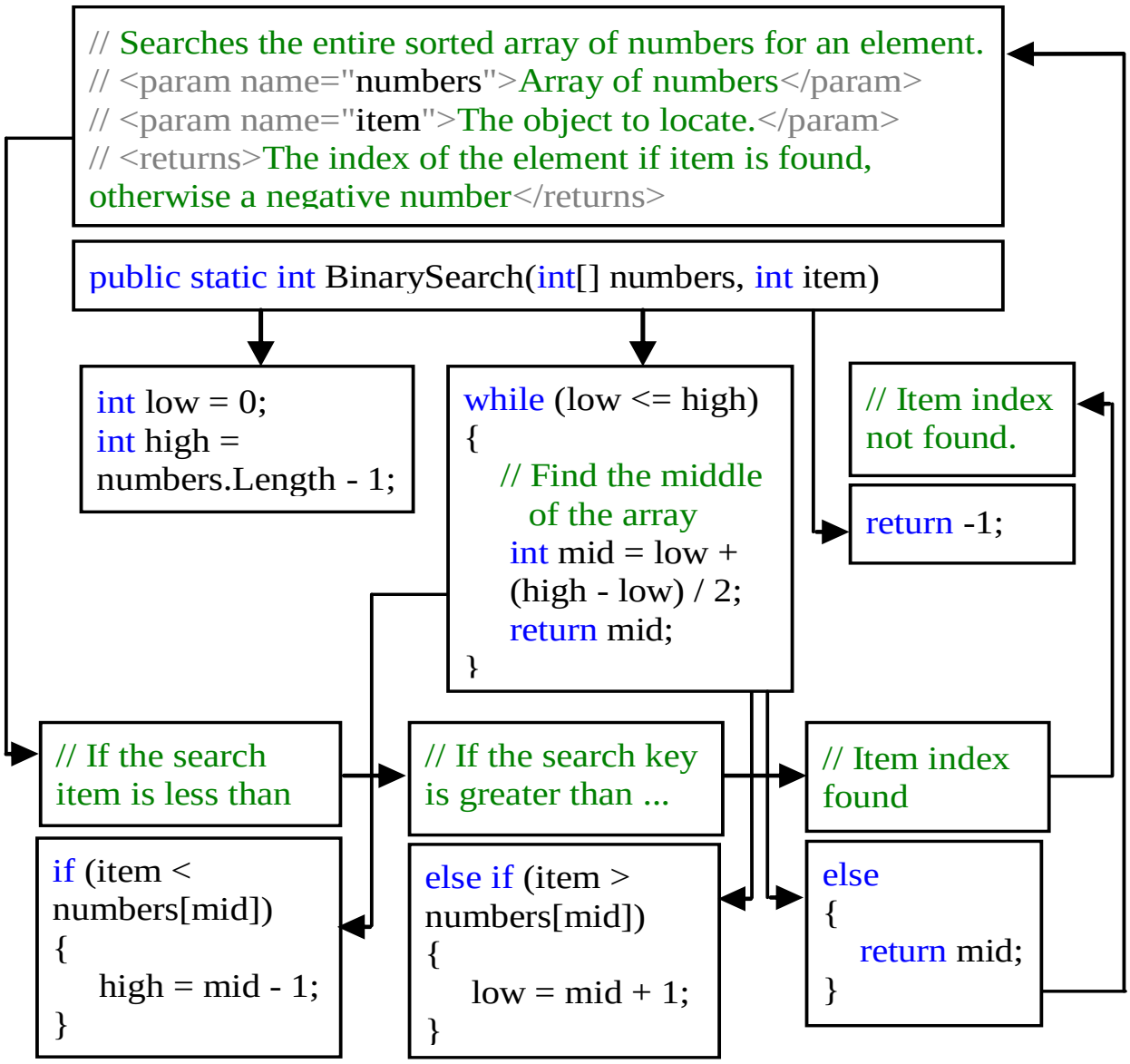


Рисунок 4.5 – Графічне представлення процесу розробки програми з типовими відхиленнями від методу покрокової деталізації

На рисунку 4.6 показаний фрагмент журналу подій у форматі eXtensible Event Stream, побудований для історії розробки програми (лістинг 4.1). Файл містить послідовність подій відповідно до типів змін коду (додавання класу, методу, змінної тощо).

```
<trace>
    <string key="developer" value="John Doe"/>
    <string key="project" value="Demo"/>
    <date key="finishedDateTime" value="2020-06-20T01:11:00.000+02:00"/>
    <date key="startedDateTime" value="2020-06-20T01:00:00.000+02:00"/>
    <string key="concept:name" value="Session1"/>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-06-20T01:00:01.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Start"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-06-20T01:01:00.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="AddClass"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-06-20T01:02:01.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="AddMethod"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-06-20T01:02:10.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="AddVariableWithAssignment"/>
    </event>
    <event>
        <string key="context" value="serialized context"/>
        <date key="time:timestamp" value="2020-06-20T01:03:00.000+02:00"/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="AddVariableWithAssignment"/>
    </event>
```

Рисунок 4.6 – Фрагмент журналу подій

Inductive Visual Miner з ProM, інструмента для Process Mining, використаний для виявлення моделі процесу розробки. Результатом є модель процесу, показана на рисунку 4.7.

