МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Кафедра програмної інженерії

Конспект лекцій

з дисципліни

«Раціональний уніфікований процес проєктування програмного забезпечення»

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

Тернопіль — 2026

УДК 004.41(076.5)

Укладачі:
І.Я. Мудрик, PhD

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.

Конспект лекцій з дисципліни «Раціональний уніфікований процес проєктування програмного забезпечення» для здобувачів другого (магістерського) рівня вищої освіти ОПП «Інженерія програмного забезпечення» всіх форм навчання / Укладач: Мудрик І.Я.,– Тернопіль: ТНТУ ім. І. Пулюя, 2026 – 110 с.

Відповідальний за випуск: І.Я. Мудрик, PhD

АНОТАЦІЯ

Пропонований конспект лекцій призначений для здобувачів вищої освіти спеціальності 121/F2 «Інженерія програмного забезпечення» і містить систематизований виклад теоретичного матеріалу з дисципліни «Раціональний уніфікований процес проєктування програмного забезпечення» (RUP).

Головною особливістю видання є поєднання фундаментальних принципів класичної ітеративної розробки із сучасними інженерними реаліями. Матеріал структуровано таким чином, щоб продемонструвати логічну еволюцію процесів життєвого циклу ПЗ: від базових фаз, робочих дисциплін та UML-проєктування в RUP до їх трансформації у гнучкі методології (Agile), практики безперервного розгортання (DevOps), мікросервісну архітектуру та застосування методів AI Engineering.

Вивчення матеріалів курсу спрямоване на формування у майбутніх інженерів програмного забезпечення комплексного розуміння архітектурних рішень та навичок управління проєктами, а також слугує теоретичним підґрунтям для:

- **Опрацювання лекційного матеріалу** та поглибленого вивчення сучасної проєктної документації.
- **Підготовки до виконання лабораторних і практичних робіт** (зокрема, збору вимог, побудови архітектурних моделей, налаштування CASE-засобів та пайплайнів).
- **Організації самостійної роботи студента (СРС)**, що включає дослідження новітніх інструментів автоматизації розробки та адаптації класичних процесів під потреби реальних команд.
- **Підготовки до модульного та підсумкового контролю** знань з дисципліни.

Видання орієнтоване на підготовку висококваліфікованих фахівців, здатних проєктувати та супроводжувати програмні системи з урахуванням найсучасніших галузевих стандартів.

ЗМІСТ

ТЕМА 5. ТРАНСФОРМАЦІЯ КЛАСИЧНИХ ПРОЦЕСІВ У СУЧАСНІ ІНЖЕНЕРНІ ПРАКТИКИ

ВСТУП

Сучасна інженерія програмного забезпечення — це стрімка та високотехнологічна галузь, у якій успіх проєкту залежить не лише від обраного стека технологій чи кваліфікації окремих розробників, але й від правильної організації процесів життєвого циклу створення програмного продукту.

Дисципліна «Раціональний уніфікований процес проєктування програмного забезпечення» (Rational Unified Process, RUP) є однією з фундаментальних складових підготовки фахівців за спеціальністю 121 «Інженерія програмного забезпечення». RUP — це не просто набір жорстких правил, це комплексна методологічна база, яка акумулювала найкращі практики індустрії щодо управління ризиками, архітектурного проєктування та ітеративної розробки.

Мета вивчення дисципліни полягає у формуванні у студентів системного мислення та стійких практичних навичок щодо організації процесу розробки ПЗ на основі ітеративного та інкрементного підходів, керованих прецедентами та орієнтованих на архітектуру.

Незважаючи на те, що сьогодні індустрія масово використовує гнучкі методології (Agile, Scrum, Kanban), розуміння класичного RUP є критично важливим. Більшість сучасних практик не виникли з нічого — вони є еволюційним розвитком або свідомим спрощенням класичних інженерних процесів. Цей конспект лекцій побудований таким чином, щоб продемонструвати цей еволюційний шлях.

Матеріал курсу охоплює як класичну теорію (фази Inception, Elaboration, Construction, Transition та основні робочі дисципліни), так і сучасні тенденції. Студенти дізнаються, як традиційна дисципліна розгортання та управління конфігураціями трансформувалася у сучасну культуру DevOps, як монолітні архітектури поступилися місцем мікросервісам, і як практики AI Engineering поступово інтегруються в кожен етап життєвого циклу, змінюючи підходи до написання коду та тестування.

Основні завдання курсу:

- Ознайомити студентів із життєвим циклом розробки ПЗ за методологією RUP.
- Навчити виявляти, специфікувати та управляти вимогами за допомогою моделі прецедентів (Use Cases).
- Сформувати навички проєктування архітектури та виділення ключових підсистем з використанням нотації UML.
- Продемонструвати принципи управління конфігураціями, змінами та ризиками в проєкті.

- Прослідкувати трансформацію класичних важковагових процесів у сучасні методології розробки (Agile) та інженерні практики безперервної інтеграції і доставки (CI/CD).

Опанувавши матеріали цього курсу, здобувачі вищої освіти отримають необхідний інструментарій для свідомого вибору методології розробки, проєктування масштабованої архітектури та ефективної взаємодії в межах крос-функціональних команд при створенні програмних систем будь-якого рівня складності.

Конспект містить 24 лекції, згрупований за п'ятьма ключовими темами:

Тема 1. Вступ до RUP. Основні принципи та концепції

- **Лекція 1. Еволюція методологій розробки ПЗ та місце RUP.** Історичний контекст, криза розробки ПЗ. Порівняння предиктивних (каскадних) та адаптивних підходів. Передумови створення Rational Unified Process.
- **Лекція 2. Фундаментальні принципи RUP.** Детальний розгляд трьох основ: Use-case driven (керованість прецедентами), Architecture-centric (орієнтованість на архітектуру), Iterative and incremental (ітеративність та інкрементність).
- **Лекція 3. Архітектура процесу RUP.** Двовимірна модель процесів. Взаємозв'язок між динамічним виміром (час, фази, ітерації) та статичним виміром (робочі процеси, дисципліни, ролі, артефакти).

Тема 2. Динамічна структура: Фази життєвого циклу

- **Лекція 4. Фаза Inception (Початковий етап).** Визначення меж системи. Формування бачення проекту (Vision). Аналіз зацікавлених сторін та економічне обґрунтування (Business Case).
- **Лекція 5. Фаза Elaboration (Проєктування).** Ідентифікація та усунення ключових технічних ризиків. Створення базової виконуваної архітектури (Executable Architecture).
- **Лекція 6. Фаза Construction (Побудова).** Ітеративна розробка функціоналу. Оптимізація ресурсів, управління кодоскопом, паралельна розробка компонентів.
- **Лекція 7. Фаза Transition (Впровадження).** Підготовка до релізу. Бета-тестування, виправлення дефектів пізніх стадій, розгортання в робочому середовищі та навчання користувачів.
- **Лекція 8. Віхи (Milestones) та метрики переходу.** Критерії успішності проходження ключових віх (LCO, LCA, IOC, PR). Оцінка готовності до переходу на наступну фазу.

Тема 3. Статична структура: Основні робочі дисципліни (Engineering Disciplines)

- **Лекція 9. Бізнес-моделювання (Business Modeling).** Аналіз бізнес-процесів замовника. Створення моделей бізнес-прецедентів та бізнес-об'єктів для узгодження ПЗ з цілями організації.
- **Лекція 10. Управління вимогами (Requirements Management).** Збір, класифікація та документування вимог. FURPS+ модель якості. Специфікація програмних вимог (SRS).
- **Лекція 11. Прецеденти (Use Cases) на практиці.** Деталізація Use-case моделі. Написання сценаріїв прецедентів, альтернативні потоки, відносини include та extend.
- **Лекція 12. Аналіз та проєктування (Analysis & Design) – Частина 1.** Перехід від вимог до концептуальної моделі. Виділення класів аналізу (Boundary, Control, Entity).
- **Лекція 13. Аналіз та проєктування (Analysis & Design) – Частина 2.** Проєктування підсистем та інтерфейсів. Застосування патернів проєктування (GoF) та відображення архітектури через UML-діаграми.
- **Лекція 14. Дисципліна реалізації (Implementation).** Організація вихідного коду, стандарти кодування, збірка системи. Взаємозв'язок з архітектурною моделлю.
- **Лекція 15. Дисципліна тестування (Testing).** Рівні тестування в RUP (Unit, Integration, System). Тестування на основі прецедентів. Управління дефектами.
- **Лекція 16. Дисципліна розгортання (Deployment).** Класичні підходи до пакування ПЗ, створення інсталяторів, підготовка документації користувача та адміністратора.

Тема 4. Допоміжні дисципліни, ролі та артефакти

- **Лекція 17. Управління конфігурацією та змінами (Configuration & Change Management).** Версіонування артефактів, створення базових ліній (Baselines), обробка запитів на зміни (CR).
- **Лекція 18. Управління проектом (Project Management).** Планування фаз та окремих ітерацій. Оцінка трудовитрат, управління ризиками, план розробки ПЗ (SDP).
- **Лекція 19. Дисципліна "Середовище" (Environment).** Підготовка робочого простору. Вибір інструментарію розробки, налаштування процесів під потреби конкретної команди (Tailoring).

- **Лекція 20. Рольова модель та ключові артефакти RUP.** Детальний розбір зон відповідальності Аналітика, Архітектора, Розробника та Тестувальника. Життєвий цикл ключових документів.

Тема 5. Трансформація класичних процесів у сучасні інженерні практики

- **Лекція 21. Від важковагових процесів до Agile.** Обмеження класичного RUP. Огляд Agile Unified Process (AUP) та OpenUP як мосту до сучасних ітеративних методологій (Scrum, Kanban).
- **Лекція 22. Еволюція дисципліни розгортання та управління конфігураціями: DevOps.** Трансформація принципів RUP у сучасні пайплайни CI/CD. Роль контейнеризації (Docker) та автоматизації збірки.
- **Лекція 23. Еволюція архітектурного підходу (Architecture-centric).** Перехід від RUP-монолітів до Service-Oriented Architecture (SOA) та сучасних мікросервісних ландшафтів.
- **Лекція 24. Майбутнє інженерії програмного забезпечення.** Інтеграція підходів AI Engineering у життєвий цикл розробки. Автоматизація генерації коду, тестування та аналізу вимог за допомогою штучного інтелекту.

ТЕМАТИЧНИЙ ПЛАН НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Форма контролю (двосеместрово): залік та екзамен.

Обсяг: 8.0 ECTS (240 годин): 84 год. аудиторних (48 лекції, 48 практичні), 156 год. самостійної роботи (з них – виконання курсових проектів24 год.).

Метою та завданням курсу є розвинення у студентів навичок та підходів щодо використання перевірених архітектурних принципів і шаблонів проектування для створення ефективних високоякісних програмних комплексів в більш стислі терміни і з меншими ризиками. Можливість формування у студентів комплексу знань та практичних навичок із планування, моніторингу та управлінню проектами від локального до корпоративного рівнів з використанням глобальної методології РУП. Розглядаються принципи розробки інформаційних систем, структурної організації команди розробників системи, їх взаємодія в процесі функціонування, перспективні напрямки розвитку сучасних інформаційних технологій, із вдосконалення навичок використання методології RUP. Це направлено на ітеративну модель розробки, а після закінчення кожного етапу і кожної ітерації створювати всі необхідні документи і досягти максимального рівня формалізації.

