МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет
імені Івана Пулюя

Кафедра програмної інженерії

Методичні вказівки для самостійної роботи

з дисципліни

«ОБ'ЄКТНІ ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТАЛЬНІ ЗАСОБИ КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

для здобувачів другого (магістерського) рівнявищої освіти
ОПП «Інженерія програмногозабезпечення»
всіх форм навчання

Тернопіль — 2025

УДК 004.41(076.5)

Укладачі:
Михалик Д.М. к.т.н., доцент
О.Р. Цебрій, к.ф.-м.н.
В.М. Стефанишин

Рецензент:
Загородна Н. В., к.т.н, доцент,
завідувач кафедрою кібербезпеки ТНТУ ім. Івана Пулюя

Розглянуто й затверджено на засіданні кафедри програмної інженерії.
Протокол № 1 від 29.08.2025.

Розглянуто й затверджено на засіданні методичної комісії факультету комп'ютерно-інформаційних систем та програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 1 від 01.09.2025.

Методичні вказівки для самостійної роботи з дисципліни «Об'єктні технології та інструментальні засоби конструювання програмного забезпечення» для здобувачів другого (магістерського) рівня вищої освіти ОПП «Інженерія програмного забезпечення» всіх форм навчання / Укладач: Михалик Д.М., Цебрій О.Р., Стефанишин В.М.,– Тернопіль: ТНТУ ім. І. Пулюя, 2025 – 40 с.

Відповідальний за випуск: О.Р. Цебрій, к.ф.-м.н.

АНОТАЦІЯ

Самостійна робота студента (СРС) є невід'ємною складовою освітнього процесу та спрямована на поглиблення, узагальнення й закріплення теоретичних знань, отриманих під час лекційних занять, формування практичних навичок самостійного опрацювання навчального матеріалу, а також підготовку до виконання лабораторних робіт.

У межах вивчення дисципліни «Об'єктні технології та інструментальні засоби конструювання програмного забезпечення» самостійна робота спрямована на формування у здобувачів другого (магістерського) рівня вищої освіти навичок аналізу, проєктування та вдосконалення програмних систем із застосуванням сучасних принципів об'єктно-орієнтованого програмування та інструментальних засобів розробки. Особлива увага приділяється питанням якості та модульності програмного забезпечення, повторного використання програмних компонентів, абстрактних типів даних, універсалізації, проєктування за контрактом, успадкування, визначення класів, управління пам'яттю та правильного використання глобальних об'єктів і констант. Зміст дисципліни також передбачає застосування сучасних засобів контролю версій, аналізу структури програмних проєктів та інших інструментів конструювання програмного забезпечення.

Основні види самостійної роботи:

- Опрацювання лекційного матеріалу та рекомендованої навчальної і науково-технічної літератури.
- Підготовка до виконання та захисту лабораторних робіт, зокрема опрацювання засобів контролю версій, аналізу модульної структури програмних проєктів, абстрактних типів даних і багатопотокового програмування.
- Самостійне опрацювання окремих розділів навчальної програми, які не виносяться на лекційні заняття.
- Аналіз прикладів програмного коду, архітектурних і проєктних рішень із позицій принципів об'єктно-орієнтованого конструювання програмного забезпечення.
- Опрацювання додаткових джерел, стандартів, технічної документації

та сучасних інструментальних засобів розробки програмного забезпечення.

- Підготовка до поточного, модульного та підсумкового контролю знань.

Самостійна робота повинна сприяти розвитку здатності здобувачів самостійно аналізувати програмні системи, обґрунтовувати проєктні рішення, застосовувати сучасні методи й засоби конструювання програмного забезпечення та оцінювати їх вплив на якість, гнучкість, підтримуваність і можливість подальшого розвитку програмного продукту.

ЗМІСТ

АНОТАЦІЯ....4
ВСТУП....6
ТЕМАТИЧНИЙ ПЛАН НАВЧАЛЬНОЇ ДИСЦИПЛІНИ....9
ВИКОНАННЯ КУРСОВОГО ПРОЄКТУ....12
КРИТЕРІЇ ОЦІНЮВАННЯ РЕЗУЛЬТАТІВ НАВЧАННЯ СТУДЕНТІВ....15
ТЕМАТИКА ТА ЗМІСТ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ (СРС)....19
МАТЕРІАЛИ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ....23
ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ....29
РЕКОМЕНДОВАНІ ІНСТРУМЕНТАРІЇ ТА ДОДАТКОВІ РЕСУРСИ....33
ВИСНОВКИ....37
РЕКОМЕНДОВАНІ ДЖЕРЕЛА....39

ВСТУП

Сучасна інженерія програмного забезпечення характеризується постійним розвитком мов програмування, технологій, бібліотек, фреймворків та інструментальних засобів. Водночас якість програмних систем значною мірою визначається не конкретним технологічним стеком, а правильністю застосування фундаментальних принципів конструювання програмного забезпечення: модульності, абстракції, повторного використання, інкапсуляції, успадкування, поліморфізму, типізації та чіткого розподілу відповідальності між програмними компонентами.

Дисципліна «**Об'єктні технології та інструментальні засоби конструювання програмного забезпечення**» спрямована на вивчення основних принципів об'єктних технологій конструювання програмного забезпечення та практичних прийомів роботи з програмним кодом. Особлива увага приділяється організації процесу конструювання програмних систем, аналізу проблем, що виникають під час їх розроблення, а також вибору обґрунтованих проєктних і технологічних рішень.

Важливою складовою освітнього процесу є **самостійна робота студента (СРС)**. Вона забезпечує поглиблення та систематизацію знань, отриманих під час лекційних занять, підготовку до виконання лабораторних робіт, опрацювання окремих питань навчальної програми та формування здатності самостійно працювати з навчальною, науковою і технічною інформацією. Робочою програмою дисципліни передбачено значний обсяг самостійної роботи: 78 годин для денної та 96 годин для заочної форми навчання.

Головна мета самостійної роботи з дисципліни полягає у поглибленні теоретичних знань і розвитку практичних навичок аналізу, проєктування, конструювання та вдосконалення програмних систем із використанням об'єктно-орієнтованих технологій і сучасних інструментальних засобів розробки.

Самостійна робота повинна сприяти формуванню здатності не лише відтворювати вивчений матеріал, а й критично оцінювати програмні рішення, аналізувати структуру коду та архітектуру системи, виявляти недоліки й обґрунтовувати способи їх усунення. Такий підхід відповідає програмним результатам навчання дисципліни, зокрема здатності оцінювати методи й моделі розроблення програмного забезпечення, розробляти та оцінювати стратегії проєктування, модифікувати архітектуру програмних систем, обґрунтовано вибирати технології та забезпечувати якість програмного забезпечення.

У процесі самостійної роботи студенти повинні:

- поглибити знання щодо принципів якості, модульності та повторного використання програмного забезпечення;
- опрацювати основні підходи до побудови та використання абстрактних

типів даних;

- дослідити особливості управління пам’яттю та ресурсами програмних систем;
- поглибити знання щодо універсалізації, проєктування за контрактом та механізмів оброблення порушень контрактів;
- засвоїти принципи правильного використання успадкування, поліморфізму та інших механізмів об’єктно-орієнтованого програмування;
- навчитися визначати класи, їх відповідальності, властивості та взаємозв’язки під час проєктування програмних систем;
- дослідити особливості використання глобальних об’єктів, констант і спільних програмних ресурсів;
- ознайомитися з гнучкими підходами до організації розроблення програмного забезпечення, зокрема Agile, Scrum і Kanban, у контексті командної роботи над об’єктно-орієнтованими системами;
- підготуватися до виконання лабораторних робіт із використання систем контролю версій, аналізу модульної структури програмних проєктів, розроблення бібліотек абстрактних типів даних та застосування багатопотоковості;
- набути навичок самостійного опрацювання професійної літератури, стандартів, технічної документації та сучасних інструментальних засобів програмної інженерії.

Перелік лекційних тем дисципліни охоплює питання якості програмного забезпечення, модульності, повторного використання, абстрактних типів даних, управління пам’яттю, універсалізації, проєктування за контрактом, успадкування, об’єктно-орієнтованих методологій, визначення класів та глобальних об’єктів і констант. Практична складова дисципліни доповнює ці питання роботою із системами контролю версій, аналізом модульної структури відкритих програмних проєктів, створенням бібліотек абстрактних типів даних та використанням багатопотокового програмування.

У межах самостійної роботи студентам рекомендується використовувати не лише навчальні матеріали курсу, а й сучасну професійну літературу, офіційну технічну документацію, міжнародні стандарти та відкриті програмні проєкти. Особливу увагу слід приділяти практичному аналізу програмного коду, оцінюванню його структури, модульності, повторного використання, підтримуваності та відповідності принципам об’єктно-орієнтованого конструювання.

Матеріали для виконання самостійної роботи, теоретичні питання, завдання для опрацювання та інші навчально-методичні матеріали розміщуються у системі дистанційного навчання дисципліни. Робочою програмою передбачено використання електронного курсу «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення» для забезпечення самостійної роботи

студентів.