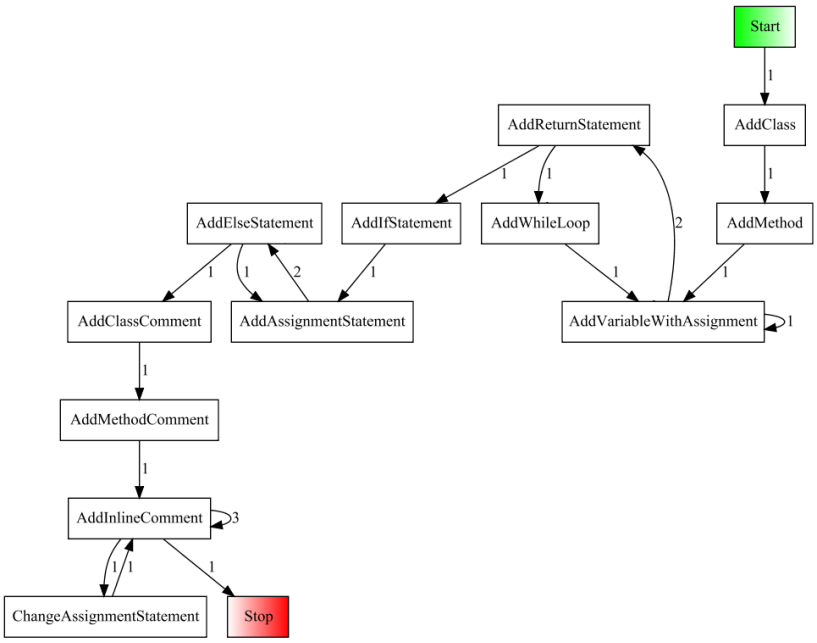


Рисунок 4.7 – Модель процесу, виявлена за допомогою Inductive Visual Miner

Візуальне 3D-представлення процесу розробки дозволяє вчителю отримувати всю необхідну інформацію в зручному вигляді. Тривимірне

представлення процесу кодування наведено на рисунку 4.8. У викладача є можливість масштабувати та обертати 3D-зображення.

Рисунок 4.8 – Візуальне 3D представлення процесу розробки програмного забезпечення

Візуалізація процесу розробки представленої програми наочно показує, що метод покрокової деталізації порушено, оскільки коментарі були написані після тексту програми, а оператор (агрегат) int mid = low + (high - low) / 2; не писався одразу.

Висновки до четвертого розділу

В ході роботи на основі конструктивно-продукційних моделей вдалося розробити засоби відстеження, моделювання та аналізу процесів розробки та налагодження програм.

Отримані інструменти дозволяють:

- фіксувати взаємодію розробників із середовищем розробки;
- формувати журнали подій про процеси розробки та налагодження;
- візуалізувати процес розробки у вигляді 3D зображення з можливістю масштабування та обертання.

За матеріалами розділу опубліковано роботи [106, 110].

ВИСНОВКИ

У дисертаційній роботі вирішено актуальну науково-практичну задачу аналізу процесів розробки та налагодження програм, формування нових підходів і розробку інструментів для підвищення якості навчання студентів програмуванню.

В результаті дисертаційного дослідження отримано такі основні наукові та практичні результати:

1. Виконано огляд та аналіз наявних підходів до отримання інформації щодо процесів розробки та налагодження програмного забезпечення. За його результатами зроблено висновок, що наявні підходи до отримання інформації про процеси розробки та налагодження тексту програм не засновані на моделях процесів і не дозволяють вивчити процес кожного конкретного розробника програмного забезпечення. Також зроблено висновок про доцільність розробки методів та інструментів для відстеження, моделювання та аналізу процесів розробки та налагодження програмного забезпечення.

2. Вперше виконано формалізацію процесів розробки та налагодження програмного забезпечення засобами конструктивно-продукційного моделювання, що на відміну від існуючих дозволяє розглядати ці процеси як послідовність елементарних дій за продукційними правилами, формалізувати процеси формування журналів подій та візуалізувати ці процеси. Розроблено три конструктори. Перший, створює модель процесу розробки програмного забезпечення під час використання середовища розробки для автоматичного аналізу. Другий, класифікує історію розробки програми за дрібними змінами та створює файл журналів подій з метою вивчення процесу кодування студентів. Третій, формує журнал налагоджувальних дій під час роботи з відповідними інструментами в середовищі розробки.

3. На відміну від інших досліджень дані щодо процесів розробки та налагодження програм фіксуються напряму з середовища розробки. Для реалізації можливості відстеження дій кожного студента як на лабораторних заняттях, так і під час самостійної роботи на основі конструктивних моделей, створено

доповнення до середовища розробки Visual Studio в якому всі дії по розробці та налагодженню фіксуються в журналах подій. Розроблені інструменти дозволяють: фіксувати взаємодію розробників із середовищем розробки; формувати журнали подій про процеси розробки та налагодження; візуалізувати процес розробки у вигляді 3D зображення з можливістю масштабування та обертання. При розробці програмної реалізації отриманих моделей застосована технологія об'єктно-орієнтованого проєктування. Розроблений інструментарій дозволяє використовувати запропоновані моделі та методи в технології навчання студентів основам програмування.

4. Розроблені моделі процесів розробки та налагодження програм засобами Process Mining, що на відміну від існуючих дають можливість автоматизувати аналіз цих процесів. За допомогою інструментів та методів Process Mining проведено порівняння процесів різних учасників з метою виявлення особливостей поведінки. Результати дослідження показують, що методи Process Mining корисні для розуміння того, як програмісти виконують налагодження та перед якими труднощами постають. Інформація, витягнута з процесу розробки програми, може бути використана для автоматичного надання студенту рекомендацій щодо вдосконалення коду та відстеження тенденції його зміни. Аналіз історії розробки програми та налагодження показує, скільки часу зайняла розробкаа програми та його налагодження, які частини викликали найбільші труднощі. Запровадження конструктивно-продукційного підходу та використання методів Process Mining представляють новий підхід до вивчення процесів розробки та налагодження програм.