Мета проведення лекційних занять полягає в ознайомленні студентів з теоретичними основами проектування ПЗ та архітектурних рішень, ознайомлення з шаблонами проектування, середовищами проєктування архітектур та моделей.

Дослідження існуючих архітектур проміжного програмного забезпечення згідно основних підходів RUP.

Мета проведення лабораторних занять полягає у тому, щоб виробити у студентів практичні навички застосування уніфікованих методів та підходів до проектування ПЗ згідно загальноприйнятих правил методології RUP та розробки архітектури програмного забезпечення у відповідності до проєктованих рішень.

При цьому завданням курсу є ознайомити студентів з процесом організації проектування та розробки ПЗ, розвинути навики організації праці в команді, допомогти майбутнім фахівцям оволодіти знаннями менеджменту розробки ПЗ з погляду процесу RUP.

2.2. За результатами вивчення дисципліни «Раціональний уніфікований процес проектування програмного забезпечення» студент повинен продемонструвати такі результати навчання (***кінцеві, підсумкові та інтегративні результати навчання, що визначають нормативний зміст підготовки):***

За результатами вивчення дисципліни «Раціональний уніфікований процес проектування програмного забезпечення» студент повинен продемонструвати такі **результати навчання**:

РН01 Знати і застосовувати сучасні професійні стандарти і інші нормативно-правові документи з інженерії програмного забезпечення

РН02 Оцінювати і вибирати ефективні методи і моделі розроблення, впровадження, супроводу програмного забезпечення та управління відповідними процесами на всіх етапах життєвого циклу.

РН03 Будувати і досліджувати моделі інформаційних процесів у прикладній області.

РН06 Розробляти і оцінювати стратегії проєктування програмних засобів; обґрунтовувати, аналізувати і оцінювати варіанти проектних рішень з точки зору якості кінцевого програмного продукту, ресурсних обмежень та інших факторів.

РН08 Розробляти і модифікувати архітектуру програмного забезпечення для реалізації вимог замовника.

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.

РН17 Збирати, аналізувати, оцінювати необхідну для розв’язання наукових і прикладних задач інформацію, використовуючи науково-технічну літературу, бази даних та інші джерела.

Вивчення навчальної дисципліни передбачає формування та розвиток у студентів **компетентностей:**

інтегральна компетентність:

- Здатність особи розв'язувати складні задачі і проблеми у певній галузі професійної діяльності або у процесі навчання, що передбачає проведення досліджень та/або здійснення інновацій та характеризується невизначеністю умов і вимог.

загальні компетентності:

- ЗК01. Здатність до абстрактного мислення, аналізу та синтезу
- ЗК05. Здатність генерувати нові ідеї (креативність). 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.
- СК05. Здатність розробляти, аналізувати та застосовувати специфікації, стандарти, правила і рекомендації в сфері інженерії програмного забезпечення.
- СК09. Здатність забезпечувати якість програмного забезпечення.
- СК10*. Здатність розробляти інноваційне програмне забезпечення для систем підтримки прийняття рішень.
- СК* - спеціальна компетентність, запропонована стейкхолдерами.

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;

Уміння: Ум1, Ум2. Спеціалізовані уміння/навички розв'язання проблем, необхідні для проведення досліджень та/або провадження інноваційної діяльності з метою розвитку нових знань та процедур. Здатність інтегрувати знання та розв'язувати складні задачі у широких або мультидисциплінарних контекстах;

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.

Тема 1. Вступ до RUP. Основні принципи та концепції

Ця тема закладає теоретичний фундамент для розуміння того, як індустрія інженерії програмного забезпечення перейшла від стихійного написання коду до систематизованих, керованих процесів. Ми розглянемо історичний контекст, що призвів до створення Rational Unified Process (RUP), та визначимо його місце серед інших методологій.

Лекція 1. Еволюція методологій розробки ПЗ та місце RUP.

- Історичний контекст, криза розробки ПЗ.
- Порівняння предиктивних (каскадних) та адаптивних підходів.
- Передумови створення Rational Unified Process.

Мета лекції: Ознайомитися з історією розвитку процесів створення програмного забезпечення, проаналізувати класичні моделі життєвого циклу та зрозуміти передумови появи методології Rational Unified Process.

Проблема масштабування та «Криза програмного забезпечення»

На зорі розвитку обчислювальної техніки (у 1950-х – на початку 1960-х років) програмування розглядалося радше як мистецтво, ніж як інженерна дисципліна. Домінував підхід **Code and Fix** (кодуй та виправляй). Розробники одразу бралися за написання коду без попереднього проєктування, а помилки виправлялися в міру їх виявлення.

Зі стрімким зростанням продуктивності апаратного забезпечення та збільшенням складності завдань цей підхід зазнав краху. Індустрія зіткнулася з явищем, яке у 1968 році на конференції НАТО отримало назву **«Криза програмного забезпечення» (Software Crisis)**.

Ознаки кризи:

- Бюджети проєктів систематично перевищувалися у кілька разів.
- Зривалися терміни випуску продуктів.
- Готове ПЗ не відповідало вимогам замовника.
- Код був настільки заплутаним, що його супровід та модернізація ставали неможливими або економічно недоцільними.

Виникла гостра потреба у застосуванні суворих інженерних підходів до розробки ПЗ — переходу від ремісництва до промислового виробництва. Це стало поштовхом до створення моделей життєвого циклу програмного забезпечення (Software Development Life Cycle, SDLC).

Еволюція моделей життєвого циклу

Щоб зрозуміти місце RUP, необхідно розглянути його попередників, оскільки RUP акумулював їхні сильні сторони та намагався нівелювати недоліки.

Каскадна модель (Waterfall Model) Запропонована Вінстоном Ройсом у 1970 році, вона стала першим формалізованим процесом. Каскадна модель передбачає послідовне виконання етапів: Збір вимог → Проєктування → Реалізація (кодування) → Тестування → Розгортання → Супровід.

- **Переваги:** Чітка структура, зрозуміла документація на кожному етапі, зручність для планування бюджету.
- **Недоліки:** Жорсткість. Змінити вимоги на етапі кодування або тестування надзвичайно дорого і складно. Робочий продукт замовник бачить лише на самому кінці процесу, що створює колосальний ризик випустити непотрібну систему.

Спіральна модель (Spiral Model) У 1986 році Баррі Боем представив спіральну модель, яка кардинально змінила фокус розробки, змістивши його на **управління ризиками**. Розробка відбувається витками спіралі, кожен з яких включає аналіз ризиків, прототипування та оцінку результатів.

- **Переваги:** Раннє виявлення критичних проблем (архітектурних, технологічних, фінансових) через створення прототипів.
- **Недоліки:** Висока складність управління процесом. Модель вимагала висококваліфікованих експертів з оцінки ризиків і погано підходила для малих проєктів.

Ітеративний та інкрементний підхід Цей підхід став логічним розвитком спіральної моделі. Замість того, щоб намагатися зробити все і відразу (як у Waterfall), система розробляється невеликими частинами — інкрементами, через повторювані цикли — ітерації. Саме цей принцип став однією з головних опор майбутнього RUP.

«Війни методів» та народження RUP

У 1990-х роках індустрію охопив бум об'єктно-орієнтованого програмування (ООП). ООП вимагало нових підходів до проєктування та аналізу. З'явилося понад 50 різних методологій та мов моделювання. Цей період отримав назву «Війн методів» (Method Wars).

Найбільш впливовими фахівцями того часу були «Три Аміго» (The Three Amigos):

1. **Граді Буч (Grady Booch)** — автор методу проєктування, орієнтованого на архітектуру та компоненти.
2. **Джеймс Рамбо (James Rumbaugh)** — творець методу OMT (Object Modeling Technique), сильного в аналізі предметної області.
3. **Івар Якобсон (Ivar Jacobson)** — розробник методу OOSE (Object-Oriented Software Engineering), який вперше запровадив концепцію Use Cases (прецедентів) для управління вимогами.

У середині 90-х років ці троє дослідників об'єднали свої зусилля в компанії Rational Software. Їхня співпраця дала два фундаментальні результати для інженерії ПЗ:

- Створення **UML (Unified Modeling Language)** — єдиної стандартизованої графічної мови для моделювання систем.
- Створення **Rational Unified Process (RUP)** — всеосяжної методології, яка описувала, *як саме* використовувати UML та організовувати роботу команди. У 2003 році компанію Rational Software придбала корпорація IBM, продовживши розвиток методології.

Місце RUP у сучасній інженерії програмного забезпечення

RUP позиціонується не просто як жорсткий алгоритм дій, а як **фреймворк процесу (Process Framework).** Це означає, що він містить величезну базу знань, шаблонів, ролей та артефактів, з яких команда повинна вибрати лише ті, що необхідні для конкретного проєкту.

Де знаходиться RUP на шкалі методологій? Якщо уявити спектр методологій розробки, де на одному полюсі знаходиться суворий і неповоротний Waterfall, а на іншому — надзвичайно гнучкий, але часто хаотичний стихійний підхід, то RUP займає збалансовану позицію ближче до центру.

RUP став перехідною ланкою між жорсткими процесами минулого та гнучкими практиками сьогодення:

- Він запровадив ітеративність, яка сьогодні є основою Scrum.
- Він легалізував важливість постійного тестування та інтеграції протягом усього циклу, що стало ідейним попередником сучасних CI/CD конвеєрів.
- Його фокус на «архітектурі як центрі розробки» дозволяє проєктувати складні, масштабовані системи, що є критично важливим для переходу до сучасних мікросервісних та хмарних ландшафтів.

Сьогодні інтеграція принципів RUP у навчальні програми дозволяє сформувати у студентів розуміння того, чому сучасні практики DevOps або AI Engineering виглядають саме так. Вони не виникли на порожньому місці — це автоматизовані, прискорені та адаптовані під нові реалії дисципліни управління конфігураціями, середовищем та розгортанням, які вперше були системно описані саме в рамках Rational Unified Process.

Лекція 2. Фундаментальні принципи RUP.

- Детальний розгляд трьох основ:
- Use-case driven (керованість прецедентами),
- Architecture-centric (орієнтованість на архітектуру), Iterative and incremental (ітеративність та інкрементність).

Мета лекції: Детально розібрати три базові принципи (стовпи), на яких тримається вся методологія Rational Unified Process, та зрозуміти, як їхня взаємодія забезпечує створення якісного програмного продукту, що відповідає очікуванням замовника.

В основі RUP лежать три ключові концепції. Їх часто називають «трьома китами» методології. Якщо забрати хоча б один із них, процес втратить свою цілісність і перетвориться на хаотичну розробку.

Керованість прецедентами (Use-Case Driven)

Традиційні методології часто фокусувалися на функціональних вимогах (наприклад, система повинна вміти сортувати масив даних). RUP кардинально змінює цей підхід, ставлячи в центр **користувача** та те, яку цінність система йому приносить.