Важливою умовою виконання всіх видів самостійної роботи є дотримання принципів **академічної доброчесності**. Студент повинен самостійно виконувати поставлені завдання, коректно використовувати та цитувати інформаційні джерела, не допускати академічного плагіату, фабрикації або фальсифікації результатів. Відповідні вимоги до самостійності та академічної доброчесності є складовою навчально-методичного забезпечення дисципліни та поширюються на всі форми навчальної роботи.

Таким чином, самостійна робота з дисципліни повинна забезпечити системне поєднання теоретичних знань з практичним аналізом програмного забезпечення та сприяти формуванню здатності самостійно приймати й обґрунтовувати інженерні рішення під час конструювання, розвитку та супроводу сучасних програмних систем.

ТЕМАТИЧНИЙ ПЛАН НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Дисципліна: «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення».

Рівень: другий (магістерський) рівень вищої освіти.

Спеціальність: F2 «Інженерія програмного забезпечення».

Форма підсумкового контролю: екзамен.

Обсяг дисципліни: 4 кредити ECTS (120 годин). Для денної форми навчання передбачено 42 години аудиторних занять, з них 28 годин лекційних та 14 годин лабораторних занять, а також 78 годин самостійної роботи. Для заочної (дистанційної) форми навчання передбачено 24 години аудиторних занять та 96 годин самостійної роботи.

Метою дисципліни є формування у здобувачів знань і практичних навичок застосування основних принципів об’єктних технологій конструювання програмного забезпечення та сучасних прийомів роботи з програмним кодом. У межах курсу розглядаються питання організації процесу конструювання програмного забезпечення, побудови його модульної та об’єктної структури, повторного використання програмних компонентів, управління складністю програмних систем, а також причини та наслідки проблем, що виникають у процесі їх розроблення й супроводу.

Лекційний курс спрямований на формування системного розуміння принципів і методів конструювання сучасного програмного забезпечення. У процесі навчання розглядаються питання якості програмного забезпечення, модульності, повторного використання, абстрактних типів даних, управління пам’яттю, універсалізації, проєктування за контрактом, успадкування, об’єктно-орієнтованих методологій, визначення класів, а також використання глобальних об’єктів і констант.

Метою проведення лабораторних занять є формування практичних навичок використання сучасних інструментальних засобів конструювання програмного забезпечення та застосування принципів об’єктно-орієнтованого програмування під час розв’язання прикладних задач. Лабораторна складова дисципліни передбачає роботу із сучасними системами контролю версій, аналіз модульної структури відкритих програмних проєктів, розроблення бібліотеки абстрактних типів даних та застосування багатопотоковості під час роботи з графічними об’єктами.

Важливою частиною вивчення дисципліни є самостійна робота здобувачів, яка спрямована на поглиблення лекційного матеріалу, підготовку до лабораторних занять, опрацювання окремих розділів програми, роботу з додатковою навчальною та науково-технічною літературою, а також підготовку до поточного і підсумкового контролю. Робочою програмою для денної форми навчання

передбачено загалом 78 годин самостійної роботи.

Завданням дисципліни є сформувати здатність аналізувати та обґрунтовано вибирати підходи до конструювання програмного забезпечення, розробляти й оцінювати проєктні рішення, удосконалювати структуру та архітектуру програмних систем, застосовувати сучасні мови, технології та інструментальні засоби, а також забезпечувати якість програмного забезпечення на різних етапах його життєвого циклу.

За результатами вивчення дисципліни студент повинен продемонструвати такі **результати навчання**:

- РН02. Оцінювати і вибирати ефективні методи і моделі розроблення, впровадження, супроводу програмного забезпечення та управління відповідними процесами на всіх етапах життєвого циклу.
- РН06. Розробляти і оцінювати стратегії проєктування програмних засобів; обґрунтовувати, аналізувати і оцінювати варіанти проєктних рішень з точки зору якості кінцевого програмного продукту, ресурсних обмежень та інших факторів.
- РН08. Розробляти і модифікувати архітектуру програмного забезпечення для реалізації вимог замовника.
- РН09. Обґрунтовано вибирати парадигми і мови програмування для розроблення програмного забезпечення; застосовувати на практиці сучасні засоби розроблення програмного забезпечення.
- РН11. Забезпечувати якість на всіх стадіях життєвого циклу програмного забезпечення, у тому числі з використанням релевантних моделей та методів оцінювання, а також засобів автоматизованого тестування і верифікації програмного забезпечення.

Вивчення навчальної дисципліни передбачає формування та розвиток у студентів таких **компетентностей**.

Інтегральна компетентність. Здатність розв’язувати складні спеціалізовані завдання або практичні проблеми інженерії програмного забезпечення, що характеризуються комплексністю та невизначеністю умов, із застосуванням теорій та методів інформаційних технологій.

Загальні компетентності

- ЗК01. Здатність до абстрактного мислення, аналізу та синтезу.
- ЗК03. Здатність проводити дослідження на відповідному рівні.
- ЗК05. Здатність генерувати нові ідеї (креативність).
- Спеціальні (фахові) компетентності
- СК01. Здатність аналізувати предметні області, формувати, класифікувати вимоги до програмного забезпечення.
- СК02. Здатність розробляти і реалізовувати наукові та/або прикладні проєкти у сфері інженерії програмного забезпечення.

- СК03. Здатність проєктувати архітектуру програмного забезпечення,

моделювати процеси функціонування окремих підсистем і модулів.

- СК08. Здатність розробляти і координувати процеси, етапи та ітерації життєвого циклу програмного забезпечення на основі застосування сучасних моделей, методів та технологій розроблення програмного забезпечення.
- СК09. Здатність забезпечувати якість програмного забезпечення.

Знання, уміння, комунікація, автономія та відповідальність

Знання:

- Зн1. Спеціалізовані концептуальні знання, що включають сучасні наукові здобутки у сфері професійної діяльності або галузі знань і є основою для оригінального мислення та проведення досліджень.
- Зн2. Критичне осмислення проблем у галузі та на межі галузей знань.

Уміння:

- Ум1. Спеціалізовані уміння та навички розв’язання проблем, необхідні для проведення досліджень та/або провадження інноваційної діяльності з метою розвитку нових знань і процедур.
- Ум2. Здатність інтегрувати знання та розв’язувати складні задачі у широких або мультидисциплінарних контекстах.
- Ум3. Здатність розв’язувати проблеми у нових або незнайомих середовищах за наявності неповної чи обмеженої інформації з урахуванням аспектів соціальної та етичної відповідальності.

Комунікація:

- К1. Зрозуміле і недвозначне донесення власних знань, висновків та аргументації до фахівців і нефахівців, зокрема до осіб, які навчаються.

Автономія та відповідальність:

- АВ1. Управління робочими або навчальними процесами, які є складними, непередбачуваними та потребують нових стратегічних підходів.
- АВ2. Відповідальність за внесок до професійних знань і практики та/або оцінювання результатів діяльності команд і колективів.
- АВ3. Здатність продовжувати навчання з високим ступенем автономії.

ВИКОНАННЯ КУРСОВОГО ПРОЄКТУ

Важливою складовою вивчення дисципліни «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення» є виконання курсового проєкту (КП), який забезпечує практичне закріплення знань, отриманих під час лекційних, лабораторних занять і самостійної роботи.

Метою курсового проєкту є поглиблення та систематизація теоретичних знань і практичних навичок з об’єктно-орієнтованого програмування, архітектурного аналізу та конструювання програмного забезпечення, а також формування здатності самостійно аналізувати існуючі програмні системи, виявляти їх недоліки, обґрунтовувати та реалізовувати відповідні покращення.

Курсовий проєкт виконується здобувачем самостійно протягом семестру відповідно до затвердженої теми, індивідуального завдання та календарного плану. Робота повинна демонструвати здатність здобувача проводити аналіз програмного продукту, приймати обґрунтовані інженерні рішення, застосовувати сучасні принципи об’єктно-орієнтованого проєктування та використовувати відповідні інструментальні засоби розробки.

Основні вимоги до змісту, структури, порядку виконання, оформлення, подання, перевірки та захисту курсового проєкту визначено в окремому виданні — «*Методичні вказівки до виконання курсового проєкту з дисципліни “Об’єктні технології та інструментальні засоби конструювання програмного забезпечення”*». У цих методичних вказівках наведено правила організації роботи, вимоги до пояснювальної записки, оформлення рисунків, таблиць, UML-діаграм, програмного коду, списку використаних джерел, академічної доброчесності, презентації та процедури захисту.

Основою курсового проєкту є **аналіз існуючого програмного продукту**, переважно проєкту з відкритим кодом, дослідження його архітектури, структури програмного коду, технологічного стеку та якості реалізації. На підставі проведеного аналізу здобувач визначає проблемні місця та пропонує обґрунтовані зміни, спрямовані на покращення архітектури, модульності, масштабованості, підтримуваності, якості або функціональних можливостей програмної системи.

У межах виконання курсового проєкту передбачається проходження повного циклу робіт:

Аналіз програмного продукту — дослідження предметної області, функціонального призначення системи, її архітектури, структури коду, технологій та наявних проблем.

Формування вимог до вдосконалення — визначення функціональних і нефункціональних вимог до запропонованих змін.