5. Перевірено та підтверджено в результаті експериментальних досліджень можливість застосування сформованих конструктивно-продукційних моделей та розроблених на їх основі інструментальних засобів для відстеження процесів розробки та налагодження програмного забезпечення. Проведено експеримент з налагодження, під час якої за допомогою розроблених інструментів відстежувалися та фіксувалися дії учасників в середовищі розробки. Для виявлення навичок налагодження застосовано метод підсіву помилок. Розроблено

програми з типовими логічними помилками. Результати показали навички студентів по налагодженню програм з типовими логічними помилками. Вдалося виявити проблеми, які є більш складними для локалізації та виправлення. Виходячи з результатів дослідження, адаптивні методи навчання можна застосовувати для студентів з різним ступенем успішності. Результати проведених досліджень та впровадження запропонованих процесів та інструментів дозволять підвищити якість навчання студентів внаслідок моніторингу їх роботи.

6. Удосконалено підхід до оцінювання робіт з програмування, який враховує не тільки результат, а й процес його досягнення. Для цього представлено інноваційний метод використання огляду коду і процесу розробки та налагодження програмного забезпечення при навчанні навичкам програмування. Застосування запропонованого підходу надасть викладачеві можливість провести сучасний огляд коду, який, на наш погляд, повинен включати не тільки перегляд коду, але й аналіз процесу розробки та налагодження. Дане дослідження дозволяє як викладачеві, так і програмісту, отримати додаткову інформацію про процеси розробки і налагодження та їх якості, тим самим, розширюючи можливості по ефективному управлінню процесами, вдосконалення навичок. Як відомо, чим більше інформації відомо про процес, тим ефективніше може бути управління.

Результати дисертаційної роботи опубліковані в 12 наукових працях. У фахових виданнях, що індексуються МНБД Scopus та Web of Science – 2. У матеріалах міжнародних конференцій, що індексуються МНБД Scopus – 2. У фахових та рекомендованих МОН України для публікації результатів дисертацій – 2. У тезах доповідей міжнародних та всеукраїнських конференцій – 6.

Дисертація є частиною науково-дослідної роботи «Конструктивно-продукційне моделювання в задачах розробки програмного забезпечення» (2021 р. № держреєстрації 0121U109167), яка була виконана на кафедрі «Комп’ютерних інформаційних технологій» Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. DeMarco T., Lister T. Peopleware: Productive Projects and Teams // USA: Addison-Wesley. – 2013. – P. 272.

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

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. – 2019. – 24 (4). – pp. 423–434. 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 // Frontiers in Education Conference. – 2017. – pp. 1–8. https://doi.org/10.1109/FIE.2017.8190658.

5. Breuker D. M., Derriks J., Brunekreef J. Measuring static quality of student code // Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science. – 2011. – pp. 13–17. https://doi.org/10.1145/1999747.1999754.

6. Keuning H., Heeren B., Jeuring J. Code quality issues in student programs // Annual Conference on Innovation and Technology in Computer Science Education. – 2017. – pp. 110–115. https://doi.org/10.1145/3059009.3059061.

7. Fonseca N. G., Macedo L., Mendes A. J. CodeInsights – Monitoring programming students' progress // ACM International Conference Proceeding Series. – 2016. – pp. 375–382. https://doi.org/10.1145/2983468.2983492.

8. Cherepovskaya E. N., Gorshkova E. V., Lyamin A. V. Assessment of students' programming skills by using multi-style code editor // Smart Innovation, Systems and Technologies. Springer Science and Business Media, Deutschland GmbH. – 2017. – pp. 225–234. https://doi.org/10.1007/978-3-319-59451-4_22.

9. Dijkstra E. W. A Discipline of Programming // Prentice-Hall. – 1976.

10. Gehani N. Program development by stepwise refinement and related topics // The Bell System Technical Journal. – 1981. – pp. 347–378.

11. Wirth N. Program development by stepwise refinement // Communications of ACM. – 1971. – pp. 221–227.

12. McConnell S. Code complete // Redmond: Microsoft Press. – 2004.

13. Bennedsen J., Caspersen M. E. Revealing the programming process // Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education. – 2005. – pp. 186–190.

14. Ambrose S. A., Bridges M. W., DiPietro M., Lovett M. C., Norman M. K. How Learning Works:Seven Research-Based Principles for Smart Teaching // Jossey-Bass. – 2010.

15. National Research Council. How people learn: Brain, mind, experience, and school: Expanded edition // National Academies Press. – 2000.

16. Koile K., Singer D. Improving learning in CS1 via tablet-PC-based in-class assessment // Proceedings of the second international workshop on Computing education research. – 2006. – pp. 119–126.

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

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

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

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

21. Negara S., Vakilian M., Chen N., Johnson R. E., Dig D. Is it dangerous to use version control histories to study source code evolution? // European Conference on

Object-Oriented Programming. – 2012. – pp. 79–103. https://doi.org/10.1007/978-3-642-31057-7_5.

22. Snipes W., Murphy-Hill E., Fritz T., Vakilian M., Damevski K., Nair A. R., Shepherd D. A Practical Guide to Analyzing IDE Usage Data // The Art and Science of Analyzing Software Data. – 2015. – pp. 85–138. https://doi.org/10.1016/B978-0-12-411519-4.00005-7.

23. 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). – 2017. – pp. 359–371. https://doi.org/10.1109/TSE.2016.2592905.

24. Snipes W., Augustine V., Nair A. R., Murphy-Hill E. Towards recognizing and rewarding efficient developer work patterns // 35th International Conference on Software Engineering. – 2013. – pp. 1277–1280. https://doi.org/10.1109/ICSE.2013.6606697.