Що таке прецедент (Use Case)?

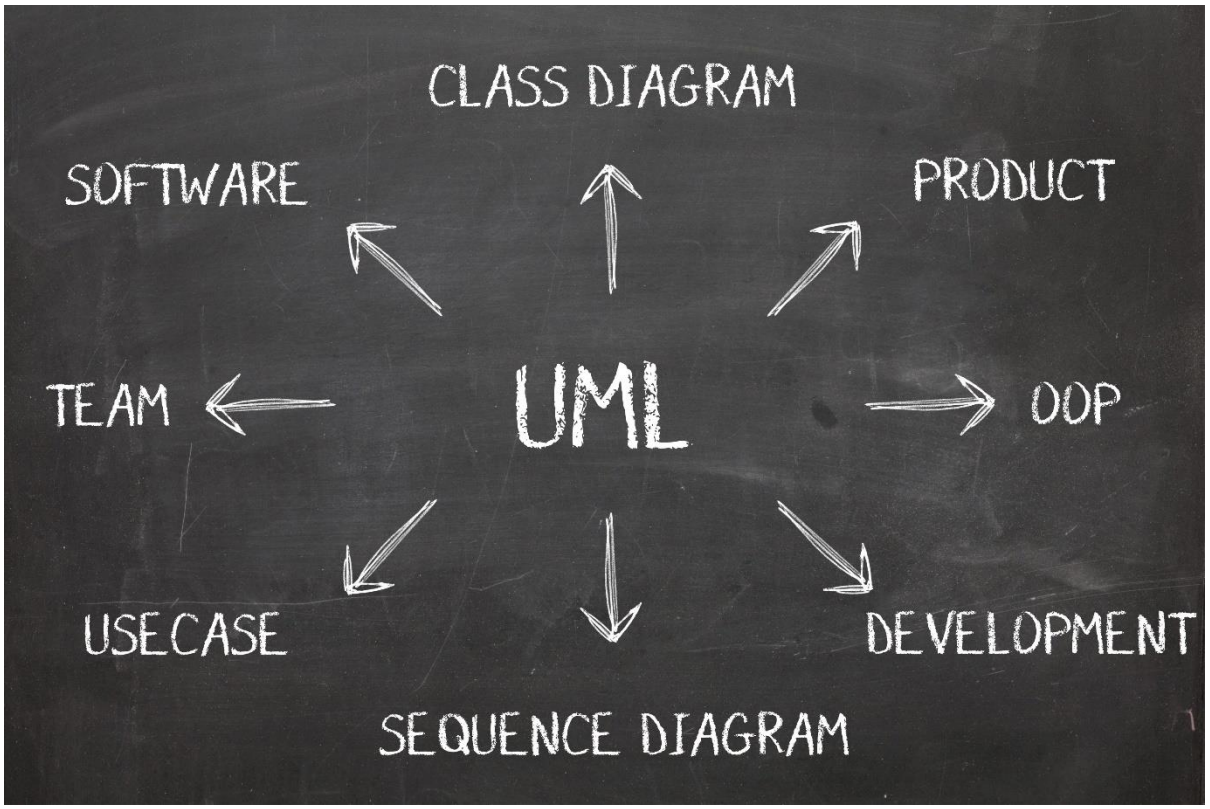
Це послідовність дій, які виконує система для того, щоб надати значущий результат певному **Актору** (користувачеві або іншій системі). Іншими словами, це сценарій використання системи.

Чому процес є «керованим» прецедентами?

Прецеденти в RUP — це не просто спосіб запису вимог. Це нитка, яка прошиває весь життєвий цикл розробки:

- **Бізнес-моделювання та Вимоги:** Аналітики спілкуються із замовником мовою прецедентів, визначаючи, *що саме* має робити система.

- **Проєктування та Архітектура:** Архітектори та розробники створюють компоненти системи, спираючись на те, як ці компоненти будуть реалізовувати конкретні прецеденти.
- **Реалізація (Кодування):** Програмісти пишуть код не для абстрактних функцій, а для виконання конкретного сценарію Use Case.
- **Тестування:** Тестувальники створюють тест-кейси безпосередньо на основі Use Cases. Якщо прецедент успішно пройдено, вимога вважається реалізованою.
- **Документування:** Інструкції користувача пишуться на основі тих самих сценаріїв.



Отже, прецеденти керують процесом створення ПЗ від першої зустрічі із замовником до фінального релізу.

Орієнтованість на архітектуру (Architecture-Centric)

Якщо прецеденти відповідають на запитання *«Що система повинна робити?»*, то архітектура відповідає на запитання *«Як система буде побудована, щоб це зробити надійно, швидко та з можливістю масштабування?»*.

У RUP архітектура не є вторинним продуктом написання коду. Вона проєктується цілеспрямовано і на ранніх етапах.

Взаємозв'язок Архітектури та Прецедентів

Ці два поняття працюють у тандемі:

1. **Прецеденти формують архітектуру:** Архітектор обирає найважливіші, критичні або найтехнологічніші прецеденти (близько 10-20% від загальної кількості) і будує базову архітектуру для їх реалізації.
2. **Архітектура обмежує прецеденти:** Щойно базова архітектура затверджена, вона диктує правила для реалізації всіх інших, менш критичних прецедентів.

Модель «4+1» (Кручтен)

Для опису архітектури в RUP використовується знаменита модель Філіпа Кручтена «4+1 Views». Архітектуру неможливо описати однією діаграмою, тому її розглядають з різних точок зору (Views):

1. **Логічний вигляд (Logical View):** Класи та їхня взаємодія (для аналітиків і розробників).
2. **Вигляд процесів (Process View):** Потоки, паралелізм, синхронізація (для системних інтеграторів).
3. **Вигляд реалізації / розробки (Development View):** Організація модулів та пакетів коду (для програмістів).
4. **Фізичний вигляд (Physical View):** Розподіл ПЗ по апаратному забезпеченню, топологія мережі (для системних адміністраторів).

- **+1. Вигляд прецедентів (Scenarios / Use Case View):** Знаходиться в центрі та пов'язує всі чотири попередні вигляди, доводячи, що архітектура дійсно вирішує поставлені бізнес-задачі.

Ітеративність та інкрементність (Iterative and Incremental)

Класичний каскадний підхід (Waterfall) передбачав, що система розробляється як єдине ціле і постачається наприкінці. RUP стверджує: розробляти великі системи за один підхід — це шлях до катастрофи.

- **Ітеративність:** Проєкт розбивається на низку міні-проєктів (ітерацій). Кожна ітерація триває фіксований час (зазвичай від 2 до 6 тижнів) і включає всі етапи життєвого циклу: збір вимог, проєктування, кодування, тестування та інтеграцію.

• **Інкрементність:** Результатом кожної успішної ітерації є *виконуваний реліз* (інкремент) — робоча частина системи. З кожною ітерацією продукт "обростає" новими функціями, наближаючись до фінального стану.

Чому це критично важливо? (Керування ризиками)

RUP є процесом, що *керований ризиками*. В ітеративній моделі найскладніші та найризикованіші завдання виконуються в перших ітераціях.

• Якщо в архітектурі є фундаментальна помилка, команда виявить її на 2-й ітерації (через місяць після старту), а не наприкінці року під час фінального тестування.

• Замовник бачить робочий продукт раніше і може коригувати вимоги (це прямий попередник сучасного Agile).

Висновки (Синергія трьох принципів)

Три фундаментальні принципи RUP не існують ізольовано. Їхня магія полягає у взаємодії:

• Ми використовуємо **ітеративний підхід**, щоб крок за кроком реалізовувати систему...
• ...кожен крок при цьому **керується прецедентами**, щоб ми створювали те, що дійсно потрібно користувачу...
• ...і вся ця розробка спирається на надійну, ранньосформовану **архітектуру,** щоб система не розвалилася під час додавання нових функцій.

Саме ця тріада зробила RUP однією з найвпливовіших методологій, принципи якої (хоч і в дещо трансформованому вигляді) ми використовуємо і сьогодні.

Лекція 3. Архітектура процесу RUP: статичний та динамічний виміри

Мета лекції: Розуміти структурну організацію методології Rational Unified Process, дослідити концепцію двовимірної моделі процесу та проаналізувати, як інженерні дисципліни розподіляються вздовж фаз життєвого циклу.

Концепція двовимірної моделі процесу

Більшість класичних методологій (наприклад, Waterfall) розглядають розробку програмного забезпечення як одновимірний процес — лінійну послідовність кроків від збору вимог до розгортання. RUP визнає, що реальна

розробка є набагато складнішою: дії виконуються паралельно, повертаються на доопрацювання і перетинаються.

Тому RUP описується за допомогою **двовимірної матричної моделі**, яка дозволяє відобразити як часову динаміку проєкту, так і логічну структуру виконуваних робіт.

- **Горизонтальна вісь (Динамічний вимір):** Відображає час. Вона показує життєвий цикл процесу, який розгортається у вигляді циклів, фаз, ітерацій та контрольних точок (віх).
- **Вертикальна вісь (Статичний вимір):** Відображає структуру процесу. Вона описує логічне групування діяльностей у робочі процеси (дисципліни), а також визначає, *хто* їх виконує, *як* і з *яким результатом*.

Динамічний вимір (Час, фази, ітерації)

Динамічний вимір відповідає на запитання: **«Коли і в якій послідовності це відбувається?»**

Життєвий цикл ПЗ у RUP розбивається на часові відрізки різного масштабу:

1. **Цикл (Cycle):** Повний прохід через усі фази, що завершується випуском нового покоління продукту (наприклад, версії 1.0, потім 2.0).
2. **Фази (Phases):** Кожен цикл складається з чотирьох послідовних фаз. Кожна фаза має свою специфічну мету та завершується головною віхою (Milestone) — точкою прийняття рішення керівництвом щодо продовження або закриття проєкту.
   - *Inception (Початковий етап):* Визначення меж системи та бізнес-цілей.
   - *Elaboration (Проєктування):* Створення архітектури та усунення ключових ризиків.
   - *Construction (Побудова):* Масова розробка та кодування.
   - *Transition (Впровадження):* Передача продукту користувачам.
3. **Ітерації (Iterations):** Кожна фаза, своєю чергою, розбивається на одну або кілька ітерацій. Ітерація — це міні-проєкт, що включає всі види робіт (від вимог до тестування) і завершується створенням виконуваного (робочого) інкременту системи.

Статичний вимір (Робочі процеси, дисципліни, ролі, артефакти)

Статичний вимір відповідає на запитання: **«Хто що робить, як саме і який результат створює?»**

Базовими елементами статичного виміру є:

- **Роль (Role - «Хто?»):** Визначає поведінку та відповідальність особи або групи осіб (наприклад, Системний аналітик, Архітектор, Програміст). Важливо розуміти, що роль — це не посада. Одна людина може виконувати кілька ролей, або одну роль може ділити ціла команда.
- **Діяльність (Activity - «Як?»):** Конкретна задача, яку виконує роль. Діяльність має чітку мету (наприклад, «Розробити сценарій прецеденту», «Створити клас аналізу»).
- **Артефакт (Artifact - «Що?»):** Відчутний продукт, який створюється, змінюється або використовується під час виконання діяльності. Це може бути модель UML, документ специфікації вимог, вихідний код, план тестування або скомпільований файл.