Проєктування покращень — розроблення архітектурних і структурних рішень, визначення змін у системі класів та взаємодії компонентів, побудова

необхідних UML-моделей.

Реалізація змін — виконання рефакторингу, розширення функціональності, оптимізації або інтеграції відповідно до поставленого завдання.

Оцінювання результатів — тестування реалізованих змін, аналіз якості коду, продуктивності, масштабованості, підтримуваності та інших характеристик програмного забезпечення.

Оформлення та захист результатів — підготовка пояснювальної записки, ілюстративного матеріалу, презентації та публічний захист отриманих результатів.

Такий порядок відповідає трьом основним макроетапам виконання проєкту: підготовчому, основному та заключному, які охоплюють відповідно аналіз, проєктування й реалізацію, а також оцінювання та оформлення отриманих результатів.

Рекомендована структура основної частини пояснювальної записки передбачає три розділи:

Аналіз існуючого програмного продукту та вимоги до вдосконалення — аналіз архітектури, програмного коду, використаних технологій, визначення проблем та формування вимог.

Проєктування вдосконалень та архітектурних рішень — розроблення запропонованих змін, UML-моделей, застосування відповідних архітектурних і проєктних рішень.

Реалізація змін та оцінка якості програмного забезпечення — впровадження запропонованих покращень, тестування та порівняльний аналіз отриманих результатів.

Під час виконання КП здобувачі повинні застосовувати знання з об’єктно-орієнтованого аналізу та проєктування, принципів ООР і SOLID, модульності, повторного використання, шаблонів проєктування, рефакторингу та сучасних архітектурних підходів. Для моделювання можуть використовуватися UML-діаграми класів, послідовностей, компонентів, розгортання та інші види моделей відповідно до специфіки проєкту.

З тематикою курсового проєкту тісно пов’язані лекційні та лабораторні заняття дисципліни. Отримані під час їх опрацювання знання є теоретичною та практичною основою для аналізу програмного продукту, формування проєктних рішень і реалізації запланованих покращень.

Тематика КП може охоплювати вдосконалення архітектури програмних систем, реінжиніринг і рефакторинг, підвищення якості та продуктивності програмного забезпечення, розподілені й хмарні системи, інтелектуальні системи, відмовостійке та безпекоорієнтоване ПЗ, системи реального часу, IoT, а також інструментальні засоби програмної інженерії. Приклади рекомендованих напрямів і тем наведено у відповідних методичних вказівках.

Оцінювання курсового проєкту

Курсовий проєкт оцінюється за 100-бальною шкалою. Під час оцінювання враховуються зміст та глибина проведеного аналізу, обґрунтованість запропонованих рішень, якість реалізації змін, коректність UML-моделей, результати тестування та оцінювання ефективності, якість оформлення пояснювальної записки, презентація результатів та відповіді здобувача під час захисту. Детальні критерії оцінювання наведено в методичних вказівках до виконання курсового проєкту.

Вимоги до оформлення курсового проєкту

Детальні вимоги до оформлення матеріалів курсового проєкту визначаються методичними вказівками до виконання курсового проєкту з цієї дисципліни. Пояснювальна записка оформлюється відповідно до вимог чинних державних стандартів України, зокрема ДСТУ 3008:2015 та ДСТУ 8302:2015. Для оформлення бібліографічних джерел також допускається використання стилю IEEE за умови дотримання єдиного стилю в межах усього документа.

Основний текст пояснювальної записки оформлюється на аркушах формату A4 шрифтом Times New Roman, 14 пт, з міжрядковим інтервалом 1,5, абзацним відступом 1,25 см та вирівнюванням тексту по ширині.

Усі рисунки, схеми, UML-діаграми та інший ілюстративний матеріал повинні бути чіткими, читабельними, мати номер і змістовну назву та обов’язково згадуватися в основному тексті перед їх наведенням. Аналогічні вимоги встановлено для таблиць і лістингів програмного коду.

Програмний код, UML-моделі та інші електронні артефакти проєкту повинні бути подані разом із пояснювальною запискою відповідно до вимог методичних вказівок. Для зберігання та супроводу вихідного коду рекомендується використання систем контролю версій і відповідного Git-репозиторію.

Отже, в цих методичних вказівках до самостійної роботи доцільно навести лише загальну характеристику курсового проєкту та його зв’язок із дисципліною, а всі детальні вимоги щодо виконання, структури, оформлення, оцінювання та захисту слід застосовувати відповідно до окремих «*Методичних вказівок до виконання курсового проєкту з дисципліни “Об’єктні технології та інструментальні засоби конструювання програмного забезпечення”*».

КРИТЕРІЇ ОЦІНЮВАННЯ РЕЗУЛЬТАТІВ НАВЧАННЯ СТУДЕНТІВ

Теоретичний курс (тестування). Оцінюється рівень засвоєння теоретичного матеріалу дисципліни, розуміння основних принципів об’єктно-орієнтованого конструювання програмного забезпечення, модульності, повторного використання, абстрактних типів даних, універсалізації, проєктування за контрактом, успадкування, визначення класів, глобальних об’єктів і констант, а також здатність аргументовано застосовувати вивчені підходи під час аналізу програмних систем.

Лабораторні роботи. Оцінюється правильність і повнота виконання лабораторних завдань, практичне застосування теоретичних знань, якість програмної реалізації, використання відповідних інструментальних засобів, коректність отриманих результатів та обґрунтованість сформульованих висновків. Лабораторний практикум охоплює роботу із системами контролю версій, аналіз модульної структури програмного проєкту, розроблення бібліотеки абстрактних типів даних та використання багатопотоковості під час роботи з графічними об’єктами.

Форма підсумкового семестрового контролю оцінювання студентів за X семестр: – екзамен, курсовий проект.

Модуль 1			Модуль 2			Підсумковий контроль	Разом з дисципліни
Аудиторна та самостійна робота			Аудиторна та самостійна робота				
Теоретичний курс (тестування)	Практична робота		Теоретичний курс (тестування)	Практична робота			
15	**20**		**20**	**20**		**25**	**100**
№ лекцій	Вид робіт	Бал	№ лекцій	Вид робіт	Бал	екзамен	
Лекція № 1	Техніка безпеки		Лекція № 8				
Лекція № 2			Лекція № 9	Лаб. роб. № 3	10		
Лекція № 3			Лекція № 10				
Лекція № 4	Лаб. роб. № 1	10	Лекція № 11				
Лекція № 5			Лекція № 12	Лаб. роб. № 4	10		
Лекція № 6			Лекція № 13				
Лекція № 7	Лаб. роб. № 2	10	Лекція № 14				
Проміжний кон-ль			Підсумковий кон-ль				

Розподіл балів відповідає робочій програмі дисципліни: 20 балів за теоретичну складову першого модуля, 15 балів за практичну; 20 і 20 балів відповідно за другий модуль та 25 балів за підсумковий контроль.

Практична складова першого модуля формується за результатами виконання лабораторних робіт № 1 та № 2, а другого модуля – лабораторних робіт № 3 та № 4.

Оцінювання курсового проєкту здійснюється окремо відповідно до вимог, критеріїв та 100-бальної шкали, визначених у «*Методичних вказівках до виконання курсового проєкту з дисципліни "Об'єктні технології та інструментальні засоби конструювання програмного забезпечення"*». Під час оцінювання КП враховуються якість аналізу існуючого програмного продукту, обґрунтованість запропонованих архітектурних та структурних покращень, якість реалізації, UML-моделей, тестування, оформлення та захисту роботи.

Оцінювання самостійної роботи

Результати самостійної роботи студентів враховуються під час поточного та модульного контролю, перевірки й захисту лабораторних робіт, тестування, опрацювання теоретичних питань та підготовки до підсумкового контролю. Самостійна робота не розглядається ізольовано від інших видів навчальної діяльності, а інтегрується в загальну систему оцінювання дисципліни.

Під час оцінювання самостійної роботи враховуються:

- глибина опрацювання теоретичного матеріалу – розуміння основних понять і принципів дисципліни, здатність пояснювати їх взаємозв'язок та практичне значення;
- якість підготовки до лабораторних робіт – попереднє опрацювання необхідних технологій, інструментів, програмних засобів і теоретичних положень;
- самостійність аналізу програмних рішень – здатність визначати переваги й недоліки структури програмного забезпечення, виявляти проблеми модульності, повторного використання, успадкування та розподілу відповідальності між класами;
- уміння працювати з інформаційними джерелами – використання навчальної та наукової літератури, стандартів, офіційної технічної документації та інших достовірних професійних джерел;
- обґрунтованість висновків – здатність аргументувати запропоновані рішення та пояснювати їх вплив на якість, підтримуваність, розширюваність і надійність програмного забезпечення;
- дотримання академічної доброчесності – самостійне виконання завдань, належне використання та цитування джерел, відсутність академічного плагіату, фабрикації й фальсифікації результатів.

Критерії оцінювання результатів навчання

Оцінка «А» (90–100 балів)

Студент демонструє системне та глибоке розуміння принципів об'єктно-орієнтованого конструювання програмного забезпечення; вільно оперує поняттями модульності, абстракції, повторного використання, універсалізації, проєктування за контрактом, успадкування та поліморфізму. Лабораторні роботи

виконані повністю та коректно, студент аргументовано пояснює прийняті програмні й проєктні рішення, здатний аналізувати альтернативні підходи та самостійно застосовувати отримані знання до нових задач.

