МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Кафедра програмної інженерії

Методичні вказівки до лабораторних робіт

з дисципліни

«Раціональний уніфікований процес проєктування програмного забезпечення»

для здобувачів другого (магістерського) рівня вищої освіти
ОПП «Інженерія програмного забезпечення»

Тернопіль — 2026

УДК 004.41(076.5)

Укладачі:
І.Я. Мудрик, PhD
М.Р Петрик., д.ф.-м.н., проф.

Рецензент:
І.О. Боднарчук, к.т.н., доцент
завідувач кафедри комп'ютерних наук ТНТУ ім. І. Пулюя

Розглянуто й затверджено на засіданні кафедри програмної інженерії.
Протокол № 12 від 30.03.2026.

Розглянуто й затверджено на засіданні методичної комісії факультету комп'ютерно-інформаційних систем та програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 4 від 09.04.2026.

Методичні вказівки до лабораторних робіт з дисципліни «Раціональний уніфікований процес проєктування програмного забезпечення» для здобувачів другого (магістерського) рівня вищої освіти ОПП «Інженерія програмного забезпечення» всіх форм навчання / Укладачі: Петрик М.Р., Мудрик І.Я.,– Тернопіль: ТНТУ ім. І. Пулюя, 2026 – 58 с.

Відповідальний за випуск: І.Я. Мудрик, PhD

АНОТАЦІЯ

Методичні вказівки містять практичні рекомендації для допомоги студентам у вивченні дисципліни «Раціональний уніфікований процес проєктування ПЗ», зокрема передбачають дослідження методології ітеративної розробки RUP, візуального моделювання за допомогою мови UML та використання методів практико-орієнтованого навчання. Практикум охоплює ключові робочі процеси: бізнес-моделювання та аналіз предметної області; управління вимогами та розробку моделі варіантів використання (прецедентів); об'єктно-орієнтований аналіз та проєктування архітектури (побудова діаграм класів, послідовності, станів, компонентів та розгортання); перехід від концептуальних UML-моделей до програмної реалізації (наприклад, з використанням сучасних екосистем розробки на зразок Node.js чи React) із дотриманням принципів SOLID. Роботи розроблено з використанням сучасних CASE-засобів та інструментальних середовищ візуального проєктування (наприклад, IBM Rational Software Architect, Enterprise Architect або StarUML).

Предмет орієнтований на студентів спеціальності F2 «Інженерія програмного забезпечення» для набуття практичних навичок ітеративного проєктування та управління життєвим циклом ПЗ. Лабораторні роботи структуровані відповідно до основних фаз RUP (Початковий етап, Уточнення, Побудова, Впровадження) та охоплюють увесь цикл розробки програмного забезпечення: від аналізу вимог та архітектурного проєктування до реалізації, тестування та розгортання системи. Особливу увагу приділено командній роботі, використанню систем контролю версій, створенню технічної документації на основі артефактів RUP та дотриманню професійних стандартів.

Методичні вказівки призначені для здобувачів другого (магістерського) рівня вищої освіти, які навчаються за освітньо-професійною програмою «Інженерія програмного забезпечення». Вони також будуть корисними для студентів технічних спеціальностей, що вивчають основи програмної інженерії та розробки програмного забезпечення.

Ключові слова: програмна інженерія, Раціональний уніфікований процес, RUP, UML, ітеративна розробка, архітектурне проєктування, візуальне моделювання, життєвий цикл розробки.

ЗМІСТ

ВСТУП 5

……

ЗАГАЛЬНІ ВИМОГИ ДО ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ 5

Лабораторна робота 1. Огляд життєвого циклу RUP. Фаза «Початковий етап» (Inception) 12

Лабораторна робота 2. Управління вимогами в RUP. Документування та специфікація 18

Лабораторна робота 3. Бізнес-моделювання. Розробка моделі прецедентів (Use Case Model) 22

Лабораторна робота 4. Робочий потік «Аналіз та проєктування» (Analysis & Design) 26

Лабораторна робота 5. Деталізація моделі проєктування та застосування принципів SOLID 29

Лабораторна робота 6. Проєктування архітектури програмної системи (фаза Elaboration) 33

Лабораторна робота 7. Робочі потоки «Реалізація» (Implementation) та «Впровадження» (Deployment) 37

Лабораторна робота 8. Управління ризиками проєкту та валідація вимог (фаза Transition). 42

ВИСНОВКИ 47

……

РЕКОМЕНДОВАНІ ДЖЕРЕЛА…….. 48

Додатки:

Додаток А. Приклад оформлення звіту лабораторної роботи.

ВСТУП

Деталі щодо створення візуальних моделей (UML-діаграм), підготовки проєктних артефактів, оформлення програмного коду та загального змісту звітів з лабораторних робіт узгоджуються з викладачем у системі електронного навчання відповідного курсу. Приклад оформлення звіту з титульним аркушем, вимогами до структури та методичними рекомендаціями наведено в додатку А.

Усі лабораторні роботи повинні бути виконані з дотриманням принципів академічної доброчесності, відповідно до Положення про академічну доброчесність учасників освітнього процесу Тернопільського національного технічного університету імені Івана Пулюя. Особлива увага звертається на самостійність проєктування архітектурних рішень, побудови об'єктно-орієнтованих моделей та недопущення плагіату при формуванні супровідної проєктної документації.

Загальні вимоги до лабораторних робіт з дисципліни

Лабораторні роботи передбачають практичне дослідження методології ітеративної розробки програмного забезпечення Rational Unified Process (RUP), а також базових принципів об'єктно-орієнтованого аналізу та архітектурного проєктування за допомогою мови UML.

Для успішного виконання завдань лабораторних робіт студенту необхідно:

- Ознайомитися з теоретичним матеріалом, що розкриває особливості конкретної фази життєвого циклу RUP (Початковий етап, Уточнення, Побудова або Впровадження) та правила побудови відповідних UML-діаграм.

- Провести дослідження предметної області та виконати проєктування відповідно до індивідуального завдання (варіанту або теми наскрізного проєкту), застосовуючи спеціалізовані CASE-засоби візуального моделювання.

- Сформувати проєктні артефакти (моделі варіантів використання, діаграми класів, діаграми взаємодії, специфікації вимог тощо).

- Відобразити результати роботи у звіті, який повинен містити постановку задачі, побудовані візуальні моделі з їх детальним описом, обґрунтування прийнятих архітектурних рішень та висновки до лабораторної роботи..

Порядок здачі та захисту лабораторних робіт

- Вибір завдання: Варіант індивідуального завдання (або тема проєкту) обирається відповідно до порядкового номера студента у списку академічної підгрупи.

- Подання результатів: Звіт з лабораторної роботи необхідно у встановлені терміни завантажити до відповідної скриньки завдань у системі електронного дистанційного навчання.

- Склад пакету матеріалів: Результатом виконання кожної роботи є оформлений звіт та супровідні проєктні файли. Завантажте ключові артефакти поточного етапу розробки:

- PDF-звіт (має містити постановку задачі, побудовані UML-діаграми, їх опис та висновки);

- Посилання на додаткові матеріали (файли моделей із вибраного CASE-засобу, посилання на репозиторій із кодом або інші робочі документи).

- **Захист роботи:** Після завантаження всіх артефактів у систему дистанційного навчання студент повинен захистити готовий звіт у викладача, продемонструвавши розуміння створених моделей та обґрунтувавши прийняті архітектурні рішення.

Узагальнена стуктура звіту:

1. 1. **Титульний аркуш**, який повинен включати такі елементи:

- назву ЗВО: Тернопільський національний технічний університет імені Івана Пулюя;
- назву факультету: Факультет комп’ютерно-інформаційних систем і програмної інженерії;
- назву кафедри: Кафедра програмної інженерії;
- назву дисципліни, порядковий номер та тему лабораторної роботи;
- інформацію про виконавця: номер академічної групи, прізвище та ініціали студента;
- інформацію про викладача, який приймає роботу;
- місто та рік захисту роботи (наприклад, Тернопіль — 2026).

2. **Основна частина звіту**, яка повинна містити:

Мету роботи та завдання (із зазначенням номера варіанту або теми проєкту).

Опис предметної області (коротка характеристика бізнес-процесів або проблеми, що вирішується).

Візуальні моделі (UML-діаграми), побудовані в CASE-засобах, відповідно до етапу RUP:

1. модель варіантів використання (Use Case Diagram);
2. діаграму класів (Class Diagram — у рамках поточної ітерації, необхідний мінімум сутностей) та опис базових класів (атрибути, методи);
3. діаграми взаємодії (наприклад, діаграму послідовності — Sequence Diagram) для ілюстрації основних методів та сценаріїв.

Програмну реалізацію та архітектуру:

- компонентну модель (опис структури коду або діаграма компонентів);
- програмний код, згенерований на основі діаграми класів у середовищі проєктування (мови C++, C#, Java тощо) або написаний власноруч;
- код має бути розбитий на модулі/компоненти, дотримуватися єдиного стилю форматування (Clean Code) та супроводжуватися необхідними коментарями.

Отримані результати виконання кожного з пунктів завдання (скріншоти згенерованих моделей, лістинги коду, результати тестування тощо).

3. Висновки

Мають містити підсумки виконаної роботи, відповіді на поставлену мету та аналіз отриманих результатів (які патерни чи архітектурні рішення були застосовані та чому).

Вимоги до оформлення та інструментарію

Програмне середовище: Лабораторні роботи виконуються у спеціалізованих CASE-засобах візуального проєктування. Можуть використовуватися IBM Rational Software Architect (або аналоги: Sparx Enterprise Architect, StarUML, тощо, затверджені викладачем).

Форматування тексту звіту:

- Звіт оформляється у текстовому редакторі на аркушах формату А4.
- **Поля:** верхнє та нижнє — 20 мм, праве — 15 мм, ліве — 25 мм.
- **Основний текст:** шрифт Times New Roman, 14 пт; вирівнювання — по ширині; міжрядковий інтервал — 1,5.

- **Лістинги коду:** шрифт Courier (або інший моноширинний, наприклад, Consolas), 10 пт; вирівнювання — по лівому краю; міжрядковий інтервал — 1,0.
- Усі рисунки (діаграми, скріншоти) та лістинги коду повинні мати нумерацію, підписи та обов'язкові посилання на них у тексті звіту.

Порядок подачі: Звіт до кожної лабораторної роботи слід роздрукувати, скріпити та представити викладачу до захисту. Електронний варіант звіту (у форматі PDF разом із файлами моделей/коду) необхідно обов'язково завантажити у скриньку відповідного завдання в системі дистанційного навчання.

СТРУКТУРА ЛАБОРАТОРНИХ РОБІТ:

Лабораторна робота 1. Огляд життєвого циклу RUP. Фаза «Початковий етап» (Inception).
Спрямування: Дослідження ітеративних методологій розробки. Визначення меж проєкту, ідентифікація ключових зацікавлених осіб (Stakeholders) та формування початкового бізнес-контексту програмної системи.

Лабораторна робота 2. Управління вимогами в RUP. Документування та специфікація.
Спрямування: Робочий потік «Управління вимогами» (Requirements). Формування документа «Бачення» (Vision), розробка глосарію термінів предметної області, збір та класифікація функціональних і нефункціональних вимог.

Лабораторна робота 3. Бізнес-моделювання. Розробка моделі прецедентів (Use Case Model).
Спрямування: Дослідження предметної області. Визначення акторів системи та створення концептуальної моделі варіантів використання (прецедентів) за допомогою візуального моделювання в UML.

Лабораторна робота 4. Робочий потік «Аналіз та проєктування» (Analysis & Design).
Спрямування: Розробка «Моделі аналізу». Перехід від вимог до концептуальної архітектури: побудова базової діаграми класів (Class Diagram) та моделювання поведінки системи за допомогою діаграм взаємодії (Sequence Diagram / Communication Diagram).

Лабораторна робота 5. Деталізація моделі проєктування та застосування принципів SOLID.
Спрямування: Розподіл обов'язків між об'єктами. Перехід від концептуальних

класів до проєктних (Design Classes) із застосуванням патернів проєктування (наприклад, GRASP) для забезпечення гнучкості та модульності програмного коду.

Лабораторна робота 6. Проєктування архітектури програмної системи (фаза Elaboration).
Спрямування: Представлення архітектури за моделлю «4+1». Визначення критеріїв хорошої архітектури, поділ системи на підсистеми та розробка компонентної моделі (Component Diagram).

Лабораторна робота 7. Робочі потоки «Реалізація» (Implementation) та «Впровадження» (Deployment).
Спрямування: Проєктування фізичної архітектури та топології мережі (Deployment Diagram). Інтеграція сучасних DevOps-практик: управління конфігураціями, версіонування (CI/CD) та підготовка релізних артефактів.

Лабораторна робота 8. Управління ризиками проєкту та валідація вимог (фаза Transition).
Спрямування: Робочі потоки «Управління проєктом» (Project Management) та «Тестування» (Test). Ідентифікація, оцінка та пом'якшення архітектурних ризиків. Верифікація розроблених моделей на відповідність початковим вимогам замовника.

Інструментарій та Ресурси

Програмне забезпечення для візуального моделювання та проєктування (CASE-засоби):

- **IBM Rational Software Architect Designer** — класичний комплексний інструмент для об'єктно-орієнтованого аналізу, проєктування та генерації коду в екосистемі RUP.
- **PlantUML** або **Mermaid.js** — інструменти для створення UML-діаграм на основі тексту (підхід «Архітектура як код» / Architecture as Code).
- **Structurizr** — засіб для багаторівневого моделювання архітектури програмного забезпечення за моделлю C4.
- **Draw.io (diagrams.net)** або **Lucidchart** — гнучкі хмарні сервіси для створення концептуальних UML-діаграм, схем баз даних та діаграм розгортання.
-

Середовища розробки (IDE) для реалізації архітектурних рішень:

- **Локальні IDE:** Visual Studio Code, а також спеціалізовані інтегровані середовища розробки від JetBrains (WebStorm, PyCharm, IntelliJ IDEA).
- **Хмарні середовища розробки (Cloud IDE):** Project IDX або GitHub Codespaces — для швидкого прототипування, ізольованого тестування та зручної командної роботи над кодом.

Платформи та фреймворки (для практичної реалізації на фазі «Побудова»):

- **Серверна частина (Backend):** платформи Node.js та Python для реалізації серверної бізнес-логіки, проєктування API та мікросервісної архітектури.
- **Клієнтська частина (Frontend):** бібліотека React для проєктування користувацьких інтерфейсів та реалізації архітектурних патернів (наприклад, MVC, Flux/Redux).
- **Хмарна інфраструктура (BaaS / Serverless):** платформи на зразок Firebase для швидкого розгортання баз даних, налаштування автентифікації та побудови безсерверної (serverless) архітектури.