Діяльності логічно згруповані у **9 ключових дисциплін** (робочих процесів):

- *6 Інженерних дисциплін:* Бізнес-моделювання, Управління вимогами, Аналіз та проєктування, Реалізація, Тестування, Розгортання.
- *3 Допоміжні дисципліни:* Управління конфігурацією та змінами, Управління проєктом, Середовище.

Взаємозв'язок вимірів (Графік «Гірба» - The RUP Hump Chart)

Магія RUP розкривається на перетині цих двох вимірів. Якщо накласти статичний вимір (дисципліни) на динамічний (фази), ми отримаємо класичний графік розподілу зусиль, відомий як **RUP Hump Chart** (Графік "Гірба").

Цей графік ілюструє найважливішу ідею RUP: **всі дисципліни виконуються на всіх фазах проєкту, але з різною інтенсивністю.**

- **На фазі Inception (Початковий етап):** Основні зусилля (найбільші "горби" на графіку) зосереджені на бізнес-моделюванні та управлінні вимогами. Однак розробка (реалізація) тут також присутня, хоча й у мінімальному обсязі (наприклад, створення Proof of Concept — доказу концепції).
- **На фазі Elaboration (Проєктування):** Пік зусиль припадає на дисципліну «Аналіз та проєктування». Вимоги уточнюються, а реалізація (кодування) зосереджується на створенні базового каркаса (архітектури).

- **На фазі Construction (Побудова):** Дисципліна «Реалізація» (написання коду) досягає свого абсолютного максимуму. Тестування також займає значну частину часу. Вимоги та проєктування вже відходять на задній план (система лише дошліфовується).
- **На фазі Transition (Впровадження):** Максимальні зусилля витрачаються на дисципліну «Розгортання» (Deployment) та «Тестування» (перевірка в реальних умовах, бета-тестування).

Висновок до Теми 1: Двовимірна архітектура RUP дозволяє команді не чекати повного завершення одного етапу, щоб перейти до іншого. Процес дихає і розвивається органічно: аналітики, архітектори, програмісти та тестувальники працюють паралельно в межах однієї ітерації, але їхня залученість гнучко змінюється залежно від поточної фази проєкту. **«Вступ до RUP. Основні принципи та концепції»** закладає фундаментальне розуміння того, як інженерія програмного забезпечення еволюціонувала від стихійного написання коду (підходу *Code and Fix*) до системних, керованих та прогнозованих процесів. Проаналізувавши історичний контекст «Кризи програмного забезпечення» та обмеження ранніх моделей життєвого циклу (каскадної та спіральної), ми визначили місце Rational Unified Process (RUP) як першої масштабної спроби об’єднати найкращі інженерні практики індустрії в єдиний гнучкий фреймворк.Опрацювання теоретичного матеріалу першого розділу дозволяє зробити кілька ключових інженерних висновків:

Синергія трьох китів RUP: Успіх проєкту за методологією RUP базується на нерозривному зв'язку його фундаментальних принципів. Процес **керується прецедентами (Use-case driven)**, що гарантує створення продукту, який точно відповідає бізнес-потребам замовника. Водночас розробка залишається **орієнтованою на архітектуру (Architecture-centric)** для забезпечення технічної стабільності системи, та реалізується через **ітеративний та інкрементний підхід**, який дозволяє виявляти й нівелювати критичні ризики на найраніших етапах.

- **Гнучкість двовимірної структури:** Матрична модель RUP наочно демонструє, що розробка складних систем не є лінійним процесом. Розподіл зусиль, відображений на класичному графіку «Гірба» (*RUP Hump Chart*), доводить: усі інженерні та допоміжні дисципліни виконуються паралельно протягом усього життєвого циклу проєкту, змінюється лише їхня інтенсивність залежно від поточної часової фази чи ітерації.
- **RUP як Process Framework:** Головна цінність RUP полягає в тому, що він є конструктором, а не догмою. Методологія вимагає обов'язкової адаптації (*Tailoring*) під масштаби, технології та кваліфікацію конкретної команди, що спростовує міф про її надмірну "важковаговість".

Базові концепції уніфікованого процесу. Чотири «П»: Персонал, Проект, Продукт і Процес

Підсумком проекту з розробки ПЗ є продукт, у створенні якого бере участь велика кількість різних людей. Процес розробки спрямовує зусилля персоналу, виступаючи як шаблон із розписаними кроками для виконання проекту. Цей процес автоматизується за допомогою відповідного інструментарію (утиліт).

Основні поняття розробки ПЗ:

Поняття	Визначення
Персонал (People)	Архітектори, розробники, тестувальники, менеджери проектів, замовники та користувачі. Це реальні люди (рушійна сила проекту), на відміну від абстрактної концепції «співробітник».
Проект (Project)	Організаційна сутність, за допомогою якої відбувається управління розробкою. Результатом проекту є випущений продукт.
Продукт (Product)	Сукупність артефактів, що створюються протягом життєвого циклу проекту (моделі, тексти програм, виконувані файли, документація).
Процес (Process)	Повний набір видів діяльності, необхідних для перетворення вимог користувача у продукт. Є своєрідним шаблоном для створення проекту.
Утиліти / Інструменти (Tools)	Програмне забезпечення, що використовується для автоматизації визначених у процесі видів діяльності.

Вплив процесу на персонал

Персонал вирішує все: фінансує продукт, планує розробку, створює, тестує та отримує вигоду. Процес розробки має бути орієнтованим на персонал і забезпечувати зручність роботи. Спосіб організації проекту суттєво впливає на команду через наступні аспекти:

- **Здійснимість (реалістичність) проекту:** Ітеративний підхід дозволяє оцінити виконуваність проекту на ранніх стадіях. Роботу над нездійсненними проектами можна припинити вчасно, зменшуючи моральні втрати команди.
- **Управління ризиками:** Дослідження та пом'якшення ризиків на ранніх стадіях підвищує ефективність персоналу, усуваючи відчуття невизначеності.
- **Структура команди:** Максимальна ефективність досягається у невеликих групах (6–8 осіб). Вдала архітектура з чіткими інтерфейсами дозволяє розподілити роботу між такими малими групами.
- **План проекту:** План має бути реалістичним. Нереалістичні плани стрімко знижують продуктивність та бойовий дух команди.
- **Зрозумілість проекту:** Кожен учасник бажає бачити загальну картину. Опис архітектури надає таку можливість усім залученим особам.

• **Відчуття закінченості:** Часті відгуки від користувачів в ітеративному життєвому циклі (ЖЦ) збільшують темп робіт та підсилюють відчуття завершеності проміжних етапів.

Співробітники, обов'язки та ролі

Завдання Уніфікованого процесу (УП) — спрямувати розробників на правильне визначення вимог, вибір надійної архітектури та створення складних систем з найменшими витратами часу та ресурсів.

Трансформація ресурсу в співробітника:

Компанія переводить людину зі статусу абстрактного «ресурсу» на конкретну позицію **«співробітника».**

• **Співробітник (Worker):** Це позиція або роль, яку людина відіграє в розробці ПЗ (наприклад, архітектор, інженер з компонентів, тестувальник інтеграції).

• **Екземпляр співробітника:** Кожен співробітник асоціюється з певним набором видів діяльності. Для ефективної роботи йому потрібна інформація про те, як виконувати ці дії та як його роль співвідноситься з іншими. Співробітник може бути представлений як однією особою, так і групою (наприклад, архітектурний відділ).

• **Обов'язки та навички:** Менеджер проекту повинен співвідносити реальні навички людей з необхідною компетенцією для позиції співробітника. Протягом проекту одна людина може змінювати ролі (наприклад, специфікатор вимог згодом стає інженером з компонентів для цієї ж підсистеми, що мінімізує втрату контексту).

Контрольні запитання до теми1:

Блок 1. Еволюція методологій та місце RUP (за матеріалами Лекції 1)

1. Що таке «криза програмного забезпечення» (Software Crisis) і які ключові фактори призвели до її виникнення?
2. Порівняйте каскадну (Waterfall) та спіральну (Spiral) моделі розробки. У чому полягають їхні головні переваги та критичні недоліки?
3. Який внесок зробили «Три Аміго» (Граді Буч, Джеймс Рамбо, Івар Якобсон) у розвиток інженерії програмного забезпечення?
4. Що означає термін «фреймворк процесу» в контексті RUP? Чому RUP не є жорстким, незмінним алгоритмом дій?

5. Яке місце займає класичний RUP на шкалі між жорсткими (предиктивними) та гнучкими (Agile) методологіями?

Блок 2. Фундаментальні принципи RUP (за матеріалами Лекції 2)

6. Розкрийте суть принципу «Керованість прецедентами» (Use-case driven). Як концепція Use Case впливає на роботу аналітиків, розробників та тестувальників?

7. Що означає принцип «Орієнтованість на архітектуру» (Architecture-centric)? Чому архітектура має першочергове значення на ранніх етапах розробки?

8. Опишіть архітектурну модель «4+1» Філіпа Кручтена. Які саме вигляди (Views) до неї входять і для яких ролей у команді вони призначені?

9. У чому полягає різниця між *ітеративністю* та *інкрементністю* розробки?

10. Як ітеративний підхід у RUP допомагає мінімізувати проєктні та технічні ризики?

Блок 3. Архітектура процесу RUP (за матеріалами Лекції 3)

11. Поясніть концепцію двовимірної моделі процесу RUP. Що відображають динамічний (горизонтальна вісь) та статичний (вертикальна вісь) виміри?

12. Назвіть 4 фази життєвого циклу проєкту за методологією RUP. Яка головна мета кожної з цих фаз?

13. Що таке ітерація в термінології RUP? Чим вона концептуально відрізняється від фази?