Оцінка «B» (82–89 балів)

Студент добре володіє теоретичним матеріалом, правильно застосовує основні принципи об’єктно-орієнтованого програмування та конструювання програмного забезпечення. Лабораторні завдання виконані якісно, можливі окремі незначні недоліки, які не впливають на загальну правильність результату. Студент здатний обґрунтовувати структуру класів, застосування успадкування, композиції, універсалізації та інших механізмів, але може допускати окремі неточності під час аналізу складніших ситуацій.

Оцінка «C» (74–81 бал)

Студент розуміє основний зміст дисципліни та здатний застосовувати вивчені підходи для розв’язання типових задач. Лабораторні роботи виконані в основному правильно, але містять окремі помилки або недостатньо обґрунтовані рішення. Розуміння окремих питань модульності, контрактів, універсалізації, успадкування або організації системи класів може бути неповним.

Оцінка «D–E» (60–73 бали)

Студент володіє мінімально необхідним обсягом знань з дисципліни, розуміє базові принципи об’єктно-орієнтованого програмування та може виконувати прості типові завдання. Лабораторні роботи містять суттєві недоліки, окремі рішення реалізовані неповно або недостатньо обґрунтовано. Під час пояснення програмної структури та прийнятих рішень студент допускає помилки, однак демонструє загальне розуміння основних тем курсу.

Оцінка «FX» (35–59 балів)

Студент демонструє фрагментарне розуміння теоретичного матеріалу. Наявні суттєві прогалини у знаннях принципів модульності, повторного використання, абстрактних типів даних, контрактного проєктування, успадкування та інших об’єктних технологій. Лабораторні роботи виконані частково або містять значні помилки, студент має труднощі з обґрунтуванням власних рішень. Для отримання позитивного результату необхідне додаткове опрацювання матеріалу та повторне проходження відповідних форм контролю.

Оцінка «F» (0–34 бали)

Студент не демонструє достатнього розуміння основних положень дисципліни, не володіє базовими принципами об’єктно-орієнтованого конструювання програмного забезпечення та не здатний застосовувати їх під час виконання практичних завдань. Значна частина лабораторних робіт не виконана або результати не відповідають поставленим завданням. Необхідне повторне вивчення дисципліни.

При виставленні підсумкової оцінки особлива увага приділяється не лише формальному виконанню завдань, а й самостійності роботи, розумінню прийнятих

програмних рішень, коректності використання професійної термінології, здатності аналізувати структуру програмного забезпечення та аргументовано обирати відповідні об’єктні технології й інструментальні засоби.

ТЕМАТИКА ТА ЗМІСТ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ (СРС)

Самостійна робота магістрантів передбачає поглиблене опрацювання теоретичного матеріалу дисципліни, підготовку до лабораторних занять, роботу з професійною літературою та технічною документацією, а також самостійний аналіз програмних рішень. Робочою програмою передбачено окреме опрацювання розділів програми, які не виносяться на лекційні заняття, а також підготовку до лабораторних робіт і підсумкового контролю.

Нижче наведено рекомендований перелік тем для самостійного опрацювання, сформований відповідно до змісту лекційного та лабораторного курсу дисципліни.

Тема 1. Якість, модульність та повторне використання програмного забезпечення

Мета: поглибити розуміння принципів побудови якісних і модульних програмних систем та дослідити практичні механізми повторного використання програмного коду і компонентів.

Завдання для самостійного опрацювання:

1. Проаналізувати взаємозв’язок між модульністю, підтримуваністю, розширюваністю та повторним використанням програмного забезпечення.
2. Дослідити основні механізми повторного використання програмних компонентів: бібліотеки, класи, компоненти, шаблони проєктування та програмні фреймворки.
3. Обрати відкритий програмний проєкт і проаналізувати його модульну структуру: визначити основні модулі, їх відповідальності та залежності.
4. Виявити приклади надмірного зв’язування компонентів або дублювання програмного коду та запропонувати можливі способи вдосконалення структури системи.

Тема безпосередньо пов’язана з лекціями про якість, модульність і повторне використання, а також із лабораторною роботою з аналізу модульної структури відкритого програмного проєкту.

Питання для самоконтролю:

1. Які характеристики програмної системи найбільше залежать від правильної модульної декомпозиції?
2. Чим повторне використання програмного компонента відрізняється від простого копіювання програмного коду?
3. Які ознаки можуть свідчити про надмірну залежність між модулями?
4. Як модульність впливає на тестування та супровід програмного забезпечення?

Тема 2. Абстрактні типи даних, універсалізація та управління ресурсами

Мета: поглибити знання щодо побудови абстракцій даних, використання універсальних програмних компонентів та організації ефективної роботи з пам’яттю й іншими ресурсами програмної системи.

Завдання для самостійного опрацювання:

1. Дослідити принципи побудови абстрактних типів даних та визначити відмінність між абстракцією і конкретною реалізацією структури даних.
2. Розглянути реалізацію типових АТД — стеку, черги, списку та дерева — у сучасних об’єктно-орієнтованих мовах програмування.
3. Порівняти необмежену та обмежену універсалізацію та визначити випадки доцільного застосування кожного підходу.
4. Проаналізувати механізми автоматичного та ручного управління пам’яттю в сучасних мовах програмування.
5. Розробити невеликий універсальний програмний компонент, наприклад параметризований контейнер або колекцію, та пояснити обмеження, які доцільно накласти на його параметри типу.

Зміст теми відповідає лекційним питанням абстрактних типів даних, управління пам’яттю та універсалізації, а також пов’язаний із лабораторною роботою з розроблення бібліотеки абстрактних типів даних.

Питання для самоконтролю:

1. У чому полягає основна перевага абстрактного типу даних порівняно з безпосередньою роботою зі структурою його реалізації?
2. Яку проблему вирішує універсалізація програмних компонентів?
3. У яких випадках доцільно використовувати обмежену універсалізацію?
4. Які переваги та недоліки мають автоматичне й ручне управління пам’яттю?
5. Як абстракція даних сприяє повторному використанню програмного забезпечення?

Тема 3. Проєктування за контрактом, успадкування та формування системи класів

Мета: сформувати здатність обґрунтовано визначати контракти програмних компонентів, будувати коректні ієрархії успадкування та виділяти класи відповідно до їх відповідальностей у програмній системі.

Завдання для самостійного опрацювання:

1. Для обраного класу визначити його передумови, постумови та інваріант і пояснити відповідальність клієнта та постачальника.
2. Проаналізувати приклади порушення контракту та визначити можливі причини виникнення відповідних помилок.
3. Для декількох пар класів визначити, який зв’язок є доцільнішим — успадкування, композиція чи агрегація — та обґрунтувати вибір.
4. Дослідити типові помилки застосування успадкування, зокрема використання його виключно заради повторного використання реалізації.
5. Для обраної предметної області сформувати перелік потенційних класів, визначити їх основні відповідальності, атрибути та операції.
6. Проаналізувати отриману систему класів та виявити можливі

надлишкові, пасивні або неправильно визначені класи.

Робоча програма приділяє цим питанням окремий блок лекцій: проєктування за контрактом, порушення контракту, принципи успадкування, правильне використання успадкування та методи визначення класів.

Питання для самоконтролю:

1. Яку роль відіграють передумови, постумови та інваріанти у забезпеченні коректності програмного компонента?
2. Хто несе відповідальність за виконання передумови — клієнт чи постачальник?
3. За якої умови між двома класами доцільно встановлювати відношення успадкування?
4. Коли композиція є кращим рішенням, ніж успадкування?
5. Які ознаки можуть свідчити про те, що потенційна сутність не повинна бути окремим класом?

Тема 4. Інструментальні засоби конструювання та аналізу програмного забезпечення

Мета: сформувати навички самостійного використання сучасних інструментальних засобів для контролю версій, аналізу структури програмних систем, контролю якості програмного коду та підтримки процесу його вдосконалення.

Завдання для самостійного опрацювання:

1. Дослідити принципи роботи розподілених систем контролю версій та основні операції Git.
2. Створити репозиторій для навчального програмного проєкту, організувати систему гілок та виконати декілька контрольованих змін програмного коду.
3. Проаналізувати можливості сучасних IDE щодо навігації по структурі класів, пошуку залежностей і автоматизованого рефакторингу.
4. Ознайомитися з можливостями засобів статичного аналізу програмного коду та визначити типи проблем, які вони дозволяють виявляти.
5. Провести аналіз невеликого відкритого програмного проєкту за допомогою одного із засобів контролю якості коду та класифікувати виявлені проблеми.
6. Дослідити можливості використання інструментальних засобів для підтримки рефакторингу, тестування та контролю якості програмного забезпечення.

Робоча програма безпосередньо передбачає лабораторне опрацювання систем контролю версій та аналіз модульної структури open-source проєкту. У методичних вказівках до курсового проєкту цей напрям розширюється використанням Git/GitHub/GitLab, IDE, засобів моделювання та інструментів статичного аналізу, зокрема SonarQube, ESLint і Pylint.