Інструменти DevOps та контейнеризації (для фаз Побудови та Впровадження):

- **Docker:** для контейнеризації, ізоляції середовищ та пакування мікросервісів.
- **Git та платформи GitHub / GitLab:** для управління версіями вихідного коду (Version Control), командної взаємодії та налаштування базових конвеєрів безперервної інтеграції і доставки (CI/CD пайплайнів).

Інструменти на базі штучного інтелекту (AI-асистенти):

- **Генеративний ШІ:** використання сучасних AI-інструментів для допомоги в аналізі архітектурних патернів, рефакторингу, оптимізації та генерації шаблонного (boilerplate) коду. *Примітка: використання таких інструментів має суворо відповідати принципам академічної доброчесності.*

Сучасні методології та архітектурні підходи (як еволюція та доповнення RUP):

- **Інтеграція Agile-практик (Scrum, Kanban):** адаптація ітеративної природи RUP до сучасних гнучких методологій розробки для швидшої доставки цінності.

- **Domain-Driven Design (DDD):** предметно-орієнтоване проєктування для глибокого аналізу бізнес-вимог, визначення обмежених контекстів (Bounded Contexts) та проєктування меж мікросервісів.
- **Cloud-Native підходи:** проєктування сучасних розподілених систем, мікросервісних (Microservices) та безсерверних (Serverless) архітектур.
- **DevOps / DevSecOps:** впровадження інженерних практик безперервної інтеграції, тестування та розгортання, а також підходу «Інфраструктура як код» (IaC).
- **Модель C4:** використання багаторівневого підходу (Context, Containers, Components, Code) як сучасної альтернативи або доповнення до класичних діаграм UML для документування архітектури програмного забезпечення.

Обчислювальні ресурси:

- Стандартний персональний комп’ютер або ноутбук із характеристиками, достатніми для запуску сучасних середовищ розробки (IDE) та систем віртуалізації (наприклад, Docker Desktop). Обов’язковою умовою є наявність стабільного доступу до мережі Інтернет для роботи з хмарними сервісами, віддаленими репозиторіями та системами управління проєктами.

ЛАБОРАТОРНА РОБОТА 1.

Тема: Огляд життєвого циклу RUP. Фаза «Початковий етап» (Inception). Бізнес-моделювання та формування бачення проєкту.

Мета роботи: Ознайомитися з архітектурою життєвого циклу методології Rational Unified Process (RUP). Навчитися проводити первинний аналіз предметної області, визначати межі проєкту, ідентифікувати зацікавлених осіб (стейкхолдерів) та формувати базове бачення майбутнього програмного продукту.

Завдання роботи:

1. **Вибір предметної області:** Обрати тему для наскрізного проєктування (наприклад, система електронної комерції, регіональний новинний портал або сучасна веб-платформа зі стеком Node.js/React). Виконати аналіз найпоширеніших методологій ПЗ (SCRUM, KANBAN, DSDM, MSF, RUP), XP, Scrumban, Agile Modeling.
2. **Аналіз проблеми:** Сформулювати основну бізнес-проблему, яку має вирішити розробка програмного забезпечення.
3. **Аналіз стейкхолдерів:** Визначити перелік ключових зацікавлених осіб (користувачі, замовники, адміністратори, розробники) та описати їхні основні потреби.
4. **Розробка документа «Бачення» (Vision):** Створити чорновий варіант документа, який описує концепцію продукту.
5. **Створення глосарію:** Сформувати глосарій ключових термінів обраної предметної області для забезпечення єдиної термінології в команді.
6. **Оформлення звіту:** Зібрати всі артефакти у звіт та підготуватися до захисту.

Теоретичні відомості: Методологія Rational Unified Process (RUP) — це ітеративний процес розробки програмного забезпечення, керований варіантами використання (Use-Case Driven), орієнтований на архітектуру (Architecture-Centric) та спрямований на зниження ризиків. Життєвий цикл RUP складається з чотирьох послідовних фаз:

1. **Початковий етап (Inception):** Визначення меж проєкту, оцінка ризиків, формування бізнес-кейсу.
2. **Уточнення (Elaboration):** Проєктування базової архітектури, деталізація вимог, зняття основних технічних ризиків.

3. **Побудова (Construction):** Розробка коду (наприклад, реалізація клієнтської частини та серверної логіки), тестування компонентів.
4. **Впровадження (Transition):** Передача продукту кінцевим користувачам, розгортання, бета-тестування.

Особливості фази Inception: Головна мета початкового етапу — досягти згоди між усіма стейкхолдерами щодо цілей проєкту та його меж. На цьому етапі не пишеться код і не проєктується детальна архітектура. Замість цього аналізується бізнес-контекст. Основні артефакти фази:

- *Документ Vision (Бачення):* Високорівневий опис продукту.
- *Глосарій (Glossary):* Словник термінів, що є основою для подальшого застосування принципів Domain-Driven Design (DDD) — формування «єдиної мови» (Ubiquitous Language).
- *Початкова модель варіантів використання:* Визначення основних акторів та ключових прецедентів (детально розглядається в наступних роботах).

Методологія - це система принципів, а також сукупність ідей, понять, методів, способів і засобів, що визначають стиль розробки програмного забезпечення. Методологія - це реалізація певного стандарту. Самі стандарти лише говорять про те, що повинно бути, залишаючи свободу вибору та адаптації.

Конкретні речі реалізується через обрану методологію. Саме вона визначає, як буде виконуватися розробка. Існує багато успішних методологій розробки програмного забезпечення. Вибір конкретної методології залежить від розміру команди, від специфіки і складності проекту, від стабільності і зрілості процесів в компанії і від особистих якостей співробітників.

Методології є ядромтеорії управління процесу розробкою програмного забезпечення. До існуючої класифікації залежно від обраної моделі життєвого циклу (водоспадні та ітераційні методології) додалася більш загальна класифікація на прогнозовані і адаптивні методології.

Прогнозовані методології фокусуються на детальному плануванні майбутнього. Відомі заплановані завдання і ресурси на весь термін проекту. Команда важко реагує на можливі зміни. План оптимізований виходячи зі складу робіт та існуючих вимог. Зміна вимог може призвести до суттєвої зміни плану, а також дизайну проекту. Часто створюється спеціальний комітет з «управління змінами», щоб у проекті враховувалися тільки найважливіші вимоги.

Адаптивні методології націлені на подолання очікуваної неповноти вимог і їх постійної зміни. Коли змінюються вимоги, команда розробників теж змінюється. Команда, що бере участь в адаптивній розробці, важко може передбачити майбутнє проекту. Існує точний план лише на найближчий час. Більш віддалені в часі плани існують лише як декларації про цілі проекту, очікувані витрати і результати.

SCRUM-методологія, призначена для невеликих команд (до 10 осіб). Весь проект поділяється на ітерації (спринти) тривалістю 30 днів кожний. Вибирається список функцій системи, які планується реалізувати протягом наступного спринту. Найважливіші умови - незмінність обраних функцій під час виконання однієї ітерації і суворе дотримання термінів випуску чергового релізу, навіть якщо до його випуску не вдасться реалізувати весь запланований функціонал. Керівник розробки проводить щоденні 20 хвилинні наради, які так і називають - scrum, результатом яких є визначення функції системи, реалізованих за попередній день, виниклі труднощі і план на наступний день. Такі наради дозволяють постійно відслідковувати хід проекту, швидко виявляти виниклі проблеми і оперативно на них реагувати.

KANBAN - гнучка методологія розробки програмного забезпечення, орієнтована на завдання.

Основні правила:

• візуалізація розробки:

• поділ роботи на завдання;

• використання поміток про стан завдання в розробці;

• обмеження робіт, що виконуються одночасно, на кожному етапі розробки;

• вимір часу циклу (середній час на виконання одного завдання) і оптимізація процесу.

Переваги KANBAN:

• зменшення числа паралельно виконуваних завдань значно зменшує час виконання кожної окремої задачі;

• швидке виявлення проблемних завдань;

• обчислення часу на виконання усередненої завдання.

DYNAMIC SYSTEM DEVELOPMENT METHOD з'явився в результаті роботи консорціум з 17 англійських компаній. Ціла організація займається розробкою посібників за цією методологією, організацією навчальних курсів, програм акредитації і т.п. Крім того, цінність DSDM має грошовий еквівалент.

Все починається з вивчення здійсненності програми і області її застосування. У першому випадку, ви намагаєтеся зрозуміти, чи підходить DSDM для даного проекту. Вивчати область застосування програми передбачається на короткій серії семінарів, де програмісти дізнаються про ту сферу бізнесу, для якої їм належить працювати. Тут же обговорюються основні положення, що стосуються архітектури майбутньої системи і план проекту.

Далі процес ділиться на три взаємопов'язаних циклу: цикл функціональної моделі відповідає за створення аналітичної документації та прототипів, цикл проектування і конструювання - за приведення системи в робочий стан, і нарешті, останній цикл - цикл реалізації - забезпечує розгортання програмної системи.

Базові принципи, на яких будується DSDM, це активна взаємодія з користувачами, часті випуски версій, самостійність розробників у прийнятті рішень і тестування протягом всього циклу робіт. Як і більшість інших гнучких методологій, DSDM використовує короткі ітерації, тривалістю від двох до шести тижнів кожна. Особливий наголос робиться на високій якості роботи і адаптованості до змін у вимогах.

MICROSOFT SOLUTIONS FRAMEWORK - методологія розробки програмного забезпечення, запропонована корпорацією Microsoft. MSF спирається на практичний досвід Microsoft і описує управління людьми і робочими процесами в процесі розробки рішення.

Базові концепції та принципи моделі процесів MSF:

• єдине бачення проекту - всі зацікавлені особи і просто учасники проекту повинні чітко уявляти кінцевий результат, усім має бути зрозуміла мета проекту;

• управління компромісами - пошук компромісів між ресурсами проекту, календарним графіком і реалізованими можливостями;

• гнучкість - готовність до мінливих проектним умовам;

• концентрація на бізнес-пріоритетах - зосередженість на тій віддачі і вигоді, яку очікує отримати споживач рішення;

• заохочення вільного спілкування всередині проекту;

• створення базових версії - фіксація стану будь-якого проектного артефакту, в тому числі програмного коду, плану проекту, керівництва користувача, настройки серверів і подальше ефективне управління змінами, аналітика проекту.

MSF пропонує перевірені методики для планування, проектування, розробки та впровадження успішних IT-рішень. Завдяки своїй гнучкості, масштабованості і відсутності жорстких інструкцій MSF здатний задовольнити потреби організації або проектної групи будь-якого розміру. Методологія MSF складається з принципів, моделей і дисциплін з управління персоналом, процесами, технологічними елементами і пов'язаними з усіма цими факторами питаннями, характерними для більшості проектів.

RATIONAL UNIFIED PROCESS - методологія розробки програмного забезпечення, створена компанією Rational Software.

В основі методології лежать 6 основних принципів:

• компонентна архітектура, реалізована і тестована на ранніх стадіях проекту;

• робота над проектом в згуртованої команді, ключова роль у якій належить архітекторам;

• рання ідентифікація і безперервне усунення можливих ризиків;

• концентрація на виконанні вимог замовників до виконуваній програмі;

• очікування змін у вимогах, проектних рішеннях і реалізації в процесі розробки;

• постійне забезпечення якості на всіх етапах розробки проекту.

Використання методології RUP направлено на ітеративну модель розробки. Особливість методології полягає в тому, що ступінь формалізації може змінюватися в залежності від потреб проекту. Можна по закінченні кожного етапу та кожній ітерації створювати всі необхідні документи і досягти максимального рівня формалізації, а можна створювати тільки необхідні для роботи документи, аж до повної їх відсутності. За рахунок такого підходу до формалізації процесів методологія є досить гнучкою і широко популярною. Дана методологія застосовна як в невеликих і швидких проектах, де за рахунок відсутності формалізації потрібно скоротити час виконання проекту і витрати, так і у великих і складних проектах, де потрібен високий рівень формалізму, наприклад, з метою подальшої сертифікації продукту. Ця перевага дає можливість використовувати одну і ту ж команду розробників для реалізації різних за обсягом і вимогам.

AGILE MODELING. Набір концепцій, принципів і методів (практик), який дозволяє швидко і легко виконувати проектування та документацію для проектів з розробки програмного забезпечення. Не включає в себе докладні інструкції з проектування, містить описи, як побудувати діаграми UML. Основна мета – ефективне моделювання та документація, але не включає в себе програмування і тестування, управління проектом, розгортання та обслуговування системи.

SCRUNBAN. Scrumban є структурою управління і гібрид Scrum і Kanban. Розробники працюють з історіями користувачів і намагаються зберігати ітерації якомога коротшими. Тут не існує конкретних ролей, як наприклад в Scum. Кожен член команди зберігає свою існуючу роль в проекті.

Таким чином, існує безліч різних методологій розробки програмного забезпечення, вони не універсальні і описуються різними принципами. Вибір методології розробки для конкретного проекту залежить від пропонованих вимог.

*Інфографіка - це графічне візуальне подання інформації, даних або знань, призначених для швидкого та чіткого відображення комплексної інформації

Додатки до лабораторної роботи №1

Додаток 1. Шаблон документа «Бачення» (Vision)

- **Назва проєкту:** [Назва вашої системи]
- **Формулювання проблеми:**
  - *Проблема:* [Опис проблеми]
  - *Зачіпає інтереси:* [Перелік стейкхолдерів]
  - *Її наслідком є:* [Негативні бізнес-наслідки]
  - *Успішним рішенням буде:* [Що саме дасть впровадження ПЗ]

- **Позиціонування продукту:** Для [цільова аудиторія], яким необхідно [потреба], продукт [назва] є [категорія продукту], який [ключова перевага/фіча].
- **Профілі стейкхолдерів:** Таблиця (Ім'я/Роль | Обов'язки | Критерії успіху).

Додаток 2. Шаблон Глосарію

- **Термін | Визначення | Синоніми / Примітки**

Висновки: У висновку студент повинен зазначити:

- Яку предметну область було досліджено.
- Які основні бізнес-проблеми було виявлено.
- Наскільки обраний підхід дозволяє чітко окреслити межі майбутньої програмної системи.
- Які ризики (технічні, організаційні) були ідентифіковані на початковому етапі.