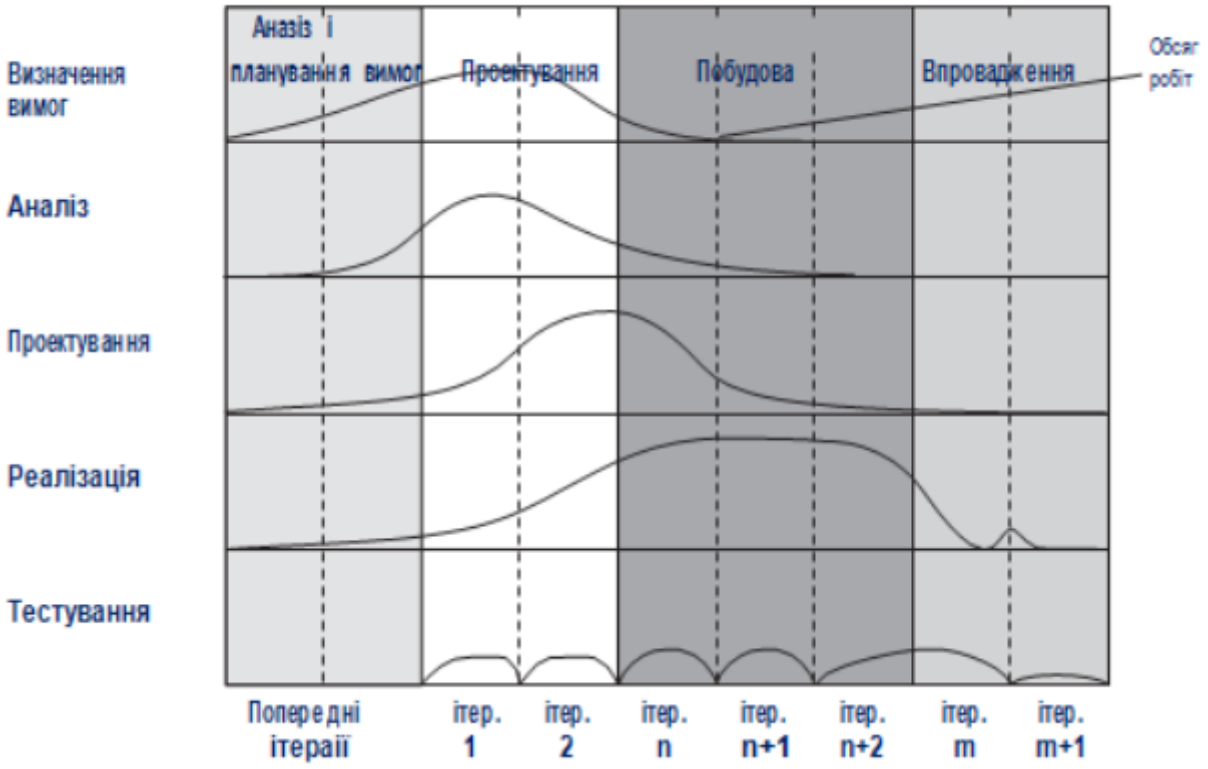
14. Дайте визначення базовим елементам статичного виміру: *Роль, Діяльність, Артефакт*. Як вони взаємодіють між собою?

15. Перелічіть 6 основних інженерних та 3 допоміжні робочі дисципліни RUP.

16. Що ілюструє графік «Гірба» (RUP Hump Chart)? Поясніть на його прикладі, як змінюється інтенсивність дисципліни «Аналіз та проєктування» при переході від фази Inception до фази Construction.

Тема 2. Динамічна структура: Фази життєвого циклу

У цій темі ми переходимо від загальних принципів до часової (динамічної) організації процесу розробки. Класичний життєвий цикл RUP розбитий на чотири послідовні фази, кожна з яких має свою унікальну мету, специфічний набір робіт та завершується жорсткою контрольною точкою — віхою (Milestone).



Лекція 4. Фаза Inception (Початковий етап). Визначення меж системи та бачення проєкту

Мета лекції: Зрозуміти сутність першої фази життєвого циклу RUP, навчитися ідентифікувати зацікавлені сторони, формувати документ Бачення (Vision) та складати економічне обґрунтування (Business Case) для старту проєкту.

Сутність та головна мета фази Inception

Типова помилка розробників-початківців — намагання одразу писати код або малювати детальну архітектуру після отримання першого запиту від замовника. RUP категорично забороняє такий підхід.

Фаза **Inception (Початковий етап)** — це найкоротша, але стратегічно найважливіша фаза. Її головне завдання — не розробити систему, а відповісти на фундаментальне запитання: **«Чи варто взагалі починати цей проєкт?»**

На цій стадії проєкт може бути скасований, якщо з'ясується, що він технічно неможливий, не принесе очікуваного прибутку або ризики занадто високі.

Основні цілі фази:

- Зрозуміти, *що* саме ми збираємося будувати (сформувати бачення).
- Визначити межі системи (що входить у проєкт, а що — ні).
- Скласти первинну оцінку вартості та термінів.
- Ідентифікувати найкритичніші ризики.

Аналіз зацікавлених сторін (Stakeholders Analysis)

Програмне забезпечення створюється для людей. Тому перший крок — зрозуміти, хто ці люди.

Зацікавлена сторона (Stakeholder) — це будь-яка особа, група осіб або організація, яка може впливати на проєкт або на яку вплине результати впровадження цього проєкту.

Кого ми маємо ідентифікувати:

- **Спонсор (Замовник):** Той, хто платить гроші і визначає бізнес-цілі (наприклад, збільшити продажі на 20%).
- **Кінцеві користувачі (End Users):** Ті, хто безпосередньо працюватиме з системою (менеджери, оператори, клієнти). Їхні вимоги часто кардинально відрізняються від бачення Спонсора.
- **Експерти предметної області (Domain Experts):** Люди, які знають, як працює бізнес «зсередини» (наприклад, головний бухгалтер для фінансової системи).
- **Технічний персонал:** Системні адміністратори, які будуть підтримувати систему після релізу.

Практична порада: Якщо на стадії Inception пропустити хоча б одну ключову зацікавлену сторону (наприклад, службу безпеки), на стадії розгортання проєкт може бути заблокований через невідповідність політикам компанії.

Визначення меж системи та Документ «Бачення» (Vision)

Після того, як ми зрозуміли, *хто* наші стейкхолдери, ми маємо узгодити з ними, *що саме* буде створено. Це фіксується у головному артефакті цієї фази — документі **Vision (Бачення).**

Документ Vision не містить технічних деталей чи архітектури баз даних. Він написаний бізнес-мовою і зазвичай включає такі розділи:

1. **Постановка проблеми (Problem Statement):** Яку проблему ми вирішуємо? *Приклад: "Через ручну обробку замовлень компанія втрачає 15% клієнтів".*
2. **Визначення меж системи (System Boundaries):** Чітке розмежування відповідальності. Що система *повинна* робити, а що знаходиться *поза* її компетенцією. Це захищає команду від "повзучого розширення меж" (Scope Creep) у майбутньому.
3. **Ключові можливості (Features):** Високорівневий перелік того, що система зможе робити.
4. **Первинна модель прецедентів (Use Case Model):** На етапі Inception не потрібно описувати всі прецеденти. Аналітик виявляє 100% акторів і лише 10-20% найважливіших (критичних) прецедентів, які формують ядро системи.

Економічне обґрунтування (Business Case) та управління ризиками

Щоб інвестор дав зелене світло проєкту, йому потрібно показати цифри. Документ **Business Case (Економічне обґрунтування)** містить відповідь на запитання, чи є проєкт фінансово і стратегічно доцільним.

Ключові компоненти Business Case:

- **Очікуваний повернення інвестицій (ROI):** Як продукт окупиться (зменшення витрат, збільшення доходу, вихід на новий ринок).
- **Груба оцінка ресурсів (Rough Order of Magnitude - ROM):** Первинна оцінка бюджету та термінів. На фазі Inception ця оцінка має похибку від 50% до 100%, і це нормально (це так званий "конус невизначеності").
- **Ідентифікація ризиків:**
  - *Бізнес-ризики:* Чи зміниться законодавство? Чи випередять конкуренти?
  - *Технічні ризики:* Чи витримає обрана технологія навантаження?
  - *Проєктні ризики:* Чи достатньо у нас кваліфікованих кадрів?

RUP наполягає: ризики не можна ігнорувати. Якщо технічний ризик занадто високий (наприклад, потрібна інтеграція із застарілим API, документації до якого немає), в рамках Inception створюється невеликий прототип (Proof of Concept), щоб перевірити, чи це взагалі можливо.

Віха LCO: Цілі життєвого циклу (Lifecycle Objective)

Фаза Inception завершується контрольною точкою — **Віхою LCO (Lifecycle Objective)**.

Щоб проєкт перейшов на наступну фазу (Elaboration), замовник, інвестори та команда розробки мають зібратися і погодитися з такими твердженнями:

1. **Згода щодо меж:** Усі стейкхолдери однаково розуміють, що ми будуємо, і згодні з документом Vision.
2. **Зрозумілість вимог:** Ключові прецеденти ідентифіковані та зрозумілі.
3. **Економічна доцільність:** Business Case виглядає переконливо, бюджет затверджено щонайменше на наступну фазу.
4. **Ризики під контролем:** Найстрашніші ризики виявлені, і ми маємо план їх пом'якшення (mitigation).

Результат віхи: Якщо критерії віхи LCO не виконані, проєкт або скасовується (Go/No-Go decision = No-Go), або фаза Inception подовжується на ще одну ітерацію для з'ясування незрозумілих моментів. Якщо рішення позитивне — команда переходить до фази проєктування (Elaboration).

Лекція 5. Фаза Elaboration (Проєктування). Ідентифікація та усунення ключових технічних ризиків. Створення базової виконуваної архітектури

Мета лекції: Спростувати міф про те, що проєктування — це лише створення паперових діаграм, та розібрати ключове завдання фази Elaboration: побудову життєздатного архітектурного каркаса системи (Executable Architecture), здатного витримати подальше нарощування функціоналу.

Сутність та завдання фази Elaboration

Якщо на фазі *Inception* команда відповідала на запитання **«Що ми робимо і чи варто це робити?»**, то на фазі *Elaboration* фокус зміщується на запитання **«Як саме ми будемо це будувати?»**.

Серед розробників-початківців існує хибна думка, що стадія проєктування (Design) полягає виключно у малюванні UML-діаграм та написанні товстих

специфікацій, після чого ці "папери" передаються програмістам для кодування. У методології RUP фаза Elaboration має принципово інший характер. Це **найважча та найризикованіша інженерна стадія проєкту**.

Основні цілі фази:

1. Стабілізувати бачення продукту та деталізувати більшість вимог (близько 80% прецедентів мають бути детально описані).
2. Спроєктувати, реалізувати та протестувати базову архітектуру.
3. Звести до мінімуму ключові технічні ризики.
4. Створити точний план (графік і бюджет) для наступної фази (Construction).

Ідентифікація та усунення технічних ризиків

RUP — це процес, керований ризиками. Ризики не записують у табличку, щоб забути про них; їх активно "атакують". На фазі Elaboration команда повинна довести, що обраний технологічний стек дійсно здатний вирішити поставлені бізнес-задачі.

Типові технічні ризики на цьому етапі:

- *Продуктивність та масштабованість:* Чи витримає система навантаження у 10 000 одночасних користувачів?
- *Інтеграція:* Чи зможемо ми підключитися до застарілої бази даних клієнта, яка не має сучасного API?
- *Технологічні прогалини:* Чи має команда достатньо досвіду роботи з новою хмарною платформою чи специфічним фреймворком?

Як RUP усуває ці ризики? Виключно через створення реального коду та прототипів (Proof of Concept). Якщо є сумніви щодо пропускної здатності бази даних, команда створює тестовий скрипт, генерує мільйон фейкових записів і вимірює час відгуку. Якщо результати незадовільні, архітектура (або технологія) змінюється *зараз*, коли написано ще мінімум коду, а не за тиждень до релізу.

Створення базової виконуваної архітектури (Executable Architecture)

Це центральне поняття фази Elaboration і всієї методології RUP.

Виконувана архітектура — це не документ і не набір діаграм. Це **реальний, скомпільований і працюючий програмний код**, який реалізує найважливіші (архітектурно значущі) прецеденти системи.