Питання для самоконтролю:

1. Які переваги дає використання системи контролю версій під час командної та індивідуальної розробки?
2. Які типи дефектів можуть бути виявлені засобами статичного аналізу коду?
3. Чим автоматизований рефакторинг відрізняється від звичайного редагування програмного коду?
4. Які показники можуть бути використані для оцінювання підтримуваності програмного забезпечення?
5. Як інструментальні засоби можуть сприяти забезпеченню якості програмного продукту?

Опрацювання наведених тем повинно бути спрямоване не лише на засвоєння додаткового теоретичного матеріалу, а й на формування здатності самостійно аналізувати програмний код, аргументувати вибір програмних рішень, користуватися професійною документацією та застосовувати сучасні інструментальні засоби програмної інженерії. Такий підхід узгоджується з програмними результатами навчання дисципліни щодо вибору методів розроблення, оцінювання проєктних рішень, модифікації архітектури та забезпечення якості програмного забезпечення.

МАТЕРІАЛИ ДЛЯ САМОСТІЙНОГО ВИВЧЕННЯ

Самостійне опрацювання матеріалу з дисципліни «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення» спрямоване на поглиблення знань щодо принципів побудови об’єктно-орієнтованих програмних систем, формування навичок аналізу програмного коду та здатності обґрунтовано застосовувати відповідні технології, механізми й інструментальні засоби.

Під час самостійної роботи студент повинен не лише ознайомлюватися з теоретичними положеннями, а й аналізувати їх практичне застосування у програмних проєктах, порівнювати альтернативні рішення, виявляти недоліки структури програмного забезпечення та пропонувати способи її вдосконалення.

1. Аналіз програмної системи та виділення класів

Важливою складовою об’єктно-орієнтованого конструювання є правильне визначення класів, їх відповідальностей та взаємозв’язків. На початкових етапах аналізу необхідно визначити суттєві поняття предметної області та відокремити їх від деталей конкретної програмної реалізації.

Клас повинен представляти чітко визначену абстракцію, мати змістовне ім’я, логічно пов’язаний набір властивостей та операцій і відповідати за одну відносно цілісну частину поведінки системи. Класи з великою кількістю непов’язаних обов’язків або, навпаки, класи, що фактично реалізують лише одну процедурну функцію, можуть свідчити про недоліки декомпозиції.

Під час визначення потенційних класів доцільно аналізувати:

- поняття та терміни предметної області;
- вимоги до програмної системи;
- об’єкти, що мають власний стан і поведінку;
- операції, які природно належать певній абстракції;
- взаємодію між потенційними об’єктами;
- семантичні обмеження та правила предметної області.

У вихідному матеріалі цей підхід також ґрунтується на тому, що класи аналізу повинні відображати предметну область, мати чіткі обов’язки, високу внутрішню зв’язність і мінімальну кількість зайвих залежностей.

Одним із допоміжних способів первинного пошуку потенційних класів є аналіз текстових вимог: іменники та іменні групи можуть вказувати на можливі класи або атрибути, а дієслова — на операції та відповідальності. Однак такий підхід не повинен використовуватися механічно: кожну потенційну абстракцію необхідно перевірити з точки зору її реальної ролі у програмній системі.

2. Модульність програмного забезпечення

Модульність є одним із ключових принципів конструювання програмного забезпечення. Вона передбачає розподіл системи на відносно незалежні програмні компоненти, кожен із яких має чітко визначене призначення та інтерфейс взаємодії.

Під час самостійного опрацювання необхідно звернути увагу на такі характеристики якісної модульної структури:

- висока внутрішня зв’язність компонентів;
- мінімальна залежність між модулями;
- чітко визначені інтерфейси;
- приховування деталей реалізації;
- можливість локальної модифікації окремих компонентів;
- можливість незалежного тестування;
- повторне використання програмних компонентів.

Студентам рекомендується обрати невеликий відкритий програмний проєкт та визначити його основні модулі, залежності між ними, потенційні циклічні залежності й компоненти з надмірною відповідальністю.

3. Абстрактні типи даних

Абстрактний тип даних визначає множину можливих значень та набір операцій над ними незалежно від конкретного способу реалізації. Такий підхід дозволяє відокремити **семантику програмного компонента від структури його внутрішнього представлення**.

Під час самостійного опрацювання рекомендується дослідити реалізацію таких абстрактних типів даних:

- стек;
- черга;
- список;
- множина;
- словник;
- дерево.

Для кожного АТД необхідно визначити основні операції, можливі обмеження, інваріанти та декілька альтернативних способів реалізації.

Особливу увагу слід приділити принципу інформаційного приховування: клієнт програмного компонента повинен залежати від його публічного інтерфейсу, а не від конкретної внутрішньої структури.

4. Повторне використання програмного забезпечення

Повторне використання передбачає застосування вже створених програмних компонентів у нових контекстах без необхідності повторної реалізації однакової функціональності.

Основними механізмами повторного використання є:

- бібліотеки;
- універсальні класи;
- успадкування;
- композиція;
- програмні компоненти;

- фреймворки;
- шаблони проєктування.

Необхідно розрізняти **повторне використання абстракції** та просте копіювання програмного коду. Копіювання створює незалежні копії реалізації та ускладнює подальший супровід, тоді як повторно використовуваний компонент має власний контракт і може застосовуватися різними клієнтами.

5. Універсалізація

Універсалізація дає можливість створювати класи та програмні компоненти, поведінка яких не залежить від одного конкретного типу даних.

Наприклад, універсальна структура *STACK [G]* може використовуватися для зберігання об’єктів різних типів без необхідності створення окремої реалізації стеку для кожного з них.

Під час самостійного опрацювання студентам необхідно:

- дослідити необмежену універсалізацію;
- дослідити обмежену універсалізацію;
- проаналізувати поєднання універсалізації та успадкування;
- розглянути механізми generics/templates у сучасних мовах програмування;
- визначити переваги статичного контролю типів під час використання універсальних компонентів.

6. Проєктування за контрактом

Проєктування за контрактом дозволяє формально визначати очікувану поведінку програмних компонентів.

Контракт операції може містити:

- передумову — умову, яку повинен забезпечити клієнт перед виконанням операції;
- постумову — властивість, яку повинна забезпечити операція після успішного виконання;
- інваріант класу — умову, що характеризує допустимий стан об’єкта.

Студентам рекомендується самостійно визначити контракт для декількох типових класів, наприклад *BANK_ACCOUNT, STACK, ORDER або RESERVATION*, та проаналізувати розподіл відповідальності між клієнтом і постачальником програмної послуги.

Особливу увагу необхідно приділити тому, що контракт є не лише механізмом виявлення помилок, а й засобом специфікації та документування програмного компонента.

7. Успадкування та поліморфізм

Успадкування використовується для побудови відношень спеціалізації між класами. Клас-нащадок повинен представляти спеціалізований різновид абстракції, визначеної базовим класом.

Під час самостійної роботи необхідно навчитися розрізняти:

- відношення is-a;
- відношення has-a;
- успадкування;
- композицію;
- агрегацію;
- клієнтський зв’язок.

Успадкування не слід використовувати лише як засіб отримання доступу до готової реалізації. Воно є обґрунтованим тоді, коли між класами існує природне семантичне відношення спеціалізації.

Важливим критерієм правильно побудованого класу є висока внутрішня зв’язність і обмежена кількість зовнішніх залежностей. Аналогічний принцип наведено й у вихідному матеріалі, де рекомендується застосовувати успадкування лише за наявності природної ієрархії абстракцій.

8. Управління пам’яттю та ресурсами

Самостійне вивчення цієї теми повинно охоплювати основні способи управління пам’яттю в сучасних мовах програмування:

- статичне розміщення;
- стекову пам’ять;
- динамічну пам’ять;
- ручне керування ресурсами;
- автоматичне збирання сміття;
- управління життєвим циклом об’єктів.

Студентам рекомендується порівняти підходи, реалізовані, наприклад, у C++, Java, C# та Python, та визначити потенційні проблеми: витоки пам’яті, висячі посилання, передчасне звільнення ресурсів і надмірне утримання об’єктів.

9. Глобальні об’єкти та константи

Глобально доступна інформація може бути необхідною в програмній системі, однак її використання повинно бути контрольованим.

Доцільно розрізняти:

символічні константи;

незмінні значення;

спільні об’єкти;

одноразово ініціалізовані ресурси;

глобальний змінний стан.

Символічні константи покращують читабельність і спрощують модифікацію програмного забезпечення. Натомість надмірне використання глобального змінного стану створює приховані залежності, ускладнює тестування й зменшує автономність компонентів.

10. Інструментальні засоби конструювання програмного забезпечення

Сучасне конструювання програмного забезпечення передбачає використання комплексу інструментальних засобів.

Для самостійного опрацювання рекомендується розглянути такі категорії:

- **системи контролю версій:** Git, GitHub, GitLab;
- **середовища розробки:** Visual Studio, IntelliJ IDEA, VS Code;
- **засоби статичного аналізу:** SonarQube, ESLint, Pylint;
- **засоби автоматизованого тестування;**
- **засоби рефакторингу;**
- **засоби моделювання:** draw.io, StarUML, Visual Paradigm, PlantUML.