Список інформаційних джерел:

1. Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.
2. Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
3. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.
4. Офіційна специфікація UML (Unified Modeling Language) від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 2

Тема: Управління вимогами в RUP. Документування програмного продукту, специфікація вимог та розбиття на релізи.

Мета роботи: Ознайомитися з робочим потоком «Управління вимогами» (Requirements Workflow) у методології Rational Unified Process. Навчитися збирати, класифікувати та документувати функціональні й нефункціональні вимоги, формувати специфікацію вимог до програмного забезпечення (SRS) та планувати ітеративні релізи продукту.

Завдання до роботи:

1. **Збір вимог:** Продовжити роботу над предметною областю, обраною в першій лабораторній роботі (наприклад, проєктування архітектури регіонального медіа-порталу, системи електронної комерції або спеціалізованої веб-платформи на стеку Node.js/React).
2. **Класифікація за моделлю FURPS+:** Сформувати перелік вимог до системи, розділивши їх на функціональні та нефункціональні категорії.
3. **Розробка специфікації (SRS):** Скласти базову Специфікацію вимог до програмного забезпечення (Software Requirements Specification), яка описує поведінку системи та її обмеження.
4. **Планування релізів:** Запропонувати план розбиття розробки продукту на ітераційні релізи (наприклад, MVP — мінімально життєздатний продукт, Release 1.0, Release 2.0), чітко вказавши, який базовий функціонал увійде до кожної ітерації.
5. **Оформлення звіту:** Зібрати сформовані артефакти (SRS та план релізів) у звіт та підготуватися до захисту.

Теоретичні відомості

Управління вимогами є одним із ключових безперервних робочих потоків (Workflows) у методології RUP. Вимога — це умова або можливість, якій повинна відповідати програмна система для задоволення потреб замовника або досягнення певного бізнес-результату. Головна мета цього етапу — трансформувати загальне «Бачення» (Vision), сформоване на початковому етапі, у конкретні технічні завдання для команди розробників.

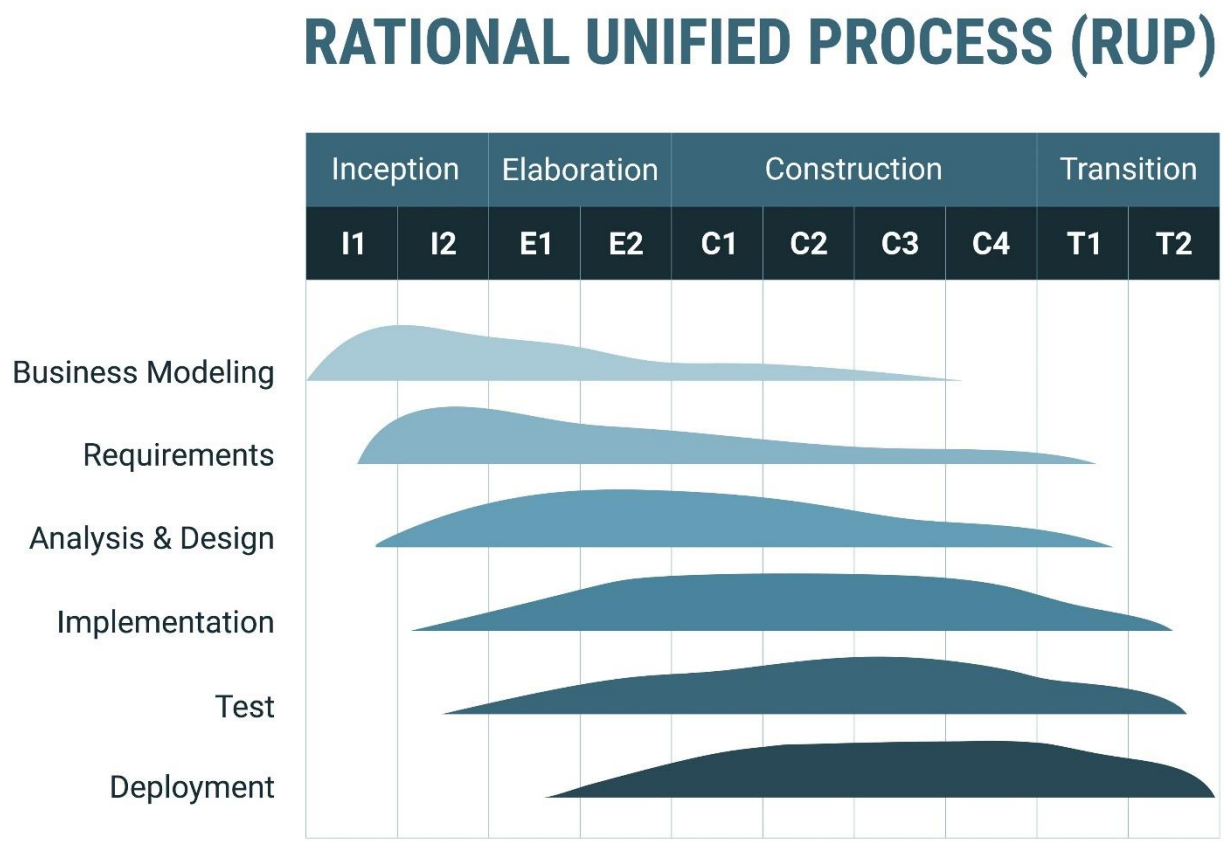

Рисунок 1. Модель RUP

Теоретичний матеріал:

У RUP та об'єктно-орієнтованому проєктуванні вимоги зазвичай класифікують за моделлю **FURPS+**, яка поділяє їх на функціональні та нефункціональні:

- **F (Functionality) — Функціональність:** Основні функції, які система повинна виконувати (наприклад, реєстрація користувача, додавання товару в кошик, обробка платежу).
- **U (Usability) — Зручність використання:** Вимоги до інтерфейсу користувача (UI/UX), доступність, час навчання роботі з системою.
- **R (Reliability) — Надійність:** Стійкість до збоїв, частота відмов, можливість відновлення після критичних помилок.
- **P (Performance) — Продуктивність:** Час відгуку системи, пропускна здатність, споживання ресурсів (наприклад, серверна частина повинна витримувати 1000 одночасних підключень).
- **S (Supportability) — Придатність до супроводу:** Легкість оновлення, тестування, масштабування та локалізації коду.

- **«+» (Додаткові обмеження):** Дизайнерські, імплементаційні (наприклад, вимога використовувати виключно реляційні бази даних), фізичні або юридичні обмеження.

Специфікація вимог до ПЗ (Software Requirements Specification - SRS) — це комплексний документ, який детально описує, як саме повинна поводитися система. SRS слугує контрактом між замовником та розробниками. Найвідомішим стандартом для написання специфікацій є ISO/IEC/IEEE 29148.

Ітеративний розвиток та розбиття на релізи Оскільки RUP є ітеративною методологією, система не розробляється цілком за один раз (на відміну від каскадної моделі). Весь процес розбивається на ітерації, кожна з яких завершується випуском внутрішнього або зовнішнього **релізу**.

- **MVP (Minimum Viable Product):** Перша робоча версія системи, яка має мінімальний набір функцій (Core Features), достатній для задоволення перших користувачів та збору зворотного зв'язку.
- **Наступні релізи:** Поступово нарощують функціонал (наприклад, додавання інтеграції зі сторонніми API, розширені аналітичні панелі, додаткові ролі користувачів), спираючись на зменшення ризиків у попередніх ітераціях.

Додатки до лабораторної роботи №2

Додаток 1. Скорочений шаблон специфікації вимог (SRS)

- **1. Вступ**
  - 1.1 Призначення документа
  - 1.2 Цільова аудиторія
- **2. Загальний опис**
  - 2.1 Перспектива продукту (контекст системи)
  - 2.2 Характеристики користувачів
  - 2.3 Обмеження та припущення (наприклад, технічні обмеження обраного фреймворку)
- **3. Системні вимоги (за моделлю FURPS+)**
  - 3.1 Функціональні вимоги (список з ідентифікаторами, наприклад, REQ-F-01, REQ-F-02)
  - 3.2 Вимоги до зручності використання (Usability)
  - 3.3 Вимоги до надійності (Reliability)
  - 3.4 Вимоги до продуктивності (Performance)
  - 3.5 Вимоги до супроводу та інші обмеження (Supportability +)

Додаток 2. Шаблон плану релізів (Release Plan)

- **Реліз 0.1 (MVP - Мінімально життєздатний продукт)**
  - *Мета:* Перевірка основної гіпотези або запуск базового потоку.
  - *Функції:* [Перелік REQ-F-...]
- **Реліз 1.0 (Базова версія для широкого загалу)**
  - *Мета:* Забезпечення стабільної роботи основного функціоналу.
  - *Функції:* [Перелік REQ-F-...]
- **Реліз 2.0 (Розширення функціоналу)**
  - *Мета:* Інтеграція додаткових сервісів та оптимізація.
  - *Функції:* [Перелік REQ-F-...]

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Які методи збору вимог були застосовані для обраної предметної області.
- З якими складнощами зіткнулися при формулюванні та вимірюванні нефункціональних вимог (наприклад, метрик продуктивності).
- За яким критерієм функціонал був розділений між MVP та наступними релізами (пріоритетність бізнес-цінності, технічна складність тощо).
- Як створена специфікація допоможе на етапі архітектурного проєктування.

Список інформаційних джерел

1. ISO/IEC/IEEE 29148:2018. *Systems and software engineering — Life cycle processes — Requirements engineering*.
2. Wiegers, K., & Beatty, J. (2013). *Software Requirements* (3rd ed.). Microsoft Press.
3. Leffingwell, D., & Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach* (2nd ed.). Addison-Wesley Professional.
4. Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 3

Тема: Дослідження предметної області. Концептуальна модель варіантів використання (Use Case Model). Аналіз вимог до продукту.

Мета роботи: Засвоїти принципи об'єктно-орієнтованого аналізу вимог за допомогою візуального моделювання. Навчитися ідентифікувати акторів (Actors) та прецеденти (Use Cases), будувати концептуальну діаграму варіантів використання в нотації UML за допомогою CASE-засобів та розробляти детальні текстові специфікації сценаріїв використання.

Завдання до роботи:

1. **Визначення меж системи (System Boundary):** Окреслити межі програмної системи, що розробляється в рамках обраної предметної області, відокремивши її від зовнішнього середовища.
2. **Ідентифікація акторів:** Визначити всіх зовнішніх сутностей (користувачів, адміністраторів, зовнішні системи або API), які будуть безпосередньо взаємодіяти із системою.
3. **Визначення варіантів використання:** Для кожного ідентифікованого актора визначити перелік бізнес-завдань (прецедентів), які він може виконувати за допомогою системи.
4. **Побудова діаграми прецедентів (Use Case Diagram):** У вибраному CASE-засобі (наприклад, IBM Rational Software Architect, StarUML, Enterprise Architect або draw.io) побудувати UML-діаграму варіантів використання. Відобразити асоціації між акторами та прецедентами, а також застосувати зв'язки <<include>>, <<extend>> та узагальнення (Generalization) там, де це доцільно.
5. **Специфікація прецедентів:** Розробити детальний текстовий опис (сценарій) для 2–3 ключових варіантів використання системи (основний потік, альтернативні потоки, передумови, постумови).
6. **Оформлення звіту:** Зібрати візуальні моделі, їх опис та специфікації у звіт і підготуватися до захисту.

Теоретичні відомості

Методологія RUP є процесом, керованим варіантами використання (Use-Case Driven). Це означає, що варіанти використання (прецеденти) визначають процес проєктування, реалізації та тестування системи.

Модель варіантів використання (Use Case Model) — це концептуальна модель, яка описує функціональні вимоги до системи з точки зору кінцевого користувача. Вона відповідає на питання *«Що робить система?»*, не заглиблюючись у те, *«Як вона це робить?»*.

Основними елементами моделі є:

- **Актор (Actor):** Будь-яка зовнішня сутність (людина, апаратний пристрій або інша програмна система), яка взаємодіє з нашою системою. Актори діляться на:
  - *Первинних (Primary Actors):* ініціюють виконання прецеденту для досягнення власної мети (наприклад, «Покупець»).
  - *Вторинних (Secondary Actors):* система звертається до них під час виконання прецеденту (наприклад, «Платіжний шлюз» або «Сервіс SMS-розсилки»).
- **Варіант використання (Use Case, Прецедент):** Послідовність дій (транзакцій), які виконує система у відповідь на подію, ініційовану актором. Кожен прецедент повинен приносити актору відчутний результат (бізнес-цінність).

Типи зв'язків на діаграмі варіантів використання:

1. **Асоціація (Association):** Базовий зв'язок між актором і прецедентом, що показує їхню комунікацію. Зображується суцільною лінією.
2. **Включення (<<include>>):** Використовується, коли один прецедент (базовий) обов'язково включає поведінку іншого прецеденту. Застосовується для уникнення дублювання коду/опису (наприклад, прецеденти «Оформити замовлення» та «Переглянути профіль» можуть *включати* прецедент «Авторизуватися»). Зображується пунктирною стрілкою, спрямованою *до* включеного прецеденту.
3. **Розширення (<<extend>>):** Вказує на необов'язкову (опціональну) поведінку, яка додається до базового прецеденту лише за певних умов (наприклад, до базового «Оплатити товар» може застосовуватися розширення «Оплатити бонусними балами», якщо у користувача вони є). Зображується пунктирною стрілкою, спрямованою *від* прецеденту розширення до базового.
4. **Узагальнення (Generalization):** Показує наслідування (спадкування) між акторами (наприклад, актор «Адміністратор» наслідує всі права актора «Авторизований користувач») або між прецедентами.

Специфікація варіанта використання: Сама по собі діаграма є лише візуальним індексом функцій. Основна цінність полягає у **текстових специфікаціях (сценаріях)**. Сценарій описує покрокову взаємодію актора з системою у форматі «Актор робить X — Система відповідає Y».

Додатки до лабораторної роботи №3

Додаток 1. Шаблон специфікації варіанта використання (Use Case Specification)

- **Назва прецеденту:** [Дієслово + іменник, наприклад, "Оформити замовлення"]
- **Унікальний ідентифікатор:** [Наприклад, UC-01]
- **Короткий опис:** [Одне-два речення про суть прецеденту]
- **Основний актор:** [Хто ініціює прецедент]
- **Вторинні актори:** [Інші системи чи ролі]
- **Передумови (Pre-conditions):** [Що має бути істиною до початку виконання, напр., "Користувач авторизований, у кошику є товари"]
- **Постумови (Post-conditions):** [Стан системи після успішного виконання, напр., "Замовлення створено в БД, користувачу надіслано email"]
- **Основний потік (Main Flow):**

1. Актор натискає кнопку "Оформити".
2. Система перевіряє наявність товару на складі.
3. Система розраховує загальну вартість та відображає форму оплати.
4. ...