Уявіть будівництво хмарочоса: виконувана архітектура — це фундамент, сталевий каркас і прокладені магістральні комунікації. У цій будівлі ще немає стін, вікон, ліфтів чи шпалер (це все буде зроблено на фазі Construction), але каркас вже стоїть і здатен витримати навантаження.

Що включає в себе Executable Architecture у сучасних реаліях:

- Налаштоване середовище розробки та розгортання. У сучасних умовах це часто означає створення базового інфраструктурного каркаса: написані конфігурації для контейнеризації (наприклад, Docker-файли для відокремлення середовищ), підняті та налаштовані вебсервери (на зразок Nginx), базове кешування (наприклад, за допомогою Redis) та налаштовані бази даних.
- Реалізовані механізми взаємодії між основними підсистемами (наприклад, API-шлюзи).
- Прописані базові класи (Domain Entities) та інтерфейси.
- Реалізований наскрізний шлях (Tracer Bullet) принаймні для одного-двох критичних сценаріїв — від клієнтського інтерфейсу до запису в базу даних.

Коли архітектура "виконувана", її можна протестувати навантажувальними тестами і переконатися, що фундаментальні рішення прийняті правильно.

Деталізація вимог та планування

Паралельно з побудовою архітектури, системні аналітики продовжують роботу з вимогами:

- Уточнюються всі альтернативні потоки в Use Cases (що буде, якщо відключиться інтернет під час оплати? Що, якщо користувач введе невалідні дані?).
- Формується детальний план розробки. Оскільки найскладніші технічні проблеми вже вирішено, "конус невизначеності" звужується. Тепер менеджер проєкту може з високою точністю (похибка близько 10%) сказати, скільки людей, часу та грошей знадобиться на масову розробку функціоналу.

Віха LCA: Архітектура життєвого циклу (Lifecycle Architecture)

Фаза Elaboration завершується другою головною віхою проєкту — **LCA (Lifecycle Architecture Milestone).**

Це момент істини для проєкту. Щоб перейти до масового написання коду (фаза Construction), команда повинна підтвердити:

1. **Стабільність вимог:** Бачення продукту та основні прецеденти зафіксовані. Замовник погоджується з тим, *як* система буде виглядати та працювати.
2. **Наявність працюючого каркаса:** Створена та протестована базова виконувана архітектура.
3. **Усунення ризиків:** Жоден серйозний технічний ризик більше не загрожує проєкту. Усі "невідомі" перетворені на "відомі".
4. **Реалістичний план:** Затверджено детальний план ітерацій для фази розробки.

Якщо віха LCA пройдена успішно, проєкт переходить у стадію відносно спокійної, конвеєрної розробки — нарощування "м'язів" на вже існуючий "скелет".

Лекція 6. Фаза Construction (Побудова). Ітеративна розробка функціоналу. Оптимізація ресурсів, управління кодовою базою, паралельна розробка компонентів

Мета лекції: Зрозуміти специфіку найтривалішої та найбільш ресурсомісткої фази життєвого циклу RUP. Дослідити, як відбувається масштабування команди, організація паралельної розробки та управління великими обсягами коду за допомогою практик безперервної інтеграції.

Сутність та головна мета фази Construction

*Я*кщо фаза *Elaboration* (Проєктування) нагадувала роботу конструкторського бюро, де створювався унікальний прототип та тестувався фундамент (Executable Architecture), то фаза **Construction (Побудова)** — це перехід до заводського конвеєра.

На цій стадії фундаментальні архітектурні рішення вже прийняті, а критичні ризики усунуті. Фокус проєкту кардинально зміщується з *дослідження та проєктування* на **продуктивність, швидкість, передбачуваність та якість кодування**.

Основні цілі фази:

1. Мінімізація витрат на розробку шляхом оптимізації процесів.
2. Швидке та ітеративне нарощування функціоналу (реалізація решти 80% прецедентів, які не ввійшли в базову архітектуру).
3. Досягнення адекватної якості коду через регулярне тестування.

4. Створення робочої версії продукту (бета-версії), готової до передачі користувачам.

Ітеративна розробка функціоналу на базі існуючої архітектури

Розробка на фазі Construction ніколи не йде "суцільним потоком" (Big Bang). Вона чітко розбита на ітерації (зазвичай тривалістю 2–4 тижні).

Оскільки технічні ризики вже вирішені, планування ітерацій тепер керується **бізнес-пріоритетами**. Замовник (Спонсор) визначає, які функції (прецеденти) є найважливішими для ринку, і саме вони потрапляють у найближчий спринт (ітерацію).

Життєвий цикл кожної ітерації на етапі Побудови:

- **Проєктування (Design):** Відбувається на мікрорівні (рівні окремих класів та модулів). Архітектуру в цілому вже не змінюють.
- **Реалізація (Implementation):** Написання бізнес-логіки. Це період найвищої активності програмістів.
- **Тестування (Testing):** Обов'язкове написання Unit-тестів розробниками та проведення інтеграційного тестування QA-інженерами. Кожна ітерація завершується створенням стабільного, скомпільованого релізу (Internal Release).

Оптимізація ресурсів та масштабування команди

Фаза Construction супроводжується стрімким збільшенням команди. Під час початкових фаз (Inception та Elaboration) на проєкті працює невелика група висококваліфікованих фахівців (архітектори, бізнес-аналітики, системні інженери). На стадії Побудови до команди масово приєднуються розробники різних рівнів (Junior, Middle), тестувальники, технічні письменники.

Чому це масштабування є безпечним? У програмній інженерії існує відомий закон Брукса: *"Додавання робочої сили до проєкту, що запізнюється, робить його ще пізнішим"*. Це відбувається через різке зростання витрат часу на комунікацію.

RUP нейтралізує цей закон саме завдяки фазі Elaboration. Оскільки базова архітектура вже створена, система розділена на незалежні підсистеми з чітко прописаними інтерфейсами (API). Це дозволяє:

- **Розподіляти роботу:** Навчати нових розробників значно простіше, адже їм не потрібно розуміти систему цілком — достатньо знати правила роботи у своєму конкретному модулі.

- **Оптимізувати витрати:** На рутинні задачі з кодування та написання тестів залучаються менш досвідчені фахівці, робота яких контролюється старшими розробниками через механізми Code Review.

Паралельна розробка компонентів та управління кодовою базою

Уявіть команду з 20–30 розробників, які одночасно пишуть код. Без жорсткої дисципліни це неминуче призведе до так званого **«інтеграційного пекла» (Integration Hell)** — коли код кожного окремого розробника працює на його локальній машині, але при спробі об'єднати його в єдину систему все ламається.

Для забезпечення паралельної розробки RUP покладається на дві критично важливі концепції, які сьогодні еволюціонували у фундамент DevOps:

1. **Управління конфігураціями (Configuration Management) та Ізоляція середовищ:**
   - Кожна команда працює над власним компонентом ізольовано. Використовуються системи контролю версій (сучасний аналог — розгалуження в Git: feature-branches).
   - Середовища розробки мають бути стандартизовані. (Сьогодні для цього активно використовуються технології контейнеризації на кшталт **Docker**, що гарантують ідентичність середовища на машині розробника, тестовому сервері та в продакшені, незалежно від налаштувань локальних вебсерверів чи баз даних).
2. **Безперервна інтеграція (Continuous Integration - CI):**
   - RUP вимагає, щоб інтеграція підсистем відбувалася постійно, а не в кінці фази.
   - Щойно розробник закінчує свій модуль, його код зливається з основною гілкою. Відбувається автоматична збірка (Build) всієї системи та запускаються автоматизовані тести.
   - Якщо збірка "падає", пріоритетом всієї команди стає її негайне виправлення. Це гарантує, що система знаходиться у робочому стані 100% часу на фазі Construction.

Віха IOC: Початкова готовність до експлуатації (Initial Operational Capability)

Фаза Construction вважається завершеною, коли команда досягає віхи **IOC (Initial Operational Capability)**. Це означає, що продукт готовий покинути "тепличні" умови середовища розробки і зустрітися з реальними користувачами.

Критерії успішного проходження віхи IOC:

1. **Code Complete (Код завершено):** Усі критичні та важливі прецеденти реалізовані. Додавання нових функцій заморожується (Feature Freeze).
2. **Стабільність:** Система працює надійно, критичних дефектів (Critical Bugs) немає. Усі виявлені баги мають відомі шляхи обходу або визнані незначними.
3. **Готовність артефактів розгортання:** Створені інсталяційні пакети, налаштовані скрипти міграції баз даних.
4. **Користувацька документація:** Написані посібники користувача (User Manuals) та інструкції для системних адміністраторів.

Якщо віха IOC пройдена успішно, проєкт отримує статус **Beta-релізу** і переходить до фінальної стадії — фази Впровадження (Transition), де система буде встановлена в реальному бізнес-середовищі замовника.

Лекція 7. Фаза Transition (Впровадження). Підготовка до релізу, бета-тестування та розгортання в робочому середовищі

Мета лекції: Ознайомитися з фінальною стадією життєвого циклу RUP, зрозуміти процеси передачі програмного продукту кінцевим користувачам, специфіку бета-тестування та критерії успішного завершення проєкту.

Сутність та головна мета фази Transition

Після успішного завершення фази *Construction* (Побудова) розробка нового функціоналу офіційно заморожується (Feature Freeze). Команда досягла точки, коли система має весь необхідний код для виконання бізнес-завдань.

Тепер починається фаза **Transition (Впровадження або Перехід)**. Її головна мета — плавно, безпечно та ефективно передати готовий програмний продукт у руки кінцевих користувачів.

Уявіть, що ви збудували сучасний житловий комплекс. Будівельні роботи завершено, але мешканці ще не можуть туди заїхати. Потрібно вивезти будівельне сміття, перевірити роботу ліфтів під навантаженням, видати ключі, пояснити ОСББ, як керувати котельнею, і виправити дрібні недоліки (наприклад, підфарбувати стіни, які подряпали під час перевірки). Саме цим і займається команда на фазі Впровадження.

Основні завдання фази:

- Тестування продукту в реальних умовах експлуатації (бета-тестування).
- Виправлення знайдених дефектів пізніх стадій.

- Міграція даних зі старих (legacy) систем замовника у нову базу даних.
- Фізичне розгортання ПЗ на серверах замовника або в робочому хмарному середовищі (Production environment).
- Навчання користувачів та передача відповідальності службі технічної підтримки.

Бета-тестування та зворотний зв'язок

На попередній фазі тестування виконували професійні QA-інженери (Альфа-тестування). Вони перевіряли систему за заздалегідь написаними сценаріями. Але якими б хорошими не були тестувальники, вони мислять як інженери.

Бета-тестування — це передача системи реальним кінцевим користувачам (або фокус-групі), які намагаються виконувати свої щоденні робочі завдання.

Чому це критично важливо?