Робочою програмою дисципліни безпосередньо передбачено практичне опрацювання систем контролю версій, аналіз модульної структури open-source проєкту, створення бібліотеки абстрактних типів даних та використання багатопотоковості.

11. UML як допоміжний засіб об'єктного моделювання

У межах цієї дисципліни UML доцільно розглядати не як самостійну центральну тему курсу, а як допоміжний інструмент представлення структури та поведінки об'єктно-орієнтованих систем.

Для самостійного опрацювання достатньо зосередитися на:

- **Class Diagram** — класи, атрибути, операції, успадкування, асоціації, агрегація та композиція.
- **Sequence Diagram** — взаємодія об'єктів під час реалізації певного сценарію.
- **Component Diagram** — поділ програмної системи на великі програмні компоненти.
- **Deployment Diagram** — фізичне розміщення компонентів програмної системи.

Вихідний матеріал також пропонує послідовне самостійне опрацювання статичного, динамічного та інфраструктурного моделювання.

ТЕМИ ДЛЯ САМОСТІЙНИХ ДОСЛІДЖЕНЬ

Для поглибленого вивчення дисципліни студентам може бути запропоновано обрати одну тему та підготувати аналітичний **звіт, реферат або презентацію.**

Блок 1. Об'єктне проєктування та якість

- Модульність як основа підтримуваності програмного забезпечення.
- Метрики зв'язності та зчеплення класів.
- Принципи SOLID та їх практичне застосування.
- Композиція проти успадкування.
- Виявлення «God Object» та інших проблем об'єктної декомпозиції.
- Code smells як ознаки недоліків об'єктної структури.

Блок 2. Повторне використання та універсалізація

- Універсальні типи в C++, Java, C# та інших мовах.
- Обмежена універсалізація та generic constraints.
- Фреймворки та бібліотеки як механізми повторного використання.

- Шаблони проєктування як засіб повторного використання проєктних рішень.
- Повторне використання через композицію та делегування.

Блок 3. Контракти, надійність та типізація

- Design by Contract у сучасних мовах програмування.
- Передумови, постумови та інваріанти як засоби забезпечення коректності.
- Контракти та автоматизоване тестування.
- Сильна типізація як засіб попередження програмних помилок.
- Оброблення винятків і порушення контрактів.

Блок 4. Сучасні інструментальні засоби

- Git як інструмент керування еволюцією програмного продукту.
- Автоматизований рефакторинг у сучасних IDE.
- Статичний аналіз програмного коду.
- SonarQube та аналіз технічного боргу.
- Автоматизоване тестування об’єктно-орієнтованих компонентів.
- Використання AI-асистентів для аналізу, рефакторингу та документування програмного коду.

Блок 5. Сучасні підходи до організації розробки

- Agile, Scrum і Kanban у процесі розроблення програмного забезпечення.
- Інтеграція принципів об’єктного проєктування у гнучкі процеси розробки.
- Continuous Integration як засіб контролю якості програмного коду.
- Code Review як механізм підвищення якості програмного забезпечення.
- Технічний борг та його вплив на розвиток програмних систем.

Результати самостійного дослідження повинні демонструвати не лише опрацювання літературних джерел, а й **здатність студента аналізувати програмні рішення, наводити практичні приклади, порівнювати альтернативні підходи та формулювати власні аргументовані висновки.**

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

Якість, модульність та повторне використання програмного забезпечення

1. Що розуміють під якістю програмного забезпечення та які характеристики визначають якість програмного продукту?
2. Яким чином модульність впливає на підтримуваність, розширюваність і тестованість програмного забезпечення?
3. Що таке зв'язність модулів і внутрішня зчепленість компонентів?
4. Які основні критерії якісної модульної декомпозиції програмної системи?
5. Що розуміють під повторним використанням програмного забезпечення?
6. Чим повторне використання програмного компонента відрізняється від копіювання програмного коду?
7. Які механізми повторного використання застосовуються в об'єктно-орієнтованому програмуванні?
8. Які переваги та потенційні недоліки використання бібліотек, компонентів і програмних фреймворків?

Ці питання відповідають першим темам лекційного курсу, присвяченим якості, модульності та підходам до повторного використання.

Абстрактні типи даних, управління пам'яттю та універсалізація

9. Що таке абстрактний тип даних і в чому полягає його відмінність від конкретної структури даних?
10. Які складові визначають специфікацію абстрактного типу даних?
11. Наведіть приклади абстрактних типів даних та охарактеризуйте їх основні операції.
12. У чому полягає принцип інформаційного приховування?
13. Які основні способи управління пам'яттю використовуються в сучасних мовах програмування?
14. У чому полягає різниця між автоматичним і ручним управлінням пам'яттю?
15. Що таке витік пам'яті та за яких умов він може виникати?
16. Що розуміють під універсалізацією програмних компонентів?
17. У чому полягає різниця між обмеженою та необмеженою універсалізацією?
18. Які переваги забезпечує використання параметризованих класів і універсальних типів?

Робоча програма окремо виділяє теми абстрактних типів даних, управління пам'яттю та універсалізації.

Проєктування за контрактом

19. Що таке проєктування за контрактом (Design by Contract)?

20. Що таке передумова операції та яку відповідальність вона покладає на клієнта?

21. Що таке постумова та яку відповідальність вона покладає на постачальника?

22. Що таке інваріант класу?

23. Яким чином контракти можуть використовуватися для документування поведінки програмних компонентів?

24. Що означає порушення контракту?

25. Які основні причини порушення передумов, постумов та інваріантів?

26. Чим порушення контракту відрізняється від штатної виняткової ситуації?

27. Як використання контрактів сприяє підвищенню надійності програмного забезпечення?

Теми проєктування за контрактом і наслідків його порушення безпосередньо передбачені лекційною частиною дисципліни.

Успадкування та поліморфізм

28. Що таке успадкування в об’єктно-орієнтованому програмуванні?

29. Які переваги та обмеження має використання успадкування?

30. Що означає відношення is-a між класами?

31. Чим відношення is-a відрізняється від has-a?

32. У яких випадках композиція є доцільнішою за успадкування?

33. Чому не рекомендується використовувати успадкування лише заради повторного використання готового програмного коду?

34. Що таке поліморфізм і як він пов’язаний з успадкуванням?

35. Які проблеми можуть виникати внаслідок надмірно глибокої ієрархії успадкування?

36. За якими ознаками можна визначити, що успадкування між двома класами застосовано неправильно?

37. Як зміни в базовому класі можуть впливати на його нащадків?

Правильному використанню успадкування в робочій програмі присвячено кілька окремих тем лекційного курсу.

Визначення класів та об’єктна декомпозиція

38. У чому полягає різниця між класом і об’єктом?

39. Які характеристики вказують на доцільність виділення поняття в окремий клас?

40. Що таке відповідальність класу?

41. Чому назва класу повинна відображати певну абстракцію, а не окрему дію?

42. У чому полягає метод аналізу «іменник / дієслово» для пошуку потенційних класів?

43. Чому автоматичне перетворення кожного іменника у вимогах на клас є неправильним підходом?

44. Що таке приховані класи та чому вони можуть не бути явно зазначені у вимогах?

45. Які ознаки свідчать про те, що клас має надмірну кількість відповідальностей?

46. Що таке «God Object» і чому його поява є небажаною?

47. Які проблеми можуть виникати в класах, що містять лише дані без відповідної поведінки?

48. Як оцінити внутрішню зчепленість і зовнішню зв’язність класу?

У наданому матеріалі окремо наголошено, що клас аналізу повинен представляти чітку абстракцію предметної області, мати змістовні обов’язки, високу внутрішню зчепленість і низьку зв’язність. Метод «іменник / дієслово» розглядається як допоміжний спосіб пошуку класів і їх операцій.

Глобальні об’єкти та константи

49. Що таке глобально доступний програмний об’єкт?

50. У чому полягає різниця між глобальним доступом і глобальним змінним станом?

51. Що таке символічна константа та які переваги вона має порівняно з безпосереднім використанням числового або рядкового значення?

52. Коли доцільно об’єднувати пов’язані константи в окремий клас або модуль?

53. Які проблеми можуть спричиняти глобальні змінні?

54. Як глобальний стан впливає на тестованість і модульність програмного забезпечення?

55. Чому спільний об’єкт не обов’язково є незмінним?

56. Які підходи можна використовувати для контрольованої одноразової ініціалізації спільного ресурсу?

Тема глобальних об’єктів і констант є завершальною темою лекційного курсу.

Інструментальні засоби конструювання програмного забезпечення

57. Для чого використовуються системи контролю версій?

58. У чому полягають основні переваги Git як розподіленої системи контролю версій?

59. Що таке репозиторій, коміт, гілка та злиття змін?

60. Для чого виконується аналіз модульної структури існуючого програмного проєкту?

61. Які засоби сучасних IDE підтримують автоматизований рефакторинг?

62. Що таке статичний аналіз програмного коду?

63. Які типи проблем можуть бути виявлені засобами статичного аналізу?

64. Що таке cyclomatic complexity та code smell?