- **Альтернативні потоки (Alternate Flows / Exceptions):**

1. 2a. Товару немає на складі: Система виводить повідомлення про помилку; прецедент завершується.
2. 4a. Платіж відхилено: Система пропонує ввести дані іншої картки.

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Які основні актори були виявлені в предметній області.
- З якими складнощами довелося зіткнутися під час декомпозиції вимог та виділення базових прецедентів.
- Для яких випадків було доцільно застосувати стереотипи <<include>> та <<extend>>.
- Як розроблена модель прецедентів та текстові сценарії вплинуть на подальше проєктування діаграм класів і діаграм послідовності (Sequence Diagrams).

Список інформаційних джерел

1. Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley Professional.

2. Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Professional.

3. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley Professional.

4. Офіційна специфікація UML (Unified Modeling Language) від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/

5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 4

Тема: Робочий потік «Аналіз та проєктування» (Analysis & Design). Розробка «Моделі аналізу» програмної системи.

Мета роботи: Ознайомитися із завданнями фази «Уточнення» (Elaboration) у RUP. Навчитися переходити від функціональних вимог (сценаріїв варіантів використання) до концептуальної об'єктно-орієнтованої архітектури. Засвоїти принципи побудови діаграм взаємодії (Sequence Diagram) та концептуальних діаграм класів (Class Diagram) у нотації UML.

Завдання до роботи:

1. **Вибір прецедентів:** Обрати 2–3 ключові варіанти використання з текстовими специфікаціями, розробленими в Лабораторній роботі №3.
2. **Розподіл обов'язків (BCE/MVC):** Для кожного обраного прецеденту виділити класи-сутності (Entity), граничні класи (Boundary) та керуючі класи (Control).
3. **Побудова діаграм послідовності (Sequence Diagram):** У вибраному CASE-засобі побудувати діаграми послідовності, що ілюструють взаємодію об'єктів у часі для виконання основного та альтернативного потоків обраних прецедентів. Використати комбіновані фрагменти (alt, opt, loop).
4. **Побудова концептуальної діаграми класів (Class Diagram):** На основі об'єктів, виявлених під час побудови діаграм послідовності, створити концептуальну діаграму класів. Визначити атрибути, методи та вказати типи зв'язків між класами (асоціація, агрегація, композиція, успадкування) з урахуванням множинності.
5. **Оформлення звіту:** Зібрати побудовані моделі та їх описи у звіт і підготуватися до захисту.

Теоретичні відомості

Фаза **Уточнення (Elaboration)** у RUP спрямована на аналіз проблеми, проєктування базової архітектури системи, усунення найвищих технічних ризиків та уточнення плану проєкту. Головним завданням робочого потоку «Аналіз та проєктування» є перетворення вимог (What?) у специфікації системи (How?).

Модель аналізу (Analysis Model) — це перша спроба структурувати систему. Вона є концептуальною (не прив'язаною до конкретної мови програмування чи СКБД) і базується на патерні **BCE (Boundary-Control-Entity)**, який є різновидом архітектурного шаблону MVC (Model-View-Controller):

- **Граничні класи (Boundary / View):** Відповідають за взаємодію системи із зовнішнім світом (акторами). Це можуть бути інтерфейси користувача (наприклад, компоненти React), API-шлюзи або інтерфейси взаємодії з апаратним забезпеченням.

- **Керуючі класи (Control / Controller):** Координують поведінку системи. Вони отримують запити від граничних класів, делегують завдання класам-сутностям та управляють бізнес-логікою сценарію (наприклад, маршрутизатори та контролери у Node.js).
- **Класи-сутності (Entity / Model):** Відображають довгострокову інформацію (дані предметної області) та базову бізнес-логіку (наприклад, об'єкти "Користувач", "Замовлення", "Товар").

Діаграма послідовності (Sequence Diagram) Це діаграма взаємодії, яка показує, як об'єкти обмінюються повідомленнями у часовій послідовності.

- *Життєва лінія (Lifeline):* Вертикальна пунктирна лінія, що позначає час існування об'єкта.
- *Специфікація виконання (Activation Box):* Прямокутник на життєвій лінії, який показує період, коли об'єкт виконує певну дію.
- *Повідомлення (Messages):*
  - *Синхронні (Synchronous)* — суцільна лінія з зафарбованою стрілкою (відправник чекає на відповідь).
  - *Асинхронні (Asynchronous)* — суцільна лінія з відкритою стрілкою (відправник не чекає).
  - *Відповідь (Return)* — пунктирна лінія.
- *Комбіновані фрагменти (Combined Fragments):* Рамки для моделювання складної логіки (alt — if/else, opt — необов'язкове виконання, loop — цикл).

Діаграма класів (Class Diagram) Статична структура системи. Клас зображується як прямокутник, поділений на три секції: Назва, Атрибути (властивості) та Операції (методи). Важливою частиною є **видимість (Visibility)**: + (Public), - (Private), # (Protected). Зв'язки між класами:

- **Асоціація (Association):** Загальний зв'язок між двома незалежними класами (суцільна лінія).
- **Агрегація (Aggregation):** Зв'язок "частина-ціле", де частина може існувати без цілого (лінія з порожнім ромбом на боці цілого). Наприклад, "Відділ" та "Співробітник".
- **Композиція (Composition):** Жорсткий зв'язок "частина-ціле", де частина знищується разом із цілим (лінія із зафарбованим ромбом). Наприклад, "Документ" та "Сторінка документа".
- **Узагальнення (Generalization):** Наслідування (суцільна лінія з порожньою трикутною стрілкою).

Додатки до лабораторної роботи №4

Додаток 1. Рекомендації щодо переходу від Use Case до Моделі аналізу

1. Для кожного прецеденту визначте один **Граничний клас** для комунікації з актором (наприклад, OrderForm).
2. Визначте один **Керуючий клас** для обробки логіки прецеденту (наприклад, OrderController).
3. Проаналізуйте іменники у тексті специфікації прецеденту (наприклад, "кошик", "товар", "рахунок") — вони стануть вашими **Класами-сутностями.**

Додаток 2. Приклад оформлення зв'язків на діаграмі класів

- Вказуйте **множинність (Multiplicity)** на кінцях асоціацій. Наприклад: 1 (рівно один), 0..1 (нуль або один), * (багато), 1..* (один або більше). Це критично важливо для подальшого проєктування реляційної або NoSQL бази даних.

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Які концептуальні класи (Entity, Boundary, Control) були виявлені для обраних прецедентів.
- Як побудова діаграми послідовності допомогла ідентифікувати методи (операції) для класів, які раніше не були очевидними.
- Які типи зв'язків (агрегація чи композиція) домінували у вашій предметній області і чому.
- Наскільки складно було розділити відповідальність (розподіл бізнес-логіки) між керуючими класами та сутностями.

Список інформаційних джерел

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall. *(Базова праця для розуміння переходу від вимог до класів).*
2. Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley Professional.
3. Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.
4. Офіційна специфікація UML (Unified Modeling Language) від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 5

Тема: Деталізація моделі проєктування (Design Model). Розподіл обов'язків за патернами GRASP, застосування принципів SOLID та моделювання станів.

Мета роботи: Навчитися переходити від концептуальної моделі аналізу до деталізованої проєктної моделі (Design Model). Опанувати принципи об'єктно-орієнтованого дизайну (SOLID) та патерни розподілу обов'язків (GRASP) для створення гнучкої, модульної та підтримуваної архітектури програмного забезпечення. Засвоїти побудову діаграм станів (State Machine Diagram) в UML.

Завдання до роботи:

1. **Трансформація діаграми класів:** Виконати рефакторинг концептуальної діаграми класів (створеної в Лабораторній роботі №4) у проєктну діаграму (Design Class Diagram). Додати конкретні типи даних для атрибутів, повні сигнатури методів та типи видимості з урахуванням обраного стеку технологій (наприклад, TypeScript, C#, Java).
2. **Розподіл обов'язків (GRASP):** Проаналізувати створену модель на відповідність базовим патернам GRASP. Обґрунтувати у звіті, який клас виступає «Творцем» (Creator) для ключових сутностей та який є «Інформаційним експертом» (Information Expert).
3. **Застосування SOLID:** Перевірити архітектуру на відповідність принципам SOLID. Продемонструвати на прикладі вашої діаграми, як було забезпечено дотримання Принципу єдиної відповідальності (SRP) та Принципу відкритості/закритості (OCP) (наприклад, через використання інтерфейсів або абстрактних класів).
4. **Побудова діаграми станів (State Machine Diagram):** Обрати один складний об'єкт системи, що має яскраво виражений життєвий цикл (наприклад, об'єкт «Замовлення», «Рахунок», «Користувач» або «Публікація»). Побудувати для нього діаграму станів, вказавши початковий/кінцевий стани, переходи, тригери та охоронні умови (Guards).
5. Розробка моделі: - Створення моделі розгортання з використанням діаграм або потокових схем. - Включення основних етапів розгортання програмного забезпечення у модель.
6. Аналіз результатів розгортання:- Оцінка ефективності та правильності виконання робочого потоку.
7. Виявлення можливих покращень або оптимізацій процесу.
8. **Оформлення звіту:** Зібрати оновлені діаграми класів, діаграму станів та текстові обґрунтування застосування патернів у звіт і підготуватися до захисту.

Теоретичні відомості

Якщо модель аналізу (Analysis Model) відповідала на запитання «Що система повинна робити?», то **модель проєктування (Design Model)** відповідає на запитання «Як система буде це реалізовувати технічно?». Проєктні класи стають прямими прототипами для написання програмного коду.

GRASP (General Responsibility Assignment Software Patterns) — це набір базових шаблонів, що допомагають правильно розподілити відповідальність між класами. Основні патерни:

- **Information Expert (Інформаційний експерт):** Відповідальність за виконання дії призначається тому класу, який володіє максимумом інформації, необхідної для її виконання.
- **Creator (Творець):** Клас A повинен створювати екземпляри класу B, якщо A містить або агрегує B, активно використовує B, або володіє даними для ініціалізації B.
- **Controller (Контролер):** Відповідальність за обробку вхідних системних подій делегується спеціальному керуючому класу, а не UI-компонентам (збігається з Control у моделі BCE).
- **Low Coupling (Низька зв'язність):** Розподіляти обов'язки так, щоб залежність між класами залишалася мінімальною (зменшує вплив змін в одному класі на інші).
- **High Cohesion (Високе зчеплення):** Обов'язки класу повинні бути сильно пов'язані між собою і сфокусовані на одній меті.

SOLID — це п'ять принципів об'єктно-орієнтованого проєктування, що базуються на GRASP, але фокусуються на архітектурній гнучкості:

- **S - Single Responsibility Principle (SRP):** Клас повинен мати лише одну причину для зміни (вирішувати лише одну задачу).
- **O - Open/Closed Principle (OCP):** Програмні сутності повинні бути відкриті для розширення, але закриті для модифікації (досягається через поліморфізм та інтерфейси).
- **L - Liskov Substitution Principle (LSP):** Об'єкти у програмі можна замінити їх спадкоємцями без зміни властивостей програми.
- **I - Interface Segregation Principle (ISP):** Багато спеціалізованих інтерфейсів краще, ніж один універсальний.
- **D - Dependency Inversion Principle (DIP):** Залежності всередині системи мають будуватися на основі абстракцій, а не конкретних реалізацій.

Діаграма станів (State Machine Diagram) В UML використовується для моделювання динамічної поведінки об'єкта протягом його життєвого циклу. Основні елементи:

- **Стан (State):** Поточна умова існування об'єкта (наприклад, Нове, Оплачене, Відправлене).
- **Перехід (Transition):** Лінія зі стрілкою, що з'єднує два стани. Показує зміну стану.
- **Тригер (Trigger) / Подія:** Те, що викликає перехід (наприклад, виклик методу pay()).
- **Охоронна умова (Guard):** Логічна умова в квадратних дужках [умова], яка має бути істинною (True), щоб перехід відбувся (наприклад, [баланс >= сума]).

Додатки до лабораторної роботи№5

Додаток 1. Чекліст трансформації класу в Design Class

- Чи всі атрибути мають вказаний тип даних (наприклад, email: String, price: Decimal)?
- Чи визначено видимість для кожного атрибута (переважно - Private) та методу (+ Public, # Protected)?
- Чи додано гетери/сетери (якщо це вимагає обрана мова програмування)?
- Чи вказано типи значень, що повертаються методами (наприклад, calculateTotal(): Decimal)?
- Чи виокремлено абстрактні класи або інтерфейси (відображаються зі стереотипом <<interface>>)?

Додаток 2. Приклад нотації переходу на діаграмі станів

- Формат підпису стрілки переходу: Подія [Охоронна умова] / Дія
- *Приклад:* скасуватиЗамовлення() [статус == "Оплачено"] / повернутиКошти()

Додаткові вказівки:

- Робочий потік повинен бути систематичним та логічним, з урахуванням послідовності етапів розробки програмного забезпечення.
- Використовуйте відповідні інструменти для моделювання, реалізації та тестування програмного забезпечення.

- Зверніть увагу на документування кожного етапу робочого потоку та збереження коду та інших матеріалів у відповідних репозиторіях або системах контролю версій.

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Яким чином змінилася діаграма класів після переходу від етапу аналізу до етапу проєктування.
- Наскільки ефективно вдалося застосувати патерни *Information Expert* та *Creator* у вашій моделі.
- Як застосування принципів SOLID (зокрема, SRP та OCP) вплине на подальшу здатність коду до розширення, рефакторингу та тестування.
- Для якого об'єкта було розроблено діаграму станів і чому саме його життєвий цикл потребував окремого моделювання.

Список інформаційних джерел

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall. *(Головне джерело для вивчення GRASP).*
2. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. *(Базове джерело для принципів SOLID).*
3. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
4. Офіційна специфікація UML (Unified Modeling Language) від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 6

Тема: Проєктування архітектури програмної системи. Модель «4+1», компонентне моделювання та критерії хорошої архітектури.

Мета роботи: Ознайомитися з основними архітектурними стилями та шаблонами. Засвоїти підхід до опису архітектури за моделлю «4+1» (Philippe Kruchten). Навчитися декомпозувати програмну систему на логічні підсистеми/модулі та будувати UML-діаграму компонентів (Component Diagram), визначаючи інтерфейси та залежності між ними.