25. Snipes W., Nair A. R., Murphy-Hill E. Experiences gamifying developer adoption of practices and tools // 36th International Conference on Software Engineering. – 2014. – pp. 105–114. https://doi.org/10.1145/2591062.2591171.

26. Amann S., Proksch S., Nadi S., Mezini M. A Study of Visual Studio Usage in Practice // 23rd International Conference on Software Analysis, Evolution, and Reengineering. – 2016. – pp. 124–134. https://doi.org/10.1109/saner.2016.39.

27. Yamashita A., Petrillo F., Khomh F., Guéhéneuc Y.-G. Developer interaction traces backed by IDE screen recordings from think aloud sessions // Proceedings of the 15th International Conference on Mining Software Repositories. – 2018. https://doi.org/10.1145/3196398.3196457.

28. Amann S., Proksch S., Nadi S. FeedBaG An interaction tracker for Visual Studio // IEEE 24th International Conference on Program Comprehension. – 2016. – pp. 1–3. https://doi.org/10.1109/ICPC.2016.7503741.

29. Brown N. C. C., Kölling M., McCall D., Utting I. Blackbox: A large scale repository of novice programmers' activity // Proceedings of the 45th ACM Technical

Symposium on Computer Science Education, SIGCSE ’14. – 2014. – pp. 223–228. https://doi.org/10.1145/2538862.2538924.

30. Brown N. C. C., Altadmri A., Sentence S., Kölling M. Blackbox, five years on: An evaluation of a large-scale programming data collection project // Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER ’18. – 2018. pp. 196–204. https://doi.org/10.1145/3230977.3230991.

31. Brown N. C. C., Altadmri A. 37 million compilations: Investigating novice programming mistakes in large-scale student data // Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE ’15. – 2015. – pp. 522–527. https://doi.org/10.1145/2676723.2677258.

32. Brown N. C. C., Altadmri A. Investigating novice programming mistakes: Educator beliefs vs. student data // Proceedings of the Tenth Annual Conference on International Computing Education Research, ICER ’14. – 2014. – pp. 43–50. https://doi.org/10.1145/2632320.2632343.

33. McCall D., Kölling M. Meaningful categorisation of novice programmer errors // 2014 IEEE Frontiers in Education Conference (FIE) Proceedings. – 2014. – pp. 1–8. https://doi.org/10.1109/FIE.2014.7044420.

34. Reestman K., Dorn B. Native language’s effect on java compiler errors // Proceedings of the 2019 ACM Conference on International Computing Education Research, ICER ’19. – 2019. – pp. 249–257. https://doi.org/10.1145/3291279.3339423.

35. Keuning H., Heeren B., Jeuring J. Code quality issues in student programs // Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE ’17. – 2017. – pp. 110–115. https://doi.org/10.1145/3059009.3059061.

36. Jadud M. A first look at novice compilation behaviour using bluej // Computer Science Education. – 15 (1). – 2005. – pp. 25–40.

37. Vihavainen A., Vikberg T., Luukkainen M., Pärtel M. Scaffolding students’ learning using test my code // Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE ’13. – 2013. – pp. 117–122. https://doi.org/10.1145/2462476.2462501.

38. Hosseini R., Vihavainen A., Brusilovsky P. Exploring problem solving paths in a java programming course // In Psychology of Programming Interest Group Conference, PPIG 2014. – 2014. – pp. 65–76.

39. Johnson P. M., Kou H., Agustin J. M., Zhang Q., Kagawa A., Yamashita T. Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from hackystat-uh // Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. – 2004. – pp. 136–144. https://doi.org/10.1109/ISESE.2004.1334901.

40. Spacco J., Hovemeyer D., Pugh W., Emad F., Hollingsworth J. K., Padua-Perez N. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses // Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITICSE '06. – 2006. – pp. 13–17. http://doi.acm.org/10.1145/1140124.1140131.

41. Spacco J., Pugh W. Helping students appreciate test-driven development // Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06. – 2006. – pp. 907–913. https://doi.org/10.1145/1176617.1176743.

42. Murphy G. C., Kersten M., Findlater L. How are java software developers using the elipse ide? // IEEE Software. –23 (4). – 2006. – pp. 76–83.

43. Minelli R., Lanza M. Visualizing the workflow of developers // 2013 First IEEE Working Conference on Software Visualization (VISSOFT). – 2013. – pp. 1–4.

44. Robinson P. E., Carroll J. An online system for monitoring and assessing the programming process // Bulletin of the Technical Committee on Learning Technology. – 2016. pp. 2–5.

45. Cassidy C., Goldman M., Miller R. C. Glanceable code history: Visualizing student code for better instructor feedback // Proceedings of the 5th Annual ACM Conference on Learning at Scale. – 2018. https://doi.org/10.1145/3231644.3231680.

46. IEEE Standard Glossary of Software Engineering Terminology. – 1990. https://doi.org/10.1109/ieeestd.1990.101064.

47. LaToza T. D., Myers B. A. Developers ask reachability questions // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE ’10. – 2010. https://doi.org/10.1145/1806799.1806829.

48. Bers M. U., Flannery L., Kazakoff E. R., Sullivan A. Computational thinking and tinkering: Exploration of an early childhood robotics curriculum // Computers & Education. – 2014. – pp. 145–157. https://doi.org/10.1016/j.compedu.2013.10.020.

49. Lee G. C., Wu J. C. Debug It: A debugging practicing system // Computers & Education. – 32 (2). – 1999. – pp. 165–179. https://doi.org/10.1016/s0360-1315(98)00063-3.