65. Яким чином інструменти аналізу програмного коду можуть сприяти

підвищенню його підтримуваності?

66. Які переваги дає автоматизоване тестування програмних компонентів?

67. Для чого використовуються засоби моделювання програмних систем?

68. Яку роль відіграють системи контролю версій під час командної розробки програмного забезпечення?

Лабораторний курс безпосередньо передбачає роботу із системами контролю версій, аналіз модульної структури open-source проєкту, розроблення бібліотеки АТД та багатопотоковість.

Об’єктно-орієнтовані підходи та сучасна організація розробки

69. У чому полягає роль об’єктно-орієнтованого підходу при конструюванні сучасних програмних систем?

70. Що таке Agile та які його базові принципи?

71. Яке призначення Scrum у процесі організації командної розробки?

72. Яке призначення Kanban та чим він відрізняється від Scrum?

73. Як гнучкі методи організації розробки співвідносяться з технічними принципами об’єктно-орієнтованого проєктування?

74. Чому процес організації розробки не замінює принципів правильної модульної та об’єктної декомпозиції?

75. Як контроль версій, автоматизоване тестування та безперервна інтеграція сприяють підтримці якості програмного забезпечення?

Робоча програма включає окрему тему щодо Agile, Scrum і Kanban та їх впливу на командну розробку програмного забезпечення.

РЕКОМЕНДОВАНІ ІНСТРУМЕНТАРІЇ ТА ДОДАТКОВІ РЕСУРСИ

Для виконання завдань самостійної роботи з дисципліни «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення» можуть використовуватися сучасні засоби розробки, контролю версій, моделювання, аналізу якості програмного коду, тестування та документування програмних систем. Вибір конкретного інструментарію залежить від поставленого завдання, мови програмування та технологічного стеку програмного проєкту.

Програмне забезпечення для моделювання та проєктування

- **Visual Paradigm, StarUML, draw.io (diagrams.net)** — засоби для створення UML-діаграм класів, послідовностей, компонентів, розгортання та інших моделей програмних систем.
- **PlantUML або Mermaid** — інструменти текстового опису діаграм, які дозволяють зберігати моделі разом із програмним кодом у системі контролю версій.
- **CASE-засоби** можуть використовуватися для аналізу структури програмної системи, представлення взаємозв’язків між класами та компонентами і документування прийнятих проєктних рішень.

У межах дисципліни моделювання розглядається передусім як допоміжний засіб для аналізу й представлення об’єктної та модульної структури програмного забезпечення. Методичні вказівки до курсового проєкту рекомендують, зокрема, draw.io, StarUML і Visual Paradigm для роботи з програмними моделями.

Середовища розробки (IDE)

Для виконання практичних завдань і самостійної роботи можуть використовуватися сучасні інтегровані середовища розробки:

- **Visual Studio** — для розроблення програмного забезпечення, зокрема мовою C# та на платформі .NET.
- **Visual Studio Code** — універсальне середовище для роботи з різними мовами програмування та технологічними стеками.
- **IntelliJ IDEA** — для розроблення об’єктно-орієнтованих програмних систем на Java та інших мовах JVM.
- **PyCharm** — для розроблення програмного забезпечення мовою Python.
- Інші IDE та редактори можуть використовуватися залежно від мови програмування і специфіки завдання.

Методичне забезпечення дисципліни передбачає можливість використання Visual Studio, Visual Studio Code та IntelliJ IDEA як сучасних засобів реалізації й супроводу програмного забезпечення.

Особливу увагу під час самостійної роботи рекомендується приділяти можливостям IDE **щодо автоматизованого рефакторингу, навігації між класами, аналізу залежностей, пошуку використань, автоматичного генерування фрагментів коду та інтеграції із системами контролю версій.**

Системи контролю версій та платформи спільної розробки

- **Git** — основний засіб контролю версій програмного коду та інших артефактів програмного проєкту.
- **GitHub та GitLab** — платформи для зберігання віддалених репозиторіїв, організації командної роботи, перегляду змін і супроводу програмних проєктів.

Студентам необхідно вміти виконувати базові операції з репозиторієм: створення комітів, роботу з гілками, злиття змін, перегляд історії та розв'язання конфліктів.

Робота із сучасними системами контролю версій безпосередньо передбачена лабораторною складовою дисципліни.

Інструменти аналізу якості програмного коду

Для поглибленого аналізу структури та якості програмного забезпечення можуть використовуватися:

- **SonarQube** — аналіз якості програмного коду, технічного боргу, дублювання, складності та потенційних проблем;
- **ESLint** — статичний аналіз програмного коду JavaScript/TypeScript;
- **Pylint** — аналіз якості програмного коду Python;

вбудовані аналізатори та засоби інспекції сучасних IDE.

Застосування подібних інструментів дозволяє виявляти потенційні дефекти, code smells, надмірну складність, дублювання коду та порушення прийнятих правил програмування. SonarQube, ESLint і Pylint безпосередньо наведені в методичних матеріалах дисципліни як приклади інструментів аналізу якості коду.

Інструменти тестування та перевірки програмних компонентів

Під час самостійної роботи рекомендується застосовувати засоби модульного та інтеграційного тестування, що відповідають обраній мові програмування та технологічному стеку.

Особливу увагу слід приділяти:

- модульному тестуванню окремих класів і компонентів;
- автоматизації повторного виконання тестів;
- перевірці граничних умов;
- тестуванню контрактів і виняткових ситуацій;
- перевірці поведінки програмного забезпечення після рефакторингу;
- порівнянню результатів до і після внесення структурних змін.

Це узгоджується з програмним результатом РН11, який передбачає забезпечення якості програмного забезпечення із використанням методів оцінювання, автоматизованого тестування та верифікації.

Засоби роботи з абстрактними типами даних і універсальними компонентами

Для практичного опрацювання тем абстрактних типів даних, універсалізації та повторного використання можуть застосовуватися стандартні засоби сучасних

об’єктно-орієнтованих мов програмування:

- **generics** у C#, Java та інших мовах;
- **templates** у C++;
- стандартні бібліотеки контейнерів і колекцій;
- власні параметризовані класи та бібліотеки абстрактних типів даних.

Окрема лабораторна робота дисципліни передбачає розроблення бібліотеки абстрактних типів даних.

Інструменти для багатопотокового програмування та аналізу виконання

Для опрацювання питань паралельного виконання та управління ресурсами використовуються засоби відповідної мови програмування й середовища розробки:

- потоки та задачі;
- механізми синхронізації;
- асинхронне програмування;
- засоби налагодження багатопотокових програм;
- профайлери та засоби аналізу використання пам’яті й ресурсів.

Практичне застосування багатопотоковості під час роботи з графічними об’єктами передбачено лабораторною програмою дисципліни.

Інструменти на базі штучного інтелекту

Під час самостійної роботи допускається використання сучасних AI-асистентів як допоміжних засобів для:

- пояснення незнайомих конструкцій програмного коду;
- пошуку можливих варіантів рефакторингу;
- аналізу структури класів і залежностей;
- створення початкових варіантів тестів;
- генерування шаблонного програмного коду;
- підготовки технічної документації;
- порівняння альтернативних способів реалізації.

Використання AI-інструментів не повинно замінювати самостійне виконання завдання. Студент повинен розуміти створений програмний код, уміти пояснити прийняті рішення, перевірити правильність отриманого результату та дотримуватися вимог академічної доброчесності.

Сучасні технології та підходи до конструювання програмного забезпечення

Під час самостійного опрацювання матеріалу доцільно розглядати класичні принципи об’єктного конструювання у зв’язку із сучасними практиками програмної інженерії:

SOLID — принципи організації відповідальностей, залежностей та розширення об’єктно-орієнтованих програмних систем;

Design Patterns — повторне використання перевірених проєктних рішень;

Clean Code — принципи створення зрозумілого та підтримуваного

програмного коду;

Refactoring — покращення внутрішньої структури програмного забезпечення без зміни його зовнішньої поведінки;

Clean Architecture та модульні архітектури — організація залежностей і відповідальностей великих програмних систем;

Continuous Integration — автоматизована перевірка змін під час розвитку програмного проєкту;

Code Review — колективний аналіз програмного коду як засіб контролю його якості.

У методичних вказівках до курсового проєкту передбачається використання принципів ООР, OOAD, SOLID, шаблонів проєктування, Clean Architecture, модульних і сервісних підходів, а також сучасних інструментів контролю й аналізу програмного коду.

Обчислювальні ресурси

Для виконання самостійної роботи достатньо **персонального комп'ютера або ноутбука**, характеристики якого забезпечують роботу обраного середовища розробки, засобів моделювання, тестування та аналізу програмного коду.

Необхідним є стабільний доступ до мережі Інтернет для:

- роботи з віддаленими Git-репозиторіями;
- отримання офіційної технічної документації;
- використання електронних навчальних матеріалів;
- встановлення необхідних бібліотек і пакетів;
- роботи з хмарними сервісами та онлайн-інструментами.

Таким чином, інструментальне забезпечення дисципліни повинно насамперед підтримувати **аналіз структури програмного забезпечення, роботу з класами та компонентами, повторне використання, рефакторинг, тестування, контроль якості та керування еволюцією програмного коду,** а конкретний набір програмних засобів студент обирає відповідно до завдання та використовуваного технологічного стеку.