Завдання до роботи:

1. **Вибір архітектурного стилю:** Визначити та обґрунтувати базовий архітектурний стиль для вашого проєкту (наприклад, багатошарова архітектура (Layered), мікросервісна (Microservices), клієнт-серверна або чиста архітектура (Clean Architecture)).
2. **Опис за моделлю «4+1»:** Сформувати текстовий опис архітектури вашої системи, розкривши її через 4 базові представлення (Логічне, Реалізації, Процесів, Розгортання) та 1 об'єднуюче (Сценарії).
3. **Декомпозиція на компоненти:** Розбити систему на великі функціональні блоки (компоненти). Визначити, які компоненти відповідають за інтерфейс користувача (Frontend), які — за бізнес-логіку (Backend API, сервіси), а які — за роботу з даними.
4. **Побудова діаграми компонентів (Component Diagram):** У вибраному CASE-засобі побудувати діаграму, що відображає компоненти системи. Обов'язково вказати **надані (Provided)** та **необхідні (Required)** інтерфейси, а також залежності між компонентами.
5. **Оцінка якості архітектури:** Проаналізувати створену архітектуру за основними критеріями якості (масштабованість, надійність, безпека тощо).
6. **Оформлення звіту:** Зібрати архітектурний опис, візуальні моделі та обґрунтування у звіт і підготуватися до захисту.

Теоретичні відомості

Архітектура програмного забезпечення — це фундаментальна організація системи, втілена в її компонентах, їх відношеннях між собою та середовищем, а також принципи, що керують її проєктуванням та еволюцією. У методології RUP архітектура відіграє центральну роль (процес є Architecture-Centric).

Модель архітектурних представлень «4+1» (Kruchten's 4+1 View Model): Оскільки єдиної діаграми недостатньо для опису всієї системи, розробник RUP Філіп Крачтен запропонував описувати архітектуру з п'яти різних точок зору, що задовольняють різних стейкхолдерів:

1. **Логічне представлення (Logical View):** Описує об'єктну модель дизайну (діаграми класів, пакетів). Цільова аудиторія — розробники.
2. **Представлення реалізації / розробки (Development / Implementation View):** Описує статичну організацію програмних модулів, бібліотек та компонентів у середовищі розробки (діаграма компонентів). Цільова аудиторія — програмісти, менеджери.
3. **Представлення процесів (Process View):** Описує паралелізм, потоки, синхронізацію та продуктивність (діаграми діяльності, комунікації). Цільова аудиторія — системні інтегратори.
4. **Представлення фізичного розгортання (Physical / Deployment View):** Показує, як програмне забезпечення встановлюється на апаратне забезпечення (діаграма розгортання). Цільова аудиторія — DevOps, системні адміністратори.
5. **Сценарії (+1) (Use Case View):** Варіанти використання (прецеденти), які об'єднують і тестують усі попередні 4 представлення.

Рисунок 2. Модульна архітектура. Декомпозиція як основа

Примітка: Сучасною альтернативою або доповненням до моделі «4+1» є **модель C4** (Context, Containers, Components, Code), яка також базується на ідеї ієрархічної декомпозиції архітектури.

Діаграма компонентів (Component Diagram) Відображає розбиття програмної системи на структурні компоненти та показує зв'язки між ними.

- **Компонент:** Зображується як прямокутник із символом компонента (або стереотипом <<component>>). Це інкапсульована частина системи (наприклад, PaymentService.dll, AuthModule, Database).
- **Наданий інтерфейс (Provided Interface):** Показує, які послуги компонент *надає* іншим (зображується як лінія з повним колом на кінці — "льодяник" / lollypop).
- **Необхідний інтерфейс (Required Interface):** Показує, які послуги компонент *вимагає* від інших для своєї роботи (зображується як лінія з півколом — "гніздо" / socket).
- **Залежність:** Показується пунктирною стрілкою від компонента, що вимагає інтерфейс, до компонента, що його надає.

Критерії хорошої архітектури:

- *Гнучкість (Maintainability):* Легкість внесення змін без руйнування існуючого функціоналу (досягається через слабку зв'язність — Low Coupling).
- *Масштабованість (Scalability):* Здатність системи справлятися зі збільшенням навантаження (наприклад, можливість додати нові екземпляри сервера).
- *Надійність (Reliability) та Відмовостійкість (Fault Tolerance):* Здатність працювати при збоях окремих компонентів.
- *Безпека (Security):* Захист даних та контроль доступу на рівні архітектурних меж.

Додатки до лабораторної роботи №6

Додаток 1. Шаблон опису архітектури (Architecture Document Extract)

- **1. Обраний архітектурний стиль:** [Наприклад, Клієнт-Серверна архітектура з REST API та монолітною базою даних].
- **2. Обґрунтування вибору:** [Чому цей стиль найкраще підходить для вирішення вашої бізнес-проблеми].
- **3. Представлення моделі «4+1» (короткий опис):**
  - *Логічне:* Основна бізнес-логіка зосереджена у модулях [Назви модулів].
  - *Розробки:* Система розбита на 3 основні репозиторії (Frontend-додаток, Backend-сервер, Бібліотека утиліт).
  - *Процесів:* Користувацькі запити обробляються асинхронно за допомогою [назва технології/брокера повідомлень].
  - *Розгортання:* Клієнтська частина завантажується в браузер, серверна розгортається у Docker-контейнерах на хмарному хостингу.

Додаток 2. Чекліст побудови Діаграми компонентів

- Чи виділені окремі компоненти для UI, бізнес-логіки та доступу до даних?
- Чи всі взаємодії між компонентами відбуваються **виключно** через інтерфейси (а не через прямий доступ до внутрішніх класів)?
- Чи чітко видно, який компонент *реалізує* інтерфейс (lollypop), а який від нього *залежить* (socket)?

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Який архітектурний стиль було обрано та які його головні переваги для конкретної предметної області.
- Які основні компоненти системи було ідентифіковано і як між ними розподілено обов'язки.
- Завдяки яким рішенням на діаграмі компонентів було досягнуто слабкої зв'язності (Low Coupling) між підсистемами.
- Які критерії "хорошої архітектури" були найбільш критичними для вашого проєкту (наприклад, безпека для фінансової системи чи продуктивність для медіа-порталу).

Список інформаційних джерел

1. Kruchten, P. (1995). *Architectural Blueprints—The "4+1" View Model of Software Architecture*. IEEE Software, 12(6), 42-50.
2. Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media.
3. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
4. Офіційна специфікація UML (Unified Modeling Language) від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 7

Тема: Робочі потоки «Реалізація» (Implementation) та «Впровадження» (Deployment). Проєктування фізичної архітектури та інтеграція DevOps-практик.

Мета роботи: Ознайомитися з фазами «Побудова» (Construction) та «Впровадження» (Transition) методології RUP у контексті сучасних підходів до розробки. Навчитися проєктувати фізичну архітектуру системи за допомогою діаграм розгортання (Deployment Diagram). Засвоїти базові принципи DevOps, контейнеризації та налаштування конвеєрів безперервної інтеграції і доставки (CI/CD).

Завдання до роботи:

1. **Визначення топології інфраструктури:** Визначити апаратні вузли, сервери, хмарні платформи та середовища виконання, необхідні для роботи розробленої у попередніх роботах системи.
2. **Побудова діаграми розгортання (Deployment Diagram):** У вибраному CASE-засобі або інструменті (draw.io, Lucidchart, PlantUML) побудувати діаграму, що відображає фізичне розміщення програмних артефактів (компонентів) на обчислювальних вузлах та протоколи зв'язку між ними.
3. **Проєктування контейнеризації:** Розробити базовий конфігураційний файл (наприклад, Dockerfile або docker-compose.yml) для пакування хоча б одного компонента вашої системи (наприклад, backend-сервера або бази даних) у контейнер.
4. **Проєктування CI/CD конвеєра:** Створити логічну схему або базовий конфігураційний файл конвеєра безперервної інтеграції та розгортання (наприклад, для GitHub Actions або GitLab CI), який включає етапи: збірка (Build), тестування (Test) та розгортання (Deploy).
5. **Оформлення звіту:** Зібрати діаграму розгортання, конфігураційні файли (або їх проєктні схеми) та обґрунтування інфраструктурних рішень у звіт і підготуватися до захисту.

Теоретичні відомості

Робочі потоки **«Реалізація» (Implementation)** та **«Впровадження» (Deployment)** є ключовими на пізніх фазах життєвого циклу RUP. Їхня мета — перетворити проєктні моделі у виконуваний код, інтегрувати компоненти в єдину систему та доставити готовий продукт кінцевим користувачам. У сучасній програмній інженерії ці процеси тісно пов'язані з культурою DevOps.

Діаграма розгортання (Deployment Diagram) В UML ця діаграма використовується для моделювання фізичної архітектури системи. Вона показує топологію апаратного забезпечення та розміщення на ньому програмних компонентів. Основні елементи:

- **Вузол (Node):** Фізичний або віртуальний елемент, що володіє обчислювальними ресурсами (зображується як тривимірний куб). Вузли поділяються на:
  - *Пристрої (Device):* Апаратні засоби (наприклад, сервер бази даних, смартфон клієнта, балансувальник навантаження).
  - *Середовища виконання (Execution Environment):* Програмні платформи, що працюють на пристроях (наприклад, ОС Linux, Node.js Runtime, Docker Container, веббраузер).
- **Артефакт (Artifact):** Фізичний файл, який розгортається на вузлі (наприклад, .jar, .dll, .exe, скомпільований бандл React-додатку).
- **Шлях зв'язку (Communication Path):** Лінія між вузлами, яка показує, що вони обмінюються даними. Зазвичай надписується мережевим протоколом (наприклад, <<HTTP/HTTPS>>, <<TCP/IP>>, <<WebSocket>>).

Основи DevOps та розгортання:

- **Контейнеризація (Docker):** Технологія пакування програмного забезпечення та всіх його залежностей у стандартизовану одиницю (контейнер) для уніфікованого запуску у будь-якому середовищі.
- **Безперервна інтеграція (CI - Continuous Integration):** Практика частого (щоденного) злиття коду в спільний репозиторій з автоматичною збіркою та запуском модульних тестів.
- **Безперервне розгортання/доставка (CD - Continuous Deployment/Delivery):** Автоматизований процес доставки перевіреного коду у середовища тестування (Staging) або на робочі сервери (Production).
- **Інфраструктура як код (IaC):** Управління конфігурацією серверів та мереж за допомогою машиночитних файлів.

Додатки до лабораторної роботи №7

Додаток 1. Чекліст побудови Діаграми розгортання

- Чи відображено клієнтський рівень (браузер або мобільний пристрій)?
- Чи відображено серверний рівень (Application Server) та рівень баз даних (Database Server)?

- Чи вказані артефакти всередині вузлів виконання?
- Чи позначені мережеві протоколи взаємодії між пристроями?

Додаток 2. Приклад структури базового Dockerfile для Node.js сервісу

Dockerfile
```
# Вибір базового образу
FROM node:18-alpine
# Встановлення робочої директорії
WORKDIR /usr/src/app
# Копіювання файлів залежностей
COPY package*.json ./
# Встановлення залежностей
RUN npm install
# Копіювання вихідного коду
COPY . .
# Відкриття порту
EXPOSE 3000
# Команда запуску
CMD ["npm", "start"]
```

Додаток 3. Шаблон етапів CI/CD конвеєра (YAML-стиль)

- **Етап 1: Build (Збірка).** Завантаження коду з Git, компіляція або встановлення залежностей (напр., npm install).
- **Етап 2: Test (Тестування).** Запуск автоматизованих модульних тестів та статичного аналізатора коду (напр., ESLint, SonarQube).
- **Етап 3: Package (Пакування).** Створення Docker-образу та його відправка у реєстр (Container Registry).
- **Етап 4: Deploy (Розгортання).** Оновлення контейнерів на цільовому сервері.

DevOps-технології. Залежно від потреб та специфіки проекту, інтегрувати та описати елементи проєкту:

1) Construction – Впровадження Continuous Integration (CI)
Мета: Автоматизувати перевірку якості коду та збірку артефактів.

- **Завдання:**
    1. Налаштувати **CI-конвеєр** (GitHub Actions, GitLab CI або Jenkins).
    2. Етапи конвеєра:
        - Linting (перевірка стилю коду).

        - Unit-тестування (з генерацією звіту про покриття — Code Coverage).
        - Збірка Docker-образу.
        - Сканування образу на вразливості (Trivy, Snyk).
    3. Налаштувати автоматичну публікацію образу в Docker Hub або GitHub Packages.

2) Infrastructure as Code (IaC) та Configuration Management

Мета: Навчитися керувати інфраструктурою як кодом, що є критичним для DevOps.

- **Завдання:**
    1. Описати необхідну інфраструктуру (сервери, мережі) за допомогою **Terraform** (або OpenTofu).
    2. (Альтернатива) Використати **Ansible** для конфігурування цільового сервера (встановлення Docker, налаштування Nginx/Reverse Proxy).
    3. Автоматизувати підняття оточення (Staging/Production).

3) Transition – Continuous Delivery & Deployment (CD)

Мета: Реалізувати автоматичне розгортання застосунку в хмарне середовище або на віддалений сервер.

- **Завдання:**
    1. Налаштувати CD-пайплайн для деплою артефакту на сервер після успішного CI.
    2. Реалізувати одну зі стратегій розгортання: **Blue-Green Deployment** або **Canary Release**.
    3. Налаштувати автоматичне оновлення бази даних (міграції) в межах деплою.

4) Замкнути життєвий цикл ПЗ через збір метрик та зворотний зв'язок.

Завдання:

1. Інтегрувати систему збору логів (ELK Stack або Grafana Loki).
2. Налаштувати моніторинг стану системи (Prometheus + Grafana).
3. Створити дашборд, що відображає Health Check сервісів та використання ресурсів (CPU/RAM).
4. Налаштувати Alerting (сповіщення в Telegram/Slack при падінні сервісу).

    Рекомендований технологічний стек:
    - **Управління:** Jira / GitHub Projects.
    - **Контейнеризація:** Docker, Docker Compose.
    - **CI/CD:** GitHub Actions / GitLab CI.

- **IaC:** Terraform / Ansible.
- **Хмара:** AWS (Free Tier) / DigitalOcean / локальні VM (Proxmox/VirtualBox).
- **Моніторинг:** Prometheus, Grafana.

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Які основні апаратні вузли та програмні середовища виконання необхідні для розгортання розробленої системи.
- Яким чином архітектура розгортання впливає на масштабованість та відмовостійкість продукту.
- У чому полягає практична цінність застосування контейнеризації (Docker) порівняно з класичним розгортанням безпосередньо на ОС сервера.
- Як впровадження CI/CD пайплайну знижує ризики на фазі Впровадження (Transition).