- *Непередбачувана поведінка:* Резервний сервер може впасти не через помилку в коді, а тому, що бухгалтер спробує завантажити звіт за 10 років в одному Excel-файлі.
- *Оцінка юзабіліті (зручності):* Кнопка "Зберегти" працює ідеально, але користувачі скаржаться, що до неї треба "тягнутися" через три екрани, що сповільнює їхню роботу вдвічі.
- *Перевірка на реальних даних:* Тестові бази даних завжди "стерильні". Реальні дані часто містять дублікати, пропущені поля та помилки кодувань, з якими нова система повинна вміти працювати.

Виправлення дефектів пізніх стадій та Release Candidates

Під час бета-тестування команда починає отримувати шквал звітів про помилки. На цьому етапі запроваджується жорсткий **процес сортування (Triage)** дефектів.

- **Блокуючі або критичні помилки** (наприклад, неможливість провести платіж або втрата даних) виправляються негайно.
- **Дрібні дефекти або запити на покращення** (наприклад, зміна кольору форми чи додавання нового фільтра) зазвичай відхиляються або переносяться в беклог для наступної версії (v.2.0). Вносити серйозні зміни в код на цій фазі суворо заборонено, оскільки кожне втручання може породити нові критичні помилки.

Після серії виправлень команда випускає **Release Candidate (RC)** — версію-кандидат. Це збірка, яка потенційно може стати фінальним продуктом, якщо під час її тестування не буде знайдено жодної критичної проблеми. Може бути випущено кілька кандидатів (RC1, RC2...), перш ніж система стабілізується.

Розгортання (Deployment) та навчання користувачів

Коли система стабілізована, настає етап безпосереднього впровадження в бізнес-середовище. Це один із найстресовіших моментів проєкту.

Міграція даних: Дуже рідко програмне забезпечення встановлюється "з чистого аркуша". Зазвичай замовник вже має якусь систему (навіть якщо це просто набір таблиць в Excel). Дані з цих старих систем потрібно безпечно перенести в нову БД, не втративши жодного запису.

Стратегії розгортання: У класичному RUP (і в сучасних DevOps-практиках) намагаються уникати розгортання за принципом "вимкнули старе — увімкнули нове". Натомість використовують паралельне впровадження або поступове розгортання (наприклад, спочатку на один відділ компанії, а через тиждень — на всю організацію).

Навчання та передача знань: Програмний продукт не вважається завершеним без:

- *Посібників користувача (User Manuals):* Інструкції для тих, хто безпосередньо працює з інтерфейсом.
- *Адміністративних посібників (Admin Guides):* Документація для системних адміністраторів щодо налаштування серверів, створення резервних копій та моніторингу логів.
- *Тренінгів:* Проведення семінарів для персоналу замовника.

Віха PR: Випуск продукту (Product Release)

Фаза Transition (і весь цикл розробки) завершується фінальною контрольною точкою — **Віхою PR (Product Release)**.

Для проходження цієї віхи замовник та команда розробки підписують акт приймання-передачі. Успіх проєкту на цій стадії визначається відповідями на такі запитання:

1. **Чи задоволений користувач?** Чи система дійсно вирішує ті бізнес-проблеми, які були описані в документі Vision на першій фазі (Inception)?

2. **Чи стабільна система?** Чи рівень дефектів є прийнятним і не блокує критичні бізнес-процеси?

3. **Чи готова підтримка?** Чи навчений персонал замовника, чи здатна служба підтримки (HelpDesk) самостійно вирішувати типові інциденти?

Що відбувається далі? Якщо віха пройдена успішно, проєкт вважається завершеним. Після цього система переходить у стадію **супроводу (Maintenance)**. Якщо замовник вирішує суттєво розширити функціонал системи або перевести її на нову технологічну платформу, запускається новий цикл розробки RUP (Еволюція системи), який знову починається з фази Inception для версії 2.0.

Лекція 8. Віхи (Milestones) та метрики переходу. Критерії успішності проходження ключових віх (LCO, LCA, IOC, PR)

Мета лекції: Зрозуміти механізм контролю та управління проєктом у RUP через систему ключових віх. Навчитися об'єктивно оцінювати готовність проєкту до переходу на наступну фазу за допомогою кількісних та якісних метрик.

Концепція віх (Milestones) у методології RUP

У традиційних моделях розробки (наприклад, каскадній) перехід між етапами часто відбувається просто тому, що "закінчився час" або "написано документ". RUP підходить до цього інакше: фаза не завершується, поки не досягнуто конкретних, вимірюваних результатів.

Точка завершення кожної з чотирьох фаз життєвого циклу називається **Головною віхою (Major Milestone)**.

Віха — це не просто дата в календарі. Це точка прийняття критичного управлінського рішення, так званий **"Go/No-Go Decision" (Рішення про продовження або зупинку)**. У цій точці замовник (Спонсор) і керівник проєкту зустрічаються, аналізують стан справ і вирішують:

- **Go:** Рухатися далі (виділяти бюджет на наступну фазу).
- **Rework:** Залишитися на поточній фазі ще на одну ітерацію, щоб допрацювати недоліки.
- **No-Go:** Закрити проєкт (якщо ризики занадто високі, а продовження економічно недоцільне).

Розглянемо чотири ключові віхи детальніше.

Віха LCO (Lifecycle Objective) — Цілі життєвого циклу

Це контрольна точка завершення фази **Inception (Початкового етапу)**. Її головне завдання — переконатися, що всі стейкхолдери однаково розуміють, *що саме* ми збираємося робити, і згодні, що це *варто* робити.

Критерії успішності проходження LCO:

1. **Узгодженість меж (Scope Agreement):** Замовник і команда розробки підписали документ Vision (Бачення). Усі чітко розуміють, які функції увійдуть до системи, а які залишаться поза її межами.
2. **Економічне обґрунтування (Business Case):** Затверджено первинний бюджет та графік проєкту. Інвестор згоден фінансувати щонайменше наступну фазу (Elaboration).
3. **Ідентифікація ризиків:** Складено реєстр основних ризиків (бізнес-, технологічних, проєктних) та визначено стратегії їх пом'якшення.
4. **Виявлення критичних прецедентів:** Основні актори та ключові (найважливіші для бізнесу) Use Cases визначені.

Якщо віху не пройдено: Проєкт ризикує перетворитися на фінансову чорну діру через постійну зміну вимог ("повзуче розширення меж").

Віха LCA (Lifecycle Architecture) — Архітектура життєвого циклу

Це найважливіша інженерна віха проєкту. Вона завершує фазу **Elaboration (Проєктування)**. На цьому етапі вирішується доля технічної реалізованості продукту.

Критерії успішності проходження LCA:

1. **Стабільна виконувана архітектура (Executable Architecture):** Команда не просто намалювала діаграми, а написала і скомпілювала базовий каркас системи. Архітектура протестована і здатна витримати заявлені навантаження.
2. **Нейтралізація технічних ризиків:** Усі критичні технологічні невідомі (наприклад, інтеграція з незнайомим API або вибір бази даних) перевірені на практиці через прототипи.
3. **Стабілізація вимог:** Близько 80% прецедентів (Use Cases) детально описані з усіма альтернативними потоками.
4. **Детальний план побудови:** Менеджер проєкту надав точний графік ітерацій, обсяг ресурсів та бюджет для наступної фази (Construction). "Конус невизначеності" звузився до мінімуму.

Якщо віху не пройдено: Переходити до масового написання коду суворо заборонено. Написання коду на нестабільній архітектурі призведе до необхідності переписувати всю систему з нуля.

Віха IOC (Initial Operational Capability) — Початкова готовність до експлуатації

Ця віха знаменує завершення фази **Construction (Побудова)**. Це точка переходу від стану "система розробляється" до стану "система готова до тестування користувачами".

Критерії успішності проходження IOC:

1. **Код завершено (Code Complete):** Усі заплановані функціональні можливості реалізовані. Додавання нових фіч заморожено.
2. **Прийнятний рівень якості:** Система пройшла внутрішнє тестування (Альфа-тестування). Немає жодного відкритого дефекту рівня "Критичний" або "Блокуючий".
3. **Готовність до розгортання:** Створені інсталяційні пакети, скрипти для налаштування баз даних.
4. **Наявність документації:** Підготовлені чорнові версії посібників користувача (User Manuals) та інструкцій з адміністрування.
5. **Готовність бета-версії:** Продукт готовий до передачі кінцевим користувачам для роботи в умовах, наближених до реальних.

Якщо віху не пройдено: Випуск сирого, нестабільного продукту на реальних користувачів знищить репутацію компанії та спричинить лавину звернень до служби підтримки.

Віха PR (Product Release) — Випуск продукту

Фінальна віха всього життєвого циклу, що завершує фазу **Transition (Впровадження).** Це момент, коли замовник офіційно приймає роботу.

Критерії успішності проходження PR:

1. **Схвалення користувачами (User Acceptance):** Бета-тестування успішно завершено. Система вирішує бізнес-завдання, описані ще в документі Vision.
2. **Успішне розгортання:** ПЗ встановлено в робочому середовищі замовника (Production). Міграція старих даних пройшла без втрат.

3. **Навчання завершено:** Кінцеві користувачі та адміністратори пройшли тренінги і знають, як користуватися системою.

4. **Передача в супровід:** Обов'язки щодо підтримки системи передані від команди розробки до служби технічної підтримки (HelpDesk / Maintenance Team).

Після проходження віхи PR проєкт вважається офіційно завершеним.

Метрики оцінки готовності до переходу

Як керівник проєкту дізнається, що критерії віхи виконані? Для цього в RUP використовуються об'єктивні метрики — кількісні показники стану проєкту.

- **Метрики стабільності вимог (Requirement Churn):** Вимірюють, скільки відсотків вимог змінюється з часом. Якщо наприкінці фази Elaboration щотижня змінюється більше 5% вимог, переходити до Construction не можна — система ще не стабільна.
- **Щільність дефектів (Defect Density):** Кількість знайдених помилок на тисячу рядків коду (KLOC) або на один Use Case. На фазі Construction цей показник спочатку стрімко зростає, але перед віхою IOC має продемонструвати стійке падіння (Trend Analysis).
- **Метрики збіжності (Convergence Metrics):** Графіки, що показують співвідношення між кількістю виявлених дефектів та кількістю виправлених. Перехід до фази Transition можливий лише тоді, коли лінія виправлених дефектів стабільно перетинає і йде вище лінії знайдених.
- **Покриття тестами (Test Coverage):** Відсоток коду або сценаріїв прецедентів, які перевіряються автоматизованими тестами.

Використання цих метрик позбавляє процес прийняття рішень від суб'єктивності ("мені здається, ми готові") і переводить управління проєктом у площину інженерного розрахунку.

Висновок до теми 2

Тема «Динамічна структура: Фази життєвого циклу» розкриває часовий вимір методології Rational Unified Process (RUP) і демонструє, як структурується життєвий цикл проєкту від появи першої ідеї до передачі готового програмного продукту замовнику. Замість лінійного, часто сліпого слідування заздалегідь визначеному плану, RUP пропонує чітко розмежовані часові етапи, кожен з яких фокусується на вирішенні конкретного класу інженерних та управлінських завдань.