50. Maalej W., Tiarks R., Roehm T., Koschke R On the Comprehension of Program Comprehension // ACM Transactions on Software Engineering and Methodology. – 23 (4). – 2014. – pp. 1–37. https://doi.org/10.1145/2622669.

51. Denny P., Luxton-Reilly A., Tempero E., Hendrickx J. Understanding the syntax barrier for novices // Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education – ITiCSE ’11. – 2011. https://doi.org/10.1145/1999747.1999807.

52. Denny P., Luxton-Reilly A., Carpenter D. Enhancing syntax error messages appears ineffectual // Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education – ITiCSE ’14. – 2014. https://doi.org/10.1145/2591708.2591748.

53. Yamamoto R., Noguchi Y., Kogure S., Yamashita K., Konishi T., Itoh Y. Design of a learning support system and lecture to teach systematic debugging to novice programmers // ICCE 2016 – 24th International Conference on Computers in Education: Think Global Act Local – Main Conference Proceedings. – 2016. – pp. 276–281.

54. Alqadi B. S., Maletic J. I. An Empirical Study of Debugging Patterns Among Novices Programmers // Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education. – 2017. https://doi.org/10.1145/3017680.3017761.

55. Bryce R. C., Cooley A., Hansen A., Hayrapetyan N. A one year empirical study of student programming bugs // 2010 IEEE Frontiers in Education Conference (FIE). – 2010. https://doi.org/10.1109/fie.2010.5673143.

56. Zeller A. Why Programs Fail, Second Edition: A Guide to Systematic Debugging // Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition. – 2009.

57. Zeller A. Isolating cause-effect chains from computer programs // Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering. – 2002. – pp. 1–10.

58. Li J., Nie C., Lei Y. Improved delta debugging based on combinatorial testing // 2012 12th International Conference on Quality Software. – 2012. – pp. 102–105.

59. Sukkerd R., Beschastnikh I., Wuttke J., Zhang S., Brun Y. Understanding regression failures through test-passing and test-failing code changes // Proceedings of the 2013 International Conference on Software Engineering, ICSE'13. – 2013. – pp. 1177–1180.

60. Yu K., Lin M., Chen J., Zhang X. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers perspectives // Journal of Systems and Software. – 85 (10). – 2012. – pp. 2305 – 2317.

61. Babenko A., Mariani L., Pastore F. Ava: Automated interpretation of dynamically detected anomalies // Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09. – 2009. – pp. 237–248.

62. Parsa S., Naree S. A. A new semantic kernel function for online anomaly detection of software // ETRI Journal. –34 (2). – 2012. – pp. 288–291.

63. Eichinger F., Krogmann K., Klug R., Bohm K. Software-defect localisation by mining dataflow-enabled call graphs // Machine Learning and Knowledge Discovery in Databases. – 2010. – pp. 425–441.

64. Eichinger F., Pankratius V., Große P., Bohm K. Localizing defects in multithreaded programs by mining dynamic call graphs // Testing–Practice and Research Techniques. – 2010. – pp. 56–71.

65. Chandra S., Torlak E., Barman S., Bodik R. Angelic debugging // Software Engineering (ICSE), 2011 33rd International Conference. – 2011. – pp. 121–130.

66. Malik M. Z., Siddiqi J. H., Khurshid S. Constraint based program debugging using data structure repair // IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST). – 2011. – pp. 190–199.

67. Abramson D., Chu C., Kurniawan D., Searle A. Relative debugging in an integrated development environment // Software: Practice and Experience. – 39 (14). – 2009. – pp. 1157–1183.

68. Zhang C., Yang J., Yan D., Yang S., Chen Y. Automated breakpoint generation for debugging // Journal of Software. – 8 (3). –2013.

69. Wei Y., Pei Y., Furia C. A., Silva L. S., Buchholz S., Meyer B., Zeller A. Automated fixing of programs with contracts // Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10. – 2010. – pp. 61–72.

70. Lei Y., Mao X., Dai Z., Wang C. Effective statistical fault localization using program slices // Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual. – 2012. – pp. 1–10.

71. Parsa S., Zareie F., Vahidi M. Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure // Experimental Algorithms. – 2011. – pp. 352–363.

72. Perscheid M., Hirschfeld R. Follow the path: Debugging tools for testdriven fault navigation // Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). – 2014. – pp. 446–449.

73. Roßler J., Fraser G., Zeller A., Orso A. Isolating failure causes through test case generation // Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA. – 2012. – pp. 309–319.

74. Yu K., Lin M., Chen J., Zhang X. Practical isolation of failureinducing changes for debugging regression faults // Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference. – 2012. – pp. 20–29.

75. Orso A. Automated debugging: Are we there yet?. – 2016.

76. Ghosh D., Singh J. A Systematic Review on Program Debugging Techniques // Smart Computing Paradigms: New Progresses and Challenges. – 2019. – pp. 193–199. https://doi.org/10.1007/978-981-13-9680-9_16.

77. Bottcher A., Thurner V., Schlierkamp K., Zehetmeier D. Debugging students' debugging process // 2016 IEEE Frontiers in Education Conference (FIE). – 2016. https://doi.org/10.1109/fie.2016.7757447.

78. R. Stallman, R. Pesch, S. Shebs Debugging with GDB // Free Software Foundation. – 2011.

79. Zeller A., Lutkehaus D. Ddd a free graphical front-end for unix debuggers // ACM Sigplan Notices. –31 (1). – 1996. – pp. 22–27.

80. Perscheid M., Siegmund B., Taeumel M., Hirschfeld R. Studying the advancement in debugging practice of professional software developers // Software Quality Journal. – 25 (1). – 2016. – pp. 83–110. https://doi.org/10.1007/s11219-015-9294-2.