Список інформаційних джерел

1. Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. (2021). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* (2nd ed.). IT Revolution Press.
2. Farley, D. (2021). *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley.
3. Офіційна специфікація UML (Unified Modeling Language) від OMG. [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/
4. Офіційна документація Docker. [Електронний ресурс] — Режим доступу: https://docs.docker.com/
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ЛАБОРАТОРНА РОБОТА 8

Тема: Управління ризиками проєкту та валідація вимог. Робочі потоки «Управління проєктом» (Project Management) та «Тестування» (Test).

Мета роботи: Опанувати методи ідентифікації, оцінки та мінімізації проєктних і технічних ризиків, що є основоположним принципом методології RUP. Навчитися проводити верифікацію та валідацію розробленого програмного забезпечення, створювати матрицю трасування вимог та планувати приймальні випробування (Acceptance Testing).

Завдання до роботи:

1. **Ідентифікація ризиків:** Для вашої предметної області (наскрізного проєкту) виявити та описати мінімум 5 потенційних ризиків (технічних, організаційних, ресурсних або бізнес-ризиків).
2. **Оцінка ризиків:** Побудувати матрицю оцінки ризиків (Probability and Impact Matrix), оцінивши кожен виявлений ризик за ймовірністю виникнення та рівнем впливу на проєкт.
3. **Розробка стратегії реагування:** Для 2-3 найбільш критичних ризиків розробити план заходів щодо їх пом'якшення (Mitigation) або уникнення (Avoidance).
4. **Трасування вимог:** Побудувати Матрицю трасування вимог (Requirements Traceability Matrix - RTM), яка пов'язує початкові бізнес-вимоги (з Лаб. №2) з прецедентами (Лаб. №3) та архітектурними компонентами (Лаб. №6).
5. **Планування валідації:** Розробити базовий план приймальних випробувань (Acceptance Test Plan) для одного ключового варіанта використання, щоб довести, що система відповідає очікуванням стейкхолдерів.
6. **Оформлення звіту:** Зібрати Реєстр ризиків, Матрицю трасування та План тестування у звіт і підготуватися до фінального захисту проєкту.

Теоретичні відомості

Управління ризиками (Risk Management)

Однією з головних відмінностей RUP від каскадної моделі (Waterfall) є те, що RUP — це процес, керований ризиками. Ітеративний підхід дозволяє виявляти та усувати найвищі технічні ризики ще на фазі «Уточнення» (Elaboration), до того як буде написано основний масив коду.

- **Ризик** — це невизначена подія або умова, яка в разі виникнення має позитивний чи негативний вплив на цілі проєкту (строки, вартість, якість).

• *Оцінка ризику* зазвичай виконується за двома параметрами: **Ймовірність (Probability)** (від 1 до 5) та **Вплив (Impact)** (від 1 до 5). Їхній добуток дає рівень критичності ризику.

• *Базові стратегії реагування на негативні ризики:*

- **Уникнення (Avoid):** Зміна плану проєкту для повного виключення загрози (наприклад, відмова від використання нестабільної бібліотеки).
- **Передача (Transfer):** Перенесення наслідків на третю сторону (наприклад, купівля страховки або перехід на керований хмарний сервіс AWS замість власного сервера).
- **Пом'якшення (Mitigate):** Зниження ймовірності або наслідків (наприклад, регулярне резервне копіювання, код-рев'ю).
- **Прийняття (Accept):** Усвідомлене рішення нічого не змінювати, але мати резервний фонд часу/грошей.

Верифікація та Валідація (V&V)

• **Верифікація (Verification):** Чи правильно ми створюємо продукт? (Перевірка того, що програма відповідає технічній специфікації, архітектурі та стандартам кодування — покривається Unit/Integration тестами).

• **Валідація (Validation):** Чи правильний продукт ми створюємо? (Перевірка того, що готова система задовольняє реальні потреби стейкхолдерів і відповідає документу Vision на фазі «Впровадження» — покривається Acceptance тестами).

Матриця трасування вимог (Requirements Traceability Matrix - RTM)

Це інструмент контролю якості, який гарантує, що жодна вимога замовника не була втрачена під час проєктування, і жоден рядок коду не був написаний для функціоналу, якого не вимагали. Вона у вигляді таблиці пов'язує унікальний ідентифікатор вимоги з відповідним Use Case, компонентом системи та тест-кейсом.

Додатки до лабораторної 8.

Додаток 1. Шаблон Реєстру ризиків (Risk Register)

ID	**Назва та опис ризику**	**Категорія**	**Ймовірність (1-5)**	**Вплив (1-5)**	**Критичність (Й х В)**	**Стратегія**	**Заплановані дії (Mitigation)**
R1	Збій інтеграції	Технічний	3	5	15 (Висока)	Пом'якшення	Розробка заглушки

ID	Назва та опис ризику	Категорія	Ймовірність (1-5)	Вплив (1-5)	Критичність (Й х В)	Стратегія	Заплановані дії (Mitigation)
	зі стороннім API банку						(Mock) для тестування; реалізація патерну Circuit Breaker.
R2	Звільнення ключового розробника	Ресурсний	2	4	8 (Середня)	Пом'якшення	Обов'язкове ведення документації (Wiki); регулярні Knowledge Sharing сесії.

Додаток 2. Шаблон Матриці трасування вимог (RTM)

ID Вимоги (з Лаб.2)	Опис вимоги	ID Прецеденту (з Лаб.3)	Компонент архітектури (з Лаб.6)	ID Тест-кейсу	Статус
REQ-F-01	Система повинна дозволяти оплату карткою	UC-05 (Оплатити замовлення)	PaymentService, OrderController	TC-01-Pay	Реалізовано
REQ-P-02	Час завантаження сторінки < 2 сек	N/A (Нефункціональна)	FrontendApp, CDN Node	TC-02-Perf	Тестується

Додаток 3. Шаблон сценарію приймального тестування (UAT - User Acceptance Test)

- **Назва тесту:** [Перевірка успішного оформлення замовлення]
- **Пов'язана вимога:** [REQ-F-01]
- **Опис кроків (Action):** [1. Авторизуватися; 2. Додати товар; 3. Натиснути "Оплатити"...]

- **Очікуваний результат (Expected Result):** [Гроші списано, статус замовлення змінено на "Оплачено", надіслано email].
- **Фактичний результат (Actual Result):** [Заповнюється після тестування — Pass/Fail].

Вимоги до висновків

У висновку до лабораторної роботи студент повинен зазначити:

- Які типи ризиків виявилися найбільш критичними для обраної предметної області.
- Як ітеративна модель розробки (RUP / Agile) допомагає мінімізувати виявлені технічні ризики порівняно з водоспадною (Waterfall) моделлю.
- Яким чином матриця трасування (RTM) допомагає при внесенні змін до вимог замовника (Change Management).
- Чи всі початкові бізнес-вимоги (з документа Vision) були покриті фінальною архітектурою та тестами.

Список інформаційних джерел

1. Project Management Institute (PMI) (2021). *A Guide to the Project Management Body of Knowledge (PMBOK Guide)* (7th ed.). Project Management Institute. *(Головний стандарт з управління ризиками та проєктом).*
2. Sommerville, I. (2020). *Software Engineering* (11th ed.). Pearson. *(Розділи про верифікацію, валідацію та тестування).*
3. Wiegers, K., & Beatty, J. (2023). *Software Requirements* (4th ed.). Microsoft Press. *(Розділ про матрицю трасування вимог).*
4. Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.
5. Навчально-методичні матеріали та конспекти лекцій з курсу кафедри програмної інженерії ТНТУ ім. І. Пулюя.

ВИСНОВКИ

Протягом перших років роботи в IT-індустрії розробники зазвичай фокусуються на написанні програмного коду та реалізації локальних рішень для конкретних задач. Однак із поглибленням професійного досвіду стає очевидним, що більшість критичних помилок і вузьких місць набагато ефективніше усувати ще на етапі аналізу та проєктування програмного забезпечення, за допомогою вдалого архітектурного рішення.

Патерни та архітектурні стилі — це перевірені часом типові плани, які забезпечують елегантне вирішення проблем, з якими ви будете стикатися знову і знову у своїй кар’єрі інженера. Проте успішне застосування патернів неможливе без системного підходу до розробки, який і забезпечує методологія ітеративного проєктування.

Протягом виконання лабораторного практикуму з дисципліни ви пройшли весь шлях створення програмного продукту — від початкового етапу (Inception) до фази впровадження (Transition). Ви навчилися:

- проводити глибокий аналіз предметної області, ідентифікувати стейкхолдерів та грамотно управляти функціональними й нефункціональними вимогами;
- будувати візуальні моделі за допомогою мови UML для документування концептуальних варіантів використання, структури класів та динаміки взаємодії об'єктів;
- правильно розподіляти обов'язки між об'єктами (застосовуючи принципи SOLID та патерни GRASP) для забезпечення низької зв'язності та високого зчеплення архітектури;
- використовувати сучасні CASE-засоби та середовища візуального моделювання (наприклад, IBM Rational Software Architect та аналоги) для проєктування та автоматизованої генерації базового каркаса коду;
- проєктувати фізичну архітектуру та готувати систему до розгортання, розуміючи роль контейнеризації та сучасних DevOps-практик;
- ідентифікувати технічні ризики, будувати матриці трасування та планувати стратегію приймального тестування для валідації кінцевого продукту.

Комплексне бачення життєвого циклу розробки та набуті практичні навички об'єктно-орієнтованого проєктування є міцним фундаментом для подальшого професійного зростання. Вони дозволять вам еволюціонувати від позиції рядового розробника (програміста) до рівня Software Architect, здатного проєктувати гнучкі, масштабовані та надійні корпоративні системи, що приносять реальну цінність бізнесу.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

Базова література:

1. Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional.
2. Sommerville, I. (2020). *Software Engineering* (11th ed.). Pearson.
3. Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
4. Wiegers, K., & Beatty, J. (2023). *Software Requirements* (4th ed.). Microsoft Press.
5. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.

Допоміжна література:

6. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
7. Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media.
8. Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.
9. Brown, S. (2020). *Software Architecture for Developers: Visualise, document and explore your software architecture*. Leanpub.
10. Vernon, V. (2016). *Domain-Driven Design Distilled*. Addison-Wesley Professional.
11. Rubin, K. S. (2022). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley.
12. Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. (2021). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations* (2nd ed.). IT Revolution Press.
13. Farley, D. (2021). *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley.

Інформаційні ресурси в мережі Інтернет:

14. Офіційна специфікація UML (Unified Modeling Language) версії 2.5.1 від OMG (Object Management Group). [Електронний ресурс] — Режим доступу: https://www.omg.org/spec/UML/2.5.1/PDF
15. Офіційний портал методології C4 (The C4 model for visualising software architecture). [Електронний ресурс] — Режим доступу: https://c4model.com/
16. ISO/IEC/IEEE 29148:2018. Systems and software engineering — Life cycle processes — Requirements engineering. [Електронний ресурс] — Режим доступу: https://www.iso.org/standard/72089.html
17. Офіційна документація та керівництва з використання сучасних екосистем розробки (наприклад, React, Node.js), систем управління базами даних та хмарних платформ (залежно від обраного технологічного стека у проєкті).

Додаток А. Приклад оформлення звіту лабораторної роботи

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет
імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем
та програмної інженерії

Кафедра програмної інженерії

ЗВІТ

про виконання лабораторні роботи № ___

з дисципліни

«Раціональний уніфікований процес проєктування програмного забезпечення»

Тема: _____

Виконав(ла): _____

_____

____

Перевірив(ла): _____

____

Тернопіль — 2026

Рекомендована структура звіту
Звіт повинен містити такі основні розділи:

1. **Вступ**
   - Мета виконання лабораторної роботи
   - Актуальність проблеми, що розглядається
   - Задачі роботи (перелік із нумерацією)

2. **Теоретичні основи**
   - Короткий виклад теоретичних концепцій програмної інженерії, необхі- дних для розв'язання задачі
   - Основні визначення та принципи
   - Огляд використаних методів та підходів програмної інженерії

3. **Практична реалізація**
   - Описання алгоритму розв'язання
   - Вихідний код (або посилання на Jupyter notebook в репозиторії)
   - Технічні особливості реалізації
   - Використані бібліотеки та інструменти
   - Архітектурні рішення (за потреби)

4. **Результати та аналіз**
   - Результати експериментів або тестування
   - Візуалізація (графіки, таблиці, діаграми)
   - Інтерпретація результатів
   - Порівняння з очікуваними результатами
   - Аналіз якості програмного продукту (за потреби)

5. **Висновки**
   - Підсумування основних результатів лабораторної роботи
   - Відповідь на поставлені в роботі питання та задачі
   - Рекомендації для подальшого розвитку або удосконалення
   - Аналіз труднощів, з якими довелося мати справу під час виконання
   - Оцінка ефективності застосованих методів програмної інженерії

6. **Список посилань**

   - Список всіх використаних джерел літератури (рекомендується використовувати стиль цитування IEEE. Включити посилання на використані бібліотеки, фреймворки та інстру- менти. Посилання на стандарти та методології програмної інженерії

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя

кафедра програмної інженерії

ЗВІТ
До лабораторної роботи №_

«Побудова архітектури ПЗ в середовищі IBM RSA на основі архітектурної моделі проектування»

з дисципліни:

«Раціональний уніфікований процес проєктування програмного забезпечення»

Виконав: ст. Гр. СПм-51

Перевірив: асист.

Тернопіль 2026

Тема: Побудова архітектури ПЗ в середовищі IBM RSA на основі архітектурної моделі проектування «Абстрактна фабрика/Abstract Factory»

Мета роботи: реалізувати в системі IBM RSA архітектурну модель (АМ) проектування ПЗ - Аабстрактна фабрика (AF) з використаннм інструментальних засобів розробки ПЗ IBM Rational Software Architect.

Загальний опис архітектурної моделі ПЗ «Абстрактна фабрика».

Абстрактна фабрика - гнучка і динамічна архітектурна модель проектування моделі предметної області системи (**МПО**), що включає взаємозв'язані ієрархії на основі інферфейсів (абстрактних класів) для створення множини взаємозв'язаних об'єктів, не специфікуючи їх конкретних класів. Спочатку проектуються абстрактні класи (базові класи ієрархій), що описують інтерфейси для створення компонентів системи, а потім створюються похідні конкретні класи, що поліморфно реалізують ці інтерфейси.