ВИСНОВКИ

Упродовж перших років роботи в ІТ-індустрії розробники часто зосереджуються переважно на реалізації окремих функцій і написанні програмного коду. Зі зростанням складності програмних систем стає очевидним, що якість програмного продукту значною мірою залежить не лише від коректності окремих фрагментів коду, а й від того, наскільки вдало визначено його модульну структуру, розподілено відповідальності між класами, організовано залежності та передбачено можливість подальшого розвитку й супроводу системи.

Саме тому дисципліна «*Об'єктні технології та інструментальні засоби конструювання програмного забезпечення*» зосереджена на фундаментальних принципах побудови якісного програмного забезпечення та їх практичному застосуванні. У межах курсу розглядаються питання якості, модульності, повторного використання, абстрактних типів даних, управління пам'яттю, універсалізації, проєктування за контрактом, успадкування, визначення класів та використання глобальних об'єктів і констант.

У процесі вивчення дисципліни та виконання лабораторних і самостійних завдань студенти набувають умінь:

- аналізувати якість програмного забезпечення та оцінювати вплив структури коду на його надійність, підтримуваність і можливість подальшого розвитку;
- застосовувати принципи модульності та повторного використання для побудови програмних компонентів із чітко визначеними відповідальностями;
- використовувати абстрактні типи даних та універсальні програмні компоненти для створення повторно використовуваних і типобезпечних рішень;
- застосовувати проєктування за контрактом, визначати передумови, постумови та інваріанти класів для формалізації поведінки програмних компонентів;
- обґрунтовано використовувати успадкування, композицію та поліморфізм, уникати надмірно складних або семантично некоректних ієрархій класів;
- визначати класи та їх відповідальності на основі аналізу предметної області та вимог до програмної системи;
- аналізувати доцільність використання глобальних об'єктів, констант і спільного стану в об'єктно-орієнтованих системах;
- використовувати сучасні системи контролю версій, аналізувати модульну структуру відкритих програмних проєктів, розробляти бібліотеки абстрактних типів даних та застосовувати багатопотоковість.

Важливим результатом вивчення дисципліни є формування здатності **не лише реалізовувати програмний код, а й критично аналізувати його структуру, знаходити слабкі місця та обґрунтовувати способи вдосконалення**. Такий підхід безпосередньо пов'язаний із програмними результатами навчання щодо

оцінювання методів розроблення, вибору проєктних рішень, модифікації архітектури, використання сучасних засобів розроблення та забезпечення якості програмного забезпечення.

Окреме значення має робота з існуючими програмними продуктами. Аналіз відкритого програмного коду дає можливість побачити реальні приклади як вдалих, так і проблемних рішень, дослідити залежності між компонентами, оцінити використання успадкування, модульності та повторного використання, а також сформувати практичні навички рефакторингу й удосконалення програмної структури. Такий підхід продовжується і під час виконання курсового проєкту, де основна увага приділяється аналізу існуючого програмного продукту, проєктуванню покращень, їх реалізації та оцінюванню отриманих результатів.

Знання та навички, отримані під час вивчення дисципліни, формують основу професійного інженерного мислення. Вони дають можливість перейти від сприйняття програмного забезпечення як сукупності окремих функцій і фрагментів коду до розуміння його як **системи взаємопов'язаних абстракцій, класів, модулів і компонентів**, які повинні бути якісно спроєктовані, зрозумілі, повторно використовувані та придатні до подальшого розвитку.

Саме здатність приймати обґрунтовані рішення щодо структури програмного забезпечення, відповідальностей класів, механізмів повторного використання, контрактів, успадкування та інструментальних засобів є важливою передумовою професійного зростання інженера програмного забезпечення та його переходу від реалізації окремих програмних задач до проєктування й удосконалення складних програмних систем.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

Базова література:

1. Meyer B. Object-Oriented Software Construction. 2nd ed. Prentice Hall, 1997.
2. Куєвда Ю. В. Об’єктно-орієнтоване програмування. Курс лекцій: навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 164 с.
3. Порєв В. М. Об'єктно-орієнтоване програмування. Конспект лекцій: навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 271 с.
4. Мнушка О. В., Савченко В. М., Маций О. Б. Об’єктно-орієнтоване програмування мовою Python: навчальний посібник. Харків : ХНАДУ, 2021. 228 с.
5. Любченко Н. Ю., Соболь М. О., Пугачов Р. В. та ін. Основи ООП C++ в прикладах. Частина 1: навчально-методичний посібник. Харків : НТУ «ХПІ», 2024. 223 с.
6. Sommerville I. Software Engineering. 10th ed. Pearson, 2022. Видання охоплює, зокрема, проєктування та реалізацію, повторне використання, компонентну розробку, тестування й еволюцію програмного забезпечення.
7. Freeman E., Robson E. Head First Design Patterns. 2nd ed. O’Reilly Media, 2020. 672 p.

Допоміжна література:

8. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media, 2020. 432 p. Видання містить матеріал щодо модульності, зв’язності, зчеплення, компонентів та архітектурних рішень.
9. Farley D. Modern Software Engineering: Doing What Works to Build Better Software Faster. Addison-Wesley Professional, 2021. 256 p.
10. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Prentice Hall, 2017.
11. Куєвда Ю. В. Об’єктно-орієнтоване програмування. Лабораторний практикум: навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 68 с. Посібник містить практичні завдання з UML, виняткових ситуацій, тестування об’єктно-орієнтованого коду та патернів проєктування.
12. Sommerville I. Engineering Software Products: An Introduction to Modern Software Engineering. Pearson, 2020. Видання охоплює сучасне конструювання програмних продуктів, надійне програмування, тестування, керування кодом та DevOps.
13. Washizaki H. (ed.). Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0. IEEE Computer Society, 2024. Для актуального електронного використання IEEE Computer Society надає оновлену редакцію SWEBOK V4.0a, позначену як актуальне оновлення версії 4.
14. ISO/IEC 25010:2023. Systems and software engineering — Systems and

software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Стандарт визначає актуальну модель якості програмного та ІКТ-продукту.

15. OMG Unified Modeling Language (OMG UML), Version 2.5.1. Object Management Group. UML у межах дисципліни доцільно використовувати як допоміжний засіб представлення класів, зв'язків, компонентів і взаємодій програмної системи.

16. Schwaber K., Sutherland J. The Scrum Guide. 2020. Офіційною чинною редакцією Scrum Guide залишається версія листопада 2020 року.

17. Coleman J., Vacanti D., Johnson C. et al. The Kanban Guide. May 2025.

Інформаційні ресурси в мережі Інтернет:

18. IEEE Computer Society — Software Engineering Body of Knowledge (SWEBOK Guide). Офіційний ресурс для опрацювання сучасної систематизації знань із програмної інженерії, зокрема питань конструювання, проєктування, архітектури, якості та тестування програмного забезпечення. Актуальна версія SWEBOK V4 містить 18 областей знань. [Електронний ресурс] — Режим доступу: https://www.computer.org/education/bodies-of-knowledge/software-engineering

19. Microsoft Learn — C# Documentation. Офіційна документація Microsoft з мови C# та платформи .NET. Рекомендується для опрацювання класів, об'єктів, успадкування, поліморфізму, інтерфейсів, універсальних типів та інших засобів об'єктно-орієнтованого програмування. [Електронний ресурс] — Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/

20. Oracle Java Documentation. Офіційна документація та навчальні матеріали з мови Java, зокрема щодо класів, успадкування, інтерфейсів, універсальних типів та параметризації типів. [Електронний ресурс] — Режим доступу: https://docs.oracle.com/en/java/

21. Git Documentation. Офіційна документація системи контролю версій Git. Містить матеріали щодо створення та клонування репозиторіїв, комітів, гілок, злиття змін, роботи з віддаленими репозиторіями та інших операцій керування версіями. [Електронний ресурс] — Режим доступу: https://git-scm.com/docs

22. GitHub Docs. Офіційна документація платформи GitHub щодо роботи з репозиторіями, гілками, pull request, командної розробки, GitHub Actions та інших засобів підтримки програмних проєктів. [Електронний ресурс] — Режим доступу: https://docs.github.com/

23. GitLab Documentation. Офіційна документація GitLab щодо репозиторіїв, системи Git, CI/CD, керування програмним кодом, тестування та автоматизації процесів розробки.[Електронний ресурс] — Режим доступу: https://docs.gitlab.com/

24. JetBrains Documentation. Офіційна документація інтегрованих середовищ розробки JetBrains, зокрема IntelliJ IDEA, PyCharm та інших IDE. Містить матеріали щодо навігації по коду, рефакторингу, статичного аналізу,

налагодження та роботи із системами контролю версій. [Електронний ресурс] — Режим доступу: https://www.jetbrains.com/help/

25. Sonar Documentation. Офіційна документація засобів Sonar для статичного аналізу та контролю якості програмного коду, виявлення дефектів, code smells, проблем підтримуваності та інших характеристик якості. [Електронний ресурс] — Режим доступу: https://docs.sonarsource.com/