81. Piorkowski D., Fleming S. D., Scaffidi C., Burnett M., Kwan I., Henley A. Z., Macbeth J., Hill C., Horvath A. To fix or to learn? how production bias affects developers' information foraging during debugging // Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), ICSME '15. – 2015. – pp. 11–20.

82. Piorkowski D. J., Fleming S. D., Kwan I., Burnett M. M., Scaffidi C., Bellamy R., Jordahl J. The whats and hows of programmers' foraging diets // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. – 2013. – pp. 3063–3072.

83. Piorkowski D., Fleming S., Scaffidi C., Bogart C., Burnett M., John B., Bellamy R., Swart C. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '12. – 2012. – pp. 1471–1480.

84. Gugerty L., Olson G. Debugging by skilled and novice programmers // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. – 1986. – pp. 171–174.

85. Ahmadzadeh M., Elliman D., Higgins C. An analysis of patterns of debugging among novice computer science students // Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05). – 2005. – pp. 84–88.

86. Gouws L. A., Bradshaw K., Wentworth P. Computational thinking in educational activities // Proceedings of the 18th ACM Conference on Innovation and Technology in

Computer Science Education – ITiCSE ’13. – 2013. https://doi.org/10.1145/2462476.2466518.

87. Altadmri A., Brown N. C. C. 37 Million Compilations // Proceedings of the 46th ACM Technical Symposium on Computer Science Education – SIGCSE ’15. – 2015. https://doi.org/10.1145/2676723.2677258.

88. Petrillo F., Mandian H., Yamashita A., Khomh F., Gueheneuc Y. G. How Do Developers Toggle Breakpoints? Observational Studies // 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). – 2017. https://doi.org/10.1109/qrs.2017.39.

89. Beller M., Spruit N., Spinellis D., Zaidman A. On the dichotomy of debugging behavior among programmers // Proceedings of the 40th International Conference on Software Engineering. – 2018. https://doi.org/10.1145/3180155.3180175.

90. Bellman C., Seet A., Baysal O. Studying developer build issues and debugger usage via timeline analysis in visual studio IDE // Proceedings of the 15th International Conference on Mining Software Repositories – MSR ’18. – 2018. https://doi.org/10.1145/3196398.3196463.

91. Damevski K., Chen H., Shepherd D., Pollock L. Interactive exploration of developer interaction traces using a hidden Markov model // Proceedings of the 13th International Workshop on Mining Software Repositories – MSR ’16. – 2016. https://doi.org/10.1145/2901739.2901741.

92. Petrillo F., Soh Z., Khomh F., Pimenta M., Freitas C., Gueheneuc Y. G. Understanding interactive debugging with Swarm Debug Infrastructure // 2016 IEEE 24th International Conference on Program Comprehension (ICPC). – 2016. https://doi.org/10.1109/icpc.2016.7503740.

93. Luxton-Reilly A., McMillan E., Stevenson E., Tempero E., Denny P. Ladebug: an online tool to help novice programmers improve their debugging skills // Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education – ITiCSE 2018. – 2018. https://doi.org/10.1145/3197091.3197098.

94. Lin Y. T., Wu C. C., Hou T. Y., Lin Y. C., Yang F. Y., Chang C. H. Tracking Students’ Cognitive Processes During Program Debugging – An Eye-Movement

Approach // IEEE Transactions on Education. – 59 (3). – 2016. – pp. 175–186. https://doi.org/10.1109/te.2015.2487341.

95. Van der Aalst W. Process Mining // ACM Transactions on Management Information Systems. – 3 (2). – 2012. – pp. 1–17. https://doi.org/10.1145/2229156.2229157.

96. Van der Aalst W., Adriansyah A., de Medeiros A. K. A., Arcieri F., Baier T., Blickle T. et. al. Process Mining Manifesto // Lecture Notes in Business Information Processing. – 2012. – pp. 169–194. https://doi.org/10.1007/978-3-642-28108-2_19.

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

98. Ardimento P., Bernardi M. L., Cimitile M., Maggi F. M. Evaluating Coding Behavior in Software Development Processes: A Process Mining Approach // 2019 IEEE/ACM International Conference on Software and System Processes (ICSSP). – 2019. https://doi.org/10.1109/icssp.2019.00020.

99. Sebu M. L., Ciocarlie H. Applied process mining in software development // 9th IEEE International Symposium on Applied Computational Intelligence and Informatics. – 2014. – pp. 55–60. https://doi.org/10.1109/SACI.2014.6840098.

100. Poncin W., Serebrenik A., Brand M. V. D. Process Mining Software Repositories // 2011 15th European Conference on Software Maintenance and Reengineering. – 2011. https://doi.org/10.1109/csmr.2011.5.

101. Sebu M. L., Ciocarlie H. Applied process mining in software development // 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI). – 2014. https://doi.org/10.1109/saci.2014.6840098.

102. Caldeira J., Abreu F. B. Software Development Process Mining: Discovery, Conformance Checking and Enhancement // 10th International Conference on the Quality of Information and Communications Technology (QUATIC). – 2016. https://doi.org/10.1109/quatic.2016.061.

103. Wil M. P. van der Aalst. Big software on the run: in vivo software analytics based on process mining (keynote) // ICSSP. – 2015. – pp. 1–5.

104. Leemans M., van der Aalst W. M. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems // 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). – 2015. – pp. 44–53.

105. Liu C., van Dongen B. F., Assy N., van der Aalst W. M. P. Component behavior discovery from software execution data // 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016. – 2016. – pp. 1–8.