АМ AF. Умови застосування :

¬ незалежність системи від способів утворення, компонування і представлення вхідних до неї об'єктів;

¬ класи вхідних колекцій об'єктів взаємозв'язані і мають використовуватися разом;

¬ конфігурування системи ч/з визначену структуру абстрактних класів, що розкривають тільки **інтерфейси** колекції об'єктів, а не **реалізацію**.

Переваги АМ AF :

¬ внесення швидких змін у місця коду ієрархій, строго визначені архітектором (підсистеми/ класи, їх зміна, розширення їх переліку), не порушуючи цілісності всієї системи;

¬ суттєве зменшення кількості класів та їх описів, що веде до оптимізації коду, підвищення рівня читабельності щодо доопрацювання (відхід від важкої спадщини попередників);

¬ забезпечення формалізованих уніфікованих запитів користувача до системи ч/з інтерфейси (відокремлення описів класів від реалізацій, інкапсуляція і захист від несанкціонованого доступу);

¬ гнучкість і незалежність розроблюваних компонент ПЗ, їх взаємозамінність, повторне використання коду .

Табл. 1 **Класи архітектурної моделі та їх відповідальність**

класи моделі AF	Відповідальність, дії
AbstractFactory (**AF**) (абстрактна фабрика)	оголошує інтерфейс для операцій, що створюють **AF**
ConcreteFactory (**CF**) (конкретна фабрика)	реалізує операції, що створюють конкретні фабрики
AbstractProduct (**AP**) (абстрактний продукт)	оголошує інтерфейс для класів **AP**

ConcreteProduct (**CP**) (конкретний продукт)	визначає конкретний об'єкт- **CP**, що створюється відповідною **CF**, реалізує інтерфейс **AP**
Client (клієнт, користувач)	формує запити, користується виключно інтерфейсами, оголошеними у класах AbstractFactory та AbstractProduct

Відношення між **класами** і **об'єктами**: **AF** передоручає створення об'єктів **CP** своєму ПК **CF**. Екземпляр класу **CF** створює **CP**-об'єкти з визначеною реалізацією. При створенні **CP** інших класів **Клієнт** повинен користуватися іншою **CF**.

1.2 UML-діаграма класів АМ Абстрактна Фабрика

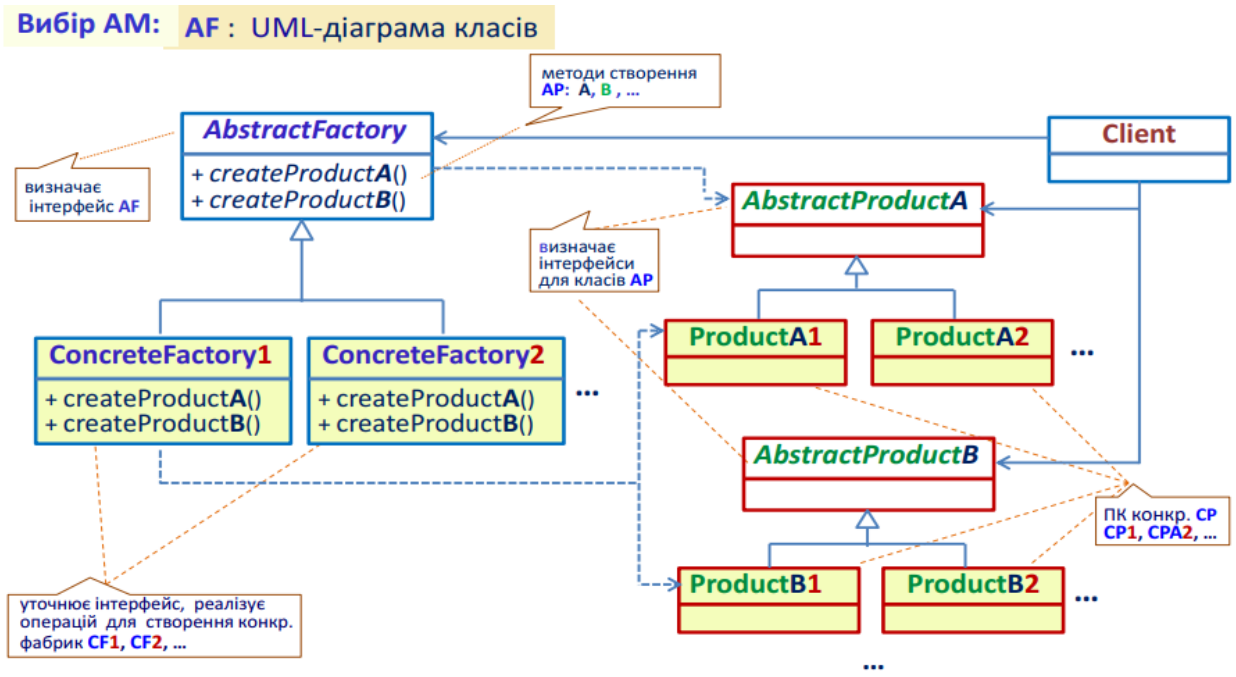

Рис.1 Діаграма класів АМ **AF**

Вказані архітектурні ієрархії звязані між собою відношеннями«використання».

2 . Опис вибраної предметної області: фабрика смарт (розумних) пристроїв

В даному випадку пропонується на основі загальної АМ AF спроектувати АМ ПЗ «Фабрику розумних пристроїв», де абстрактна фабрика смат пристроїв, де клас Абстрактна фабрика смат-пристроїв реалізує абстрактні методи для створення абстрактних смарт продуктів, використовуючи дві ієрархії класів: смарт фабрик і смарт продуктів. Для цього сутність “клієнт” використовує абстрактні класи фабрики (***Factory***) і продукту (***Smart_Device***). Це дозволяє не прив’язуватись до конкретних реалізацій цих фабрик. Конкретні фабрики

реалізують методи для створення конкретних продуктів, які успадковуються від абстрактних класів (***Factory*** і ***Smart_Device***), що задає єдиний формат користувача. Клієнт (користувач) при цьому не зобов'язаний знати всю внутрішню структу вказаних ієрархій. Це внутрішня справа самих ієрархій.

Опис ієрархії класів АМ AF МПО (відповідальності класів, описи атрибутів і методів).

В ієрархії АМ AF (рис. 5, підрозділ 5) присутні::

клас Client (використовує клас фабрики для створення об'єктів, а саме абстрактні класи, завдяки чому, не прив'язаний до конкретних реалізацій);

абстрактний клас Factory (визначає методи для створення об'єктів, при чому методи повертають абстрактні продукти, а не їх реалізацію):

метод create_device() (визначає шаблон створення продукції);

абстрактний клас Smart_device (визначає інтерфейс для класів, об'єкти яких будуть створюватись в програмі):

int ROM (поле, що визначає обсяг оперативної пам'яті для конкретного продукту);

int RAM (поле, що визначає обсяг постійної пам'яті для конкретного продукту);

float screen (поле, що визначає розмір екрану для конкретного продукту);

messages() (метод, що визначає функцію надсилання повідомлення для конкретного продукту);

play_music() (метод, що визначає функцію відтворення музики для конкретного продукту);

конкретні класи фабрик Xiaomi_factory, Meizu_factory, Doogee_factory (реалізують абстрактні методи базового класу і визначають які конкретні продукти використовувати):

метод create_device() (конкретний метод для створення продукції);

get_state() (конкретний метод для повернення статистика по створенню пристроїв);

конкретні класи продуктів Smartphone, Smart_watch, Tablet (представляють конкретну реалізацію абстрактних класів):

унаслідують поля RAM, ROM, screen від абстрактного класу Smart_device;

call() (метод класу Smartphone, реалізує функцію дзвінків);

count_steps() (метод класу Smart_watch, реалізує функцію підрахування кроків);

take_stylus() (метод класу Tablet, реалізує функцію активації стилуса).

2.2 Використання інструментальних засобів проектування для побудови АМ смарт AF

/// Тут студент повинен в розгорнутому вигляді представити покрокову інструкцію з скріншотами з використання ним середовища проектування IBM Rational Software Architect в ході виконання лабораторної роботи. Також розмістити спосіб застосування даного середовища в проектуванні тієї чи іншої діаграми, спосіб застосування для генерації «скелета» програмного коду на тій чи іншій мові програмування. ///

2.3 Реалізація проекту:

У IBM Software Architect відкриваю вікно « Моделювання» та створила новий проект, вибравши пункт « Файл»> «Новий»> «Модельний проект» у головному меню.

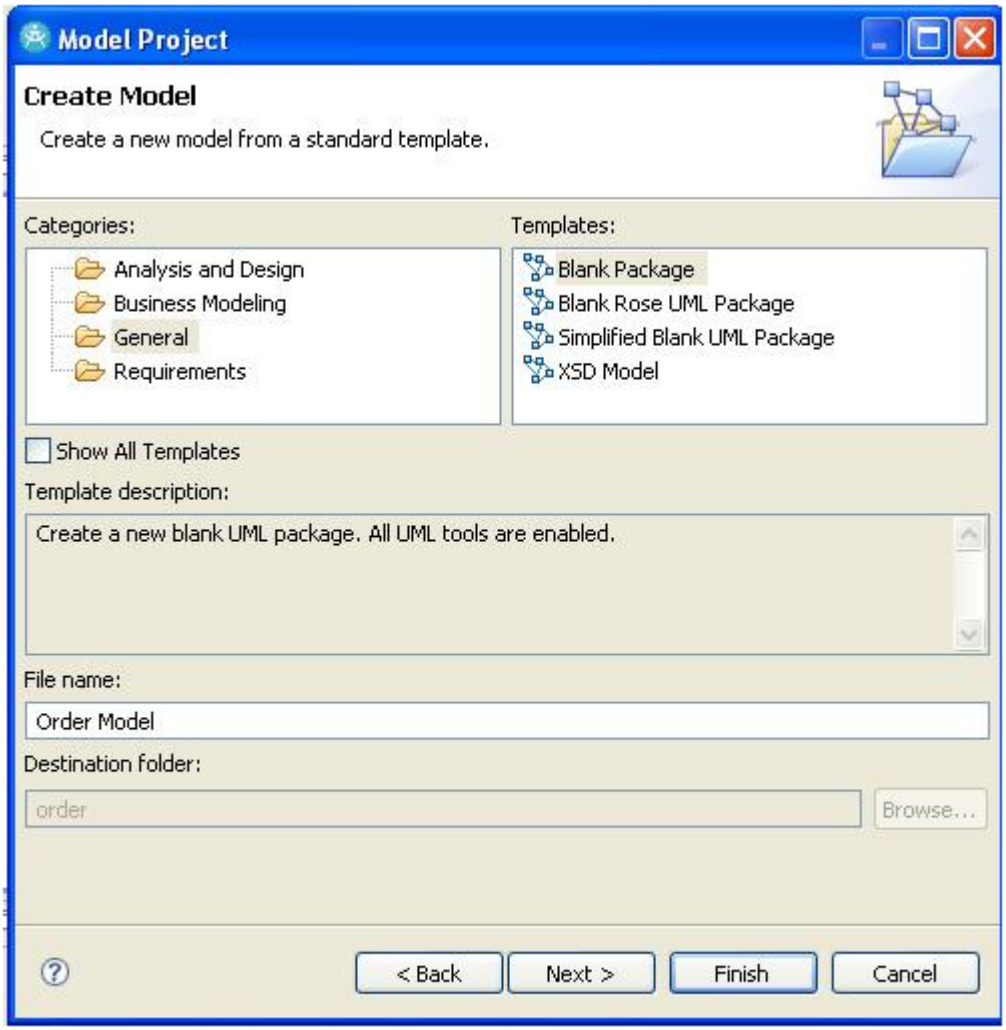

Рисунок 1 - Створення майстра UML-моделі

Додаю пакет, який викликається 2lab до створеної моделі

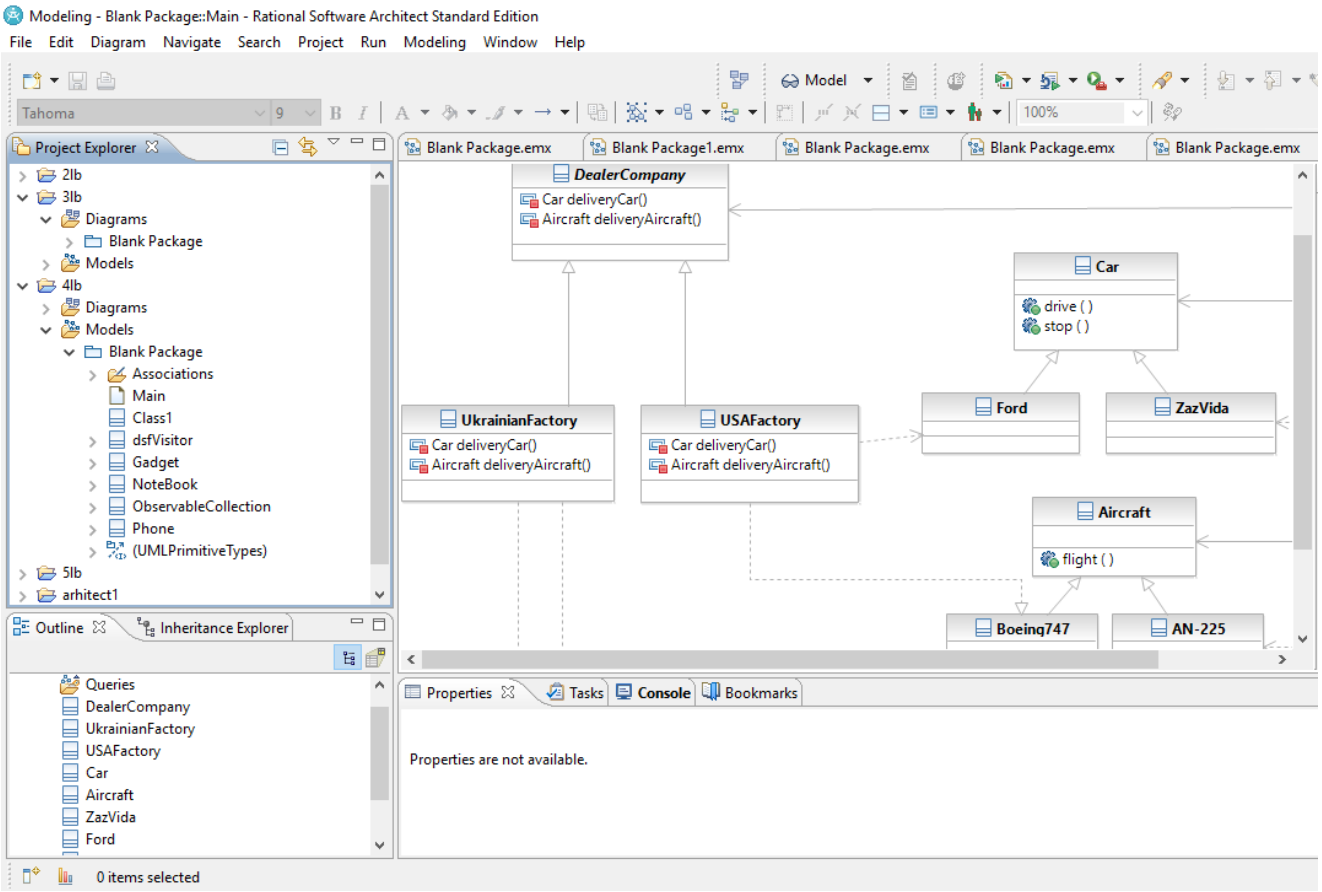

Рисунок 2 - Модель UML-зразка створена

Розробляю діаграму класів та діаграму варіантів використання “Project Explorer” + “Add diagram” + “Class Diagram and Use Case Diagram”