106. Shynkarenko V., Zhevago O. Visualization of program development process // IEEE 14th International Conference on Computer Sciences and Information Technologies. – 2019. – pp. 142–145. https://doi.org/10.1109/stc-csit.2019.8929774.

107. Жеваго О. О., Шинкаренко В.І. Дослідження методів відстеження процесів розробки та відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XI Міжнародної науково-практичної конференції. – 2017. – С. 103.

108. Жеваго О. О., Шинкаренко В.І. Дослідження стилю відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIII Міжнародної науково-практичної конференції. – 2019. – С. 73.

109. Shynkarenko V., Zhevaho O. Constructive modeling of the software development process for modern code review // IEEE 15th International Conference on Computer Sciences and Information Technologies. – 2020. – pp. 392–395. https://doi.org/10.1109/CSIT49958.2020.9322002.

110. 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). – 2020. – pp. 29–38. https://doi.org/10.15587/1729-4061.2020.215090.

111. Шинкаренко В. І., Жеваго О. О. Експериментальні дослідження процесів налагодження комп'ютерних програм студентами з використанням Process Mining

// Вісник Херсонського національного технічного університету – 2021 – С. 83–98. https://doi.org/10.35546/kntu2078-4481.2021.3.10.

112. Жеваго О. О., Шинкаренко В.І. Виявлення навичок відлагодження програм методом підсіву помилок // Інформаційні Технології в Металургії та Машинобудуванні: тези Міжнародної науково-технічної конференції ІТММ 2021. – 2021. – С. 339–340.

113. Жеваго О. О., Шинкаренко В. І. Аналіз та вдосконалення навчального процесу з програмної інженерії з використанням методів Process Mining // Сучасні тенденції розвитку інформаційних систем і телекомунікаційних технологій: тези III Міжнародної науково-практичної конференції. – 2021. – С. 73–74.

114. Zhevaho O. O. An overview of tools for collecting data on software development and debugging processes from integrated development environments // Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту – 2021 – С. 24–37. https://doi.org/10.15802/stp2021/242042.

115. 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). – 2014. – pp. 103–116. https://doi.org/10.1016/j.infsof.2013.10.001.

116. 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). – 2014. – pp. 655–662. https://doi.org/10.1007/s10559-014-9655-z.

117. Shynkarenko V. I., Ilman V. M. Constructive-Synthesizing Structures and Their Grammatical Interpretations. II. Refining Transformations // Cybernetics and Systems Analysis. – 50 (6). – 2014. – pp. 829–841. https://doi.org/10.1007/s10559- 014-9674-9.

118. 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). – 2009. – pp. 329–339. https://doi.org/10.1007/s10559-009-9118-0.

119. 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). – 2009. – pp. 544–550. https://doi.org/10.1007/s10559-009-9122-4.

120. Kabaale E., Wen L., Wang Z., Rout T. An axiom based metamodel for software process formalisation: An ontology approach // Communications in Computer and Information Science. – 2017. – pp. 226–240. https://doi.org/10.1007/978-3-319-67383-7_17.

121. Savchuk T., Pryimak N. Modeling of software development process with the Markov processes // Eastern-European Journal of Enterprise Technologies. – 2017. – pp. 33–38. https://doi.org/10.15587/1729-4061.2017.103340.

122. Shinkarenko V. I., Zhevago O. O. Generating university course timetable using constructive modeling // Radio Electronics, Computer Science, Control. – 2019. – pp. 152–162. https://doi.org/10.15588/1607-3274-2019-3-17.

123. Жеваго О. О., Шинкаренко В. І. Составление расписания занятий на основе конструктивного моделирования // Проблеми математичного моделювання: Матеріали Всеукраїнської науково-методичної конференції, м. Кам'янське. – 2018. – С. 59–63.

124. Shynkarenko V., Lytvynenko K., Chyhir R., Nikitina I. Modeling of Lightning Flashes in Thunderstorm Front by Constructive Production of Fractal // Advances in Intelligent Systems and Computing. – 2019. – pp. 173–185. https://doi.org/10.1007/978-3-030-33695-0_13.

125. Shynkarenko V. I., Vasetska T. M. Modeling the Adaptation of Compression Algorithms by Means of Constructive-Synthesizing Structures // Cybernetics and Systems Analysis. – 51 (6). – 2015. – pp. 849–862. https://doi.org/10.1007/s10559-015-9778-x.

126. Kuropiatnyk O., Shynkarenko V. Text borrowings detection system for natural language structured digital documents // CEUR Workshop Proceedings. – 2020. – pp. 294–305.

127. Skalozub V., Ilman V., Shynkarenko V. Development of ontological support of constructive-synthesizing modeling of information systems // Eastern-European Journal of Enterprise Technologies. – 2017. – pp. 58–69. https://doi.org/10.15587/1729-4061.2017.119497.

128. Skalozub V., Ilman V., Shynkarenko V. Ontological support formation for constructive-synthesizing modeling of information systems development processes // Eastern-European Journal of Enterprise Technologies. – 2018. – pp. 55–63. https://doi.org/10.15587/1729-4061.2018.143968.

129. Жеваго О. О., Шинкаренко В.І. Конструктивне моделювання та аналіз процесів розробки і відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIV Міжнародної науково-практичної конференції. – 2020. – С. 129.

130. Verbeek H. M. W., Buijs J. C. A. M., Van Dongen B. F., Van Der Aalst W. M. P. XES, XESame, and ProM 6 // Lecture Notes in Business Information Processing. – 2011. – pp. 60–75. https://doi.org/10.1007/978-3-642-17722-4_5.

131. Van der Aalst W. Process mining: Data science in action // Springer, Berlin, Heidelberg, Germany. – 2016. – pp. 1–467. https://doi.org/10.1007/978-3-662-49851-4.