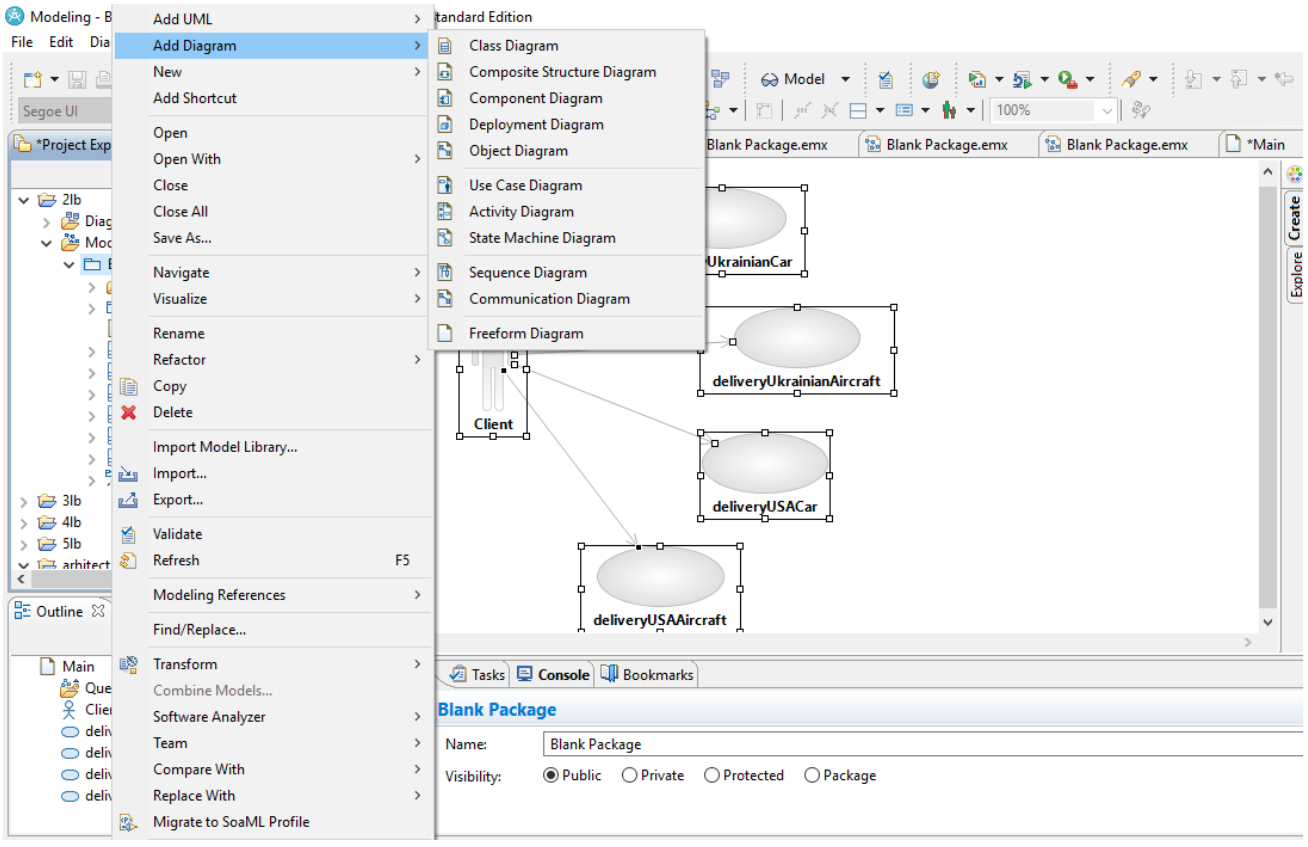

Рисунок 3 – Створення діаграм

Проектування діаграми варіантів використання смарт-АМ в IBM RSA

На діаграмі варіантів використання, що на рис. 2, присутні:

Client (використову клас фабрики для створення об’єктів);

варіант використання для створення продукції (Create device);

варіанти використання для використання конкретного продукта (Make call, Activate stylus, Count steps);

варіанти використання для використання будь-якого продукта (Listen music, Send messages). Послідовність дій по створенню діаграми класів, що описує архітектурну модель Abstract Factory, подана нижче.

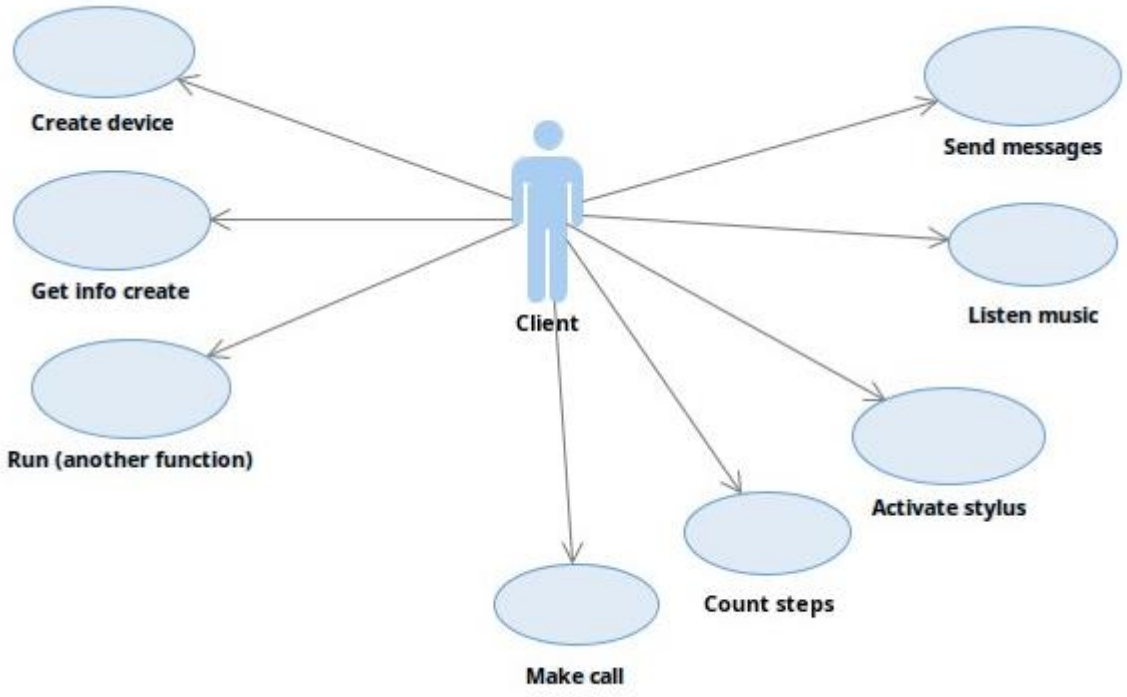

Рисунок 4 — діаграма варіантів використання

Для побудови діаграми варіантів використання (Use Case) використовувалось контнксне меню побудови UML діаграм різних типів (рис.3) з вказання вибору у списку UML діаграм випадаючогоу меню справа **Use CaseDiagramme**.

4 . Проектування Class –діаграми АМ смарт AF в IBM RSA

Для побудови діаграми класів АМ см арт AF (**Class Diagram**) також використовувалось контнксне меню побудови UML (рис.3), вказавши у списку вибору UML діаграм випадаючогоу меню справа **ClassDiagram**. В результаті побудови одержуємо діаграми класів АМ смарт AF, що описує архітектуру проектованого ПЗ (рис.4).

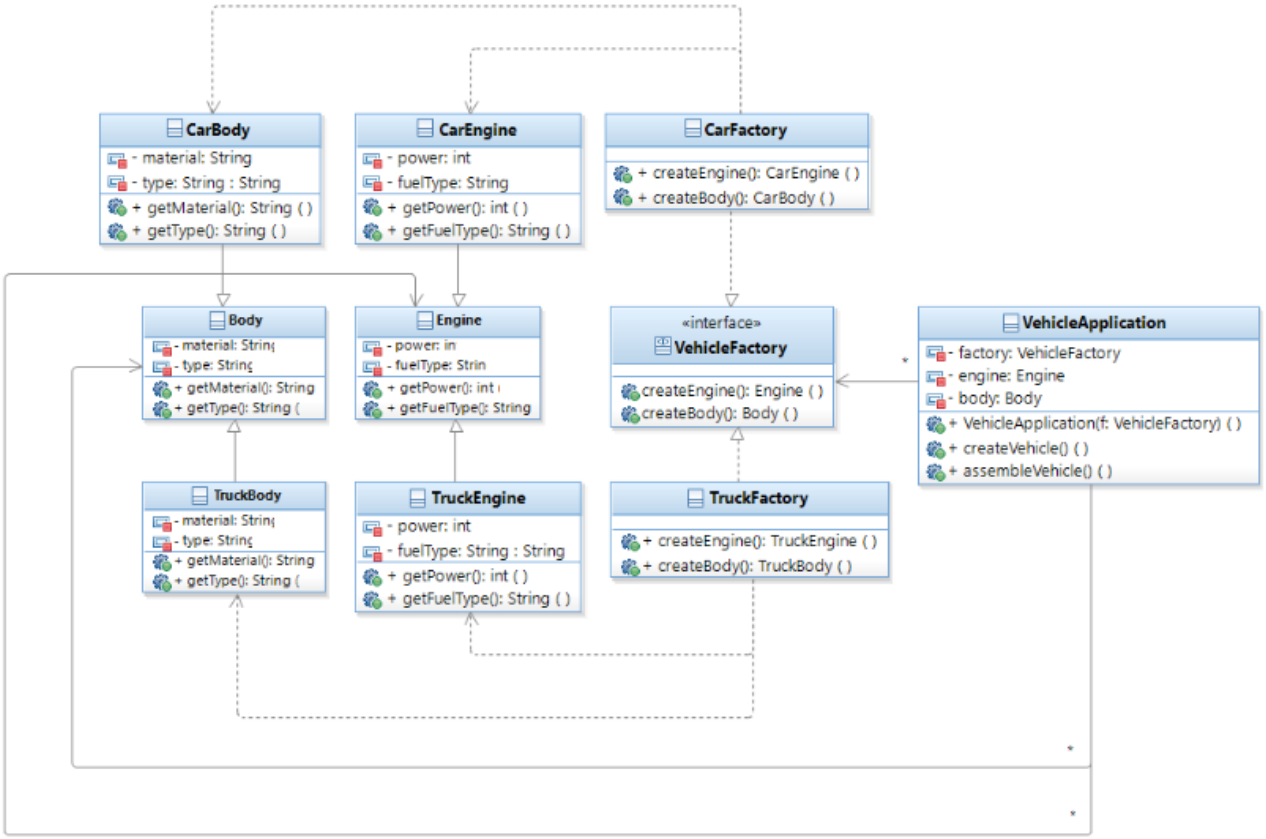

Рис. 4 – UML діаграма класів АМ смарт **AF в IBM RSA**

Проектування Sequence –діаграми АМ смарт AF в IBM RSA

Описи основних меню IBM RSA та послідовності їх застосування при створенні проекту і побудові основних діаграм для АМ смарт AF

Ґенерація програмного коду на основі побудваної архітектурної моделі – UML-

Створення C++ проекту.З генерований код на мові C++ за допомогою інструментів IBM Rational Software Architect:

Factory.cpp:

```
// абстрактний клас
фабрики public abstract class
Factory{ public:
  // метод для створення
  продукції void create_phone() =
  0;
}
```

Smart_device.cpp:

```
// абстрактний клас пристрою
public abstract class Smart_device{
protected:
  // приватне поле оперативної
  пам'яті int RAM;
  // приватне поле постійної
  пам'яті int ROM;
```

```
  // приватне поле діагоналі
  екрану float screen;
public:
  // чисто віртуальний метод, що відповідає за надсилання
  повідомлень string messages() = 0;
  // чисто віртуальний метод, що відповідає за відтворення
  музики void play_music() = 0;
}
```

Smartphone.cpp

```
// клас конкретного
продукту public class
Smartphone{ private:
  // приватне поле оперативної
  пам'яті int RAM;
  // приватне поле постійної
  пам'яті int ROM;
  // приватне поле діагоналі
  екрану float screen;
public:
  // реалізація методу, що відповідає за
  здійсненнядзвінків void call() {}
  // реалізація методу, що відповідає за
  надсиланняповідомлень string messages() {}
  // реалізація методу, що відповідає за відтворення
  музики void play_music() {} }
```

Висновки:

В ході виконання даної роботи проведений доменний аналіз предметної області, на основі якої формалізовані вимлги до проекту на основі UML Use case діаграми. Досліжена і розроблена архітектурна модель проектування ПЗ - Абстрактна фабрика із застоуванням інструментального засобу проектування ПЗ Rational Software Architect у вигляді моделі предметної області (МПО), описана діаграмою класів (UML Class Diagram). Це дозволяє виконувати комплексне моделювання сутностей системи, оперативно вносити зміни у проект згідно нових вимог користувача, не витрачаючи додаткових зусиль на перепрограмування всієї системи, забезпечуючи її архітектурну цілісність. Внесені зміни при цьому не будуть впливати на зміни усієї архітектури системи. Відповідної до вибраної АМ смарт АF проектування здійснена автоматична ґенерація програмного коду на ООП-мовах Java і C++. Даний підхід проектування ПЗ дозволяє також здійснити зворотню транформацію коду (reengineering) в UML- модель, що є дуже важливим для повторного використання раніше розробленого і працюючого коду.