132. Leemans S. J. J., Poppe E., Wynn M. T. Directly Follows-Based Process Mining: Exploration & a Case Study // International Conference on Process Mining. – 2019. – pp. 25–32. https://doi.org/10.1109/ICPM.2019.00015.

133. Leemans S. J. J., Fahland D., van der Aalst W. M. P. Scalable process discovery and conformance checking // Software & Systems Modeling. – 17 (2). – 2018. – pp. 599–631. https://doi.org/10.1007/s10270-016-0545-x.

ДОДАТОК А
СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ ТА ВІДОМОСТІ ПРО АПРОБАЦІЮ РЕЗУЛЬТАТІВ ДИСЕРТАЦІЙНОЇ РОБОТИ

Праці включені до МНБД Scopus та Web of Science:

1. Shynkarenko, V., Zhevaho, O. Development of a toolkit for analyzing software debugging processes using the constructive approach // Eastern-European Journal of Enterprise Technologies. – 2020. – pp. 29–38. https://doi.org/10.15587/1729-4061.2020.215090. *Стаття у фаховому журналі, що індексується у МНБД Scopus та має Q3 за даними Scimago Journal & Country Rank, зараховується як дві публікації.*

2. Shinkarenko V. I., Zhevago O. O. Generating university course timetable using constructive modeling // Radio Electronics, Computer Science, Control. – 2019. – pp. 152–162. https://doi.org/10.15588/1607-3274-2019-3-17. *Стаття у фаховому журналі, що індексується у МНБД Web of Science.*

3. Shynkarenko V., Zhevaho O. Constructive modeling of the software development process for modern code review // IEEE 15th International Conference on Computer Sciences and Information Technologies. – 2020. – pp. 392–395. https://doi.org/10.1109/CSIT49958.2020.9322002. *Матеріали конференції, індексується у МНБД Scopus.*

4. Shynkarenko, V., Zhevago, O. Visualization of program development process // IEEE 14th International Conference on Computer Sciences and Information Technologies. – 2019. – pp. 142–145. https://doi.org/10.1109/stc-csit.2019.8929774. *Матеріали конференції, індексується у МНБД Scopus.*

Праці у фахових виданнях затверджених МОН України:

5. Zhevaho O. O. An overview of tools for collecting data on software development and debugging processes from integrated development environments // Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту – 2021 – С. 24–37. https://doi.org/10.15802/stp2021/242042.

6. Шинкаренко В. І., Жеваго О. О. Експериментальні дослідження процесів налагодження комп'ютерних програм студентами з використанням Process Mining // Вісник Херсонського національного технічного університету – 2021 – С. 83–98. https://doi.org/10.35546/kntu2078-4481.2021.3.10.

Матеріали наукових конференцій:

7. Жеваго О. О., Шинкаренко В.І. Дослідження методів відстеження процесів розробки та відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XI Міжнародної науково-практичної конференції, м. Дніпро. – 2017. – С. 103.

8. Жеваго О. О., Шинкаренко В.І. Дослідження стилю відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIII Міжнародної науково-практичної конференції, м. Дніпро. – 2019. – С. 73.

9. Жеваго О. О., Шинкаренко В.І. Конструктивне моделювання та аналіз процесів розробки і відлагодження програм // Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті: тези XIV Міжнародної науково-практичної конференції, м. Дніпро. – 2020. – С. 129.

10. Жеваго О. О., Шинкаренко В.І. Аналіз та вдосконалення навчального процесу з програмної інженерії з використанням методів Process Mining // Сучасні тенденції розвитку інформаційних систем і телекомунікаційних технологій: тези III Міжнародної науково-практичної конференції, м. Київ. – 2021. – С. 73–74.

11. Жеваго О. О., Шинкаренко В. І. Виявлення навичок відлагодження програм методом підсіву помилок // Інформаційні Технології в Металургії та Машинобудуванні: тези Міжнародної науково-технічної конференції ІТММ 2021, м. Дніпро. – 2021. – С. 339–340.

12. Жеваго О. О., Шинкаренко В. І. Составление расписания занятий на основе конструктивного моделирования // Проблеми математичного моделювання: Матеріали Всеукраїнської науково-методичної конференції, м. Кам’янське. – 2018. – С. 59–63.

ДОДАТОК Б
АКТ ВПРОВАДЖЕННЯ

ЗАТВЕРДЖУЮ

Проректор з науково-педагогічної,
економічної роботи, перспективного
та інноваційного розвитку
Дніпровського національного
університету залізничного
транспорту ім. акад. В. Лазаряна

_____ д.т.н., проф. Радкевич А. В.

«16» червня 2021 р.

АКТ

про впровадження результатів дисертаційної роботи
«Методи інтелектуального аналізу стилю розробки та відлагодження комп'ютерних програм» на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки»

Жеваго Олександра Олександровича

В рамках дисертаційної роботи розроблені інструменти для реалізації можливості вимірювання характеристик стилю роботи кожного студента як на лабораторних заняттях, так і під час самостійної роботи. Створено доповнення до середовища розробки Visual Studio в якому всі дії по розробці та відлагодженню фіксуються в журналах подій.

Розроблені програмні засоби з відстеження процесів розробки та відлагодження програмного забезпечення були використані при проведені олімпіади з відлагодження, яка була проведена в університеті 21-22 травня згідно наказу ректора № 193 а-г. від 12.05.2021 р. і планується проводитись щорічно.

Голова оргкомітету олімпіади,
декан факультету
«Комп'ютерні технології і системи»
доктор технічних наук, професор В. В. Скалозуб