Опрацювання матеріалу цього розділу дозволяє сформувати кілька критично важливих висновків для майбутнього інженера програмного забезпечення:

- **Логіка чотирьох фаз як основа життєздатності проєкту:** Кожна фаза RUP має унікальний вектор зусиль. **Inception** захищає проєкт від фінансового краху на старті, відповідаючи на запитання доцільності. **Elaboration** усуває технічну сліпоту, створюючи перевірений на практиці архітектурний каркас. **Construction** переводить проєкт у режим високопродуктивного виробничого конвеєра, а **Transition** забезпечує делікатне та безпечне впровадження продукту в реальне бізнес-середовище.
- **Віхи (Milestones) як інструмент суворого контролю якості:** Головні віхи проєкту (**LCO, LCA, IOC, PR**) є фундаментальними "воротами якості" (Quality Gates) та точками прийняття рішень (*Go/No-Go*). RUP відкидає формальний перехід між етапами лише за календарною датою. Проєкт рухається вперед тільки тоді, коли досягнуто вимірюваних результатів і усунуто відповідні ризики. Найважливішою інженерною точкою є віха **LCA (Архітектура життєвого циклу)**, яка забороняє масове кодування, поки не доведено стабільність виконуваної архітектури.
- **Звуження «конуса невизначеності»:** Динамічна структура RUP дозволяє менеджменту та замовнику бачити, як похибка в оцінці термінів і бюджету зменшується з кожною фазою. Завдяки ранній стабілізації вимог та архітектури на етапі проєкування, етап побудови стає прогнозованим, мінімізуючи хаос та зриви дедлайнів.

Контрольні запитання до Теми 2

Блок 1. Бізнес-моделювання та Управління вимогами

1. Чому розробка програмного забезпечення в RUP починається з бізнес-моделювання, а не зі збору технічних вимог?
2. Поясніть різницю між моделями бізнес-процесів *"As-Is"* («Як є») та *"To-Be"* («Як має бути»). Яку роль ця різниця відіграє для аналітика?
3. Розкрийте суть моделі якості **FURPS+**. Які категорії вимог вона описує і що означає символ «+»?
4. Що таке Специфікація програмних вимог (**SRS**) і чому її вважають «контрактом» між замовником та командою розробки?
5. Опишіть призначення **Матриці трасування (Traceability Matrix)**. Як вона допомагає при аналізі впливу змін (*Impact Analysis*)?

Блок 2. Модель прецедентів (Use Case Model)

6. Чому графічна діаграма прецедентів UML є лише «змістом» вимог? З яких обов'язкових елементів складається текстова специфікація Use Case?

7. Поясніть різницю між *Основним потоком (Happy Path)*, *Альтернативними потоками* та *Потоками виключень*.

8. У чому полягає принципова різниця між відносинами включення <<include>> та розширення <<extend>> на діаграмі прецедентів? Наведіть приклади.

9. Які основні правила написання якісних специфікацій прецедентів ви знаєте? Чому варто уникати деталей інтерфейсу користувача (UI) у тексті Use Case?

Блок 3. Аналіз, проєктування та реалізація

10. Опишіть призначення техніки *Аналізу стійкості (Robustness Analysis)*. Які три типи класів (**BCE**) виділяють на цьому етапі?

11. Сформулюйте правила взаємодії між класами *Boundary*, *Control* та *Entity*. Які зв'язки між ними є архітектурно забороненими?

12. Як концепція класів BCE трансформувалася у сучасні архітектурні підходи (наприклад, Clean Architecture чи MVC)?

13. Поясніть принципи *High Cohesion* та *Low Coupling* у контексті проєктування підсистем та інтерфейсів.

14. Наведіть приклади застосування патернів GoF (*Facade*, *Strategy*, *Adapter*) для структурування підсистем на етапі Design.

15. Що таке *Дисципліна реалізації* та як ідеальні UML-моделі проєктування відображаються (*mapping*) на фізичний вихідний код?

16. Яку інженерну проблему вирішує концепція *Безперервної інтеграції (CI)* та регулярних збірок (*Daily Builds*) на фазі Construction?

Блок 4. Тестування та Розгортання

17. Чому тестування в RUP вважається безперервним процесом, а не окремою фінальною фазою проєкту?

18. Опишіть ієрархію рівнів тестування в RUP (*Unit*, *Integration*, *System*). Хто відповідає за кожен із цих рівнів?

19. Як саме специфікації Use Cases використовуються для генерації тест-кейсів?

20. Опишіть стандартний життєвий цикл дефекту (*Bug Lifecycle*) від моменту виявлення до закриття.

21. У чому полягає суть процедури *Тріажу дефектів (Defect Triage)*? Чим відрізняються поняття *Severity* та *Priority*?

22. Які основні завдання виконує програма-інсталятор у класичному підході до розгортання ПЗ? Що таке проблема *«DLL Hell»*?

23. Які типи супровідної документації мають бути створені на фазі Transition і для яких аудиторій вони призначені?

24. Як класична дисципліна розгортання RUP еволюціонувала у сучасні DevOps-практики (*Docker, IaC, SaaS*)?

Блок 5. Допоміжні дисципліни, ролі та артефакти. Управління конфігураціями, змінами та проєктами

25. Що таке *Елемент конфігурації (CI)* та *Базова лінія (Baseline)* в RUP? Навіщо вони потрібні?

26. Опишіть покроковий процес обробки *Запиту на зміну (Change Request)*. Яку роль у цьому процесі відіграє Рада з управління змінами (*CCB*)?

27. Поясніть суть дворівневого планування в RUP. Чим *План фаз (Phase Plan)* відрізняється від *Плану ітерації (Iteration Plan)*?

28. Що таке *«Конус невизначеності»* і як ітеративний підхід RUP допомагає звужувати його в процесі розробки?

29. Як розраховується показник *Рівня ризику (Risk Exposure)*? Які основні стратегії роботи з ризиками використовує менеджер проєкту?

30. Яка структура та призначення головного керівного документа — *План розробки ПЗ (SDP)*?

31. Чому в RUP заборонено оцінювати прогрес проєкту на основі суб'єктивних відсотків виконання типу "код готовий на 90%"? Які об'єктивні метрики використовуються натомість?

Блок 6. Середовище, Ролі та Артефакти

32. Що таке *Адаптація процесу (Process Tailoring)* і чому RUP не можна застосовувати на проєктах «з коробки» без змін?

33. Що таке *Процес проєкту (Development Case)* та які Guidelines (інструкції) створюються в межах дисципліни «Середовище»?

34. Поясніть концепцію ролей у RUP. Чому твердження «одна роль = одна посада в штатному розкладі» є помилковим?

35. Опишіть ключові зони відповідальності та основні артефакти для чотирьох базових інженерних ролей: *Аналітика, Архітектора, Розробника* та *Тестувальника*.

36. Який життєвий цикл проходять ключові документи системи (наприклад, SRS або SAD) під час переходу проєкту від фази Inception до фази Transition?

ТЕМА 3. СТАТИЧНА СТРУКТУРА: ОСНОВНІ РОБОЧІ ДИСЦИПЛІНИ (ENGINEERING DISCIPLINES)

Якщо попередній розділ (динамічний вимір) відповідав на запитання **«Коли?»** виконується та чи інша робота, то статичний вимір RUP відповідає на запитання **«Хто, що і як робить?»**.

У цій темі ми детально розберемо 6 основних інженерних дисциплін. Важливо пам’ятати графік «Гірба»: ці процеси не виконуються суворо один за одним (як у каскадній моделі), вони відбуваються паралельно протягом усього життєвого циклу, але з різною інтенсивністю.

Лекція 9. Бізнес-моделювання (Business Modeling)

Мета лекції: Зрозуміти, чому розробка програмного забезпечення починається не з написання коду і навіть не зі збору вимог до системи, а з вивчення самого бізнесу замовника. Розглянути інструменти моделювання предметної області та зв'язок цієї класичної дисципліни із сучасними підходами, такими як Domain-Driven Design (DDD).

Навіщо інженерам моделювати бізнес?

Часта помилка молодих розробників (і технічно орієнтованих команд) полягає у відірваності програмного продукту від реалій бізнесу. Програмне забезпечення ніколи не існує у вакуумі — воно створюється для того, щоб автоматизувати, спростити або прискорити певні процеси в організації.

Якщо ми не розуміємо, *як* працює компанія, ми ризикуємо створити технічно ідеальну систему, яка буде абсолютно марною для користувачів.

Дисципліна «Бізнес-моделювання» відповідає на такі запитання:

- У чому полягає основна діяльність компанії-замовника?
- Хто є її клієнтами і яку цінність вони отримують?
- Як інформація передається між відділами?
- Де існують "вузькі місця" (bottlenecks), які можна усунути за допомогою впровадження ПЗ?

Моделі "As-Is" (Як є) та "To-Be" (Як має бути)

Робота системного або бізнес-аналітика в межах цієї дисципліни поділяється на два етапи:

1. **Моделювання "As-Is":** Опис поточного стану справ. Аналітик проводить інтерв'ю зі співробітниками та фіксує існуючі бізнес-процеси (навіть якщо вони виконуються на папері або в Excel). Це дозволяє виявити дублювання функцій та неефективність.
2. **Моделювання "To-Be":** Опис того, як виглядатимуть бізнес-процеси *після* впровадження нової інформаційної системи. Саме ця різниця між "As-Is" та "To-Be" і формує основу для майбутніх вимог до ПЗ.

Ключові артефакти бізнес-моделювання

У RUP для опису бізнесу використовується та сама мова UML, що й для проєктування архітектури, але з іншим рівнем абстракції. Створюються два головні артефакти:

А) Модель бізнес-прецедентів (Business Use Case Model) Вона показує взаємодію компанії із зовнішнім світом.

- *Бізнес-актор (Business Actor):* Хтось поза межами компанії (Клієнт, Постачальник, Податкова інспекція).
- *Бізнес-прецедент (Business Use Case):* Повний цикл надання цінності актору.
- *Приклад:* Бізнес-прецедентом для ресторану є «Обслуговування відвідувача» (від зустрічі на вході до оплати рахунку), а не «Натискання кнопки "Створити замовлення" в терміналі».

Б) Модель бізнес-об'єктів (Business Object Model / Domain Model) Вона показує "внутрішню кухню" компанії — як саме співробітники взаємодіють між собою, щоб виконати бізнес-прецедент.

- *Бізнес-працівник (Business Worker):* Роль всередині організації (Офіціант, Кухар, Менеджер).
- *Бізнес-сутність (Business Entity):* Інформація або фізичний об'єкт, з яким працюють (Меню, Замовлення, Рахунок).

Також обов'язково формується **Глосарій (Glossary)** — єдиний словник термінів. Якщо для бухгалтерії "Клієнт" — це юридична особа, а для відділу маркетингу — це відвідувач сайту, розробники повинні знати про це до початку проєктування бази даних.

Еволюція дисципліни: Від RUP до Domain-Driven Design (DDD)

У сучасній інженерії програмного забезпечення класичні артефакти бізнес-моделювання RUP трансформувалися у підхід **Domain-Driven Design (Предметно-орієнтоване проєктування)**, який є критично важливим для побудови правильної мікросервісної архітектури.

- Створення Глосарію RUP еволюціонувало у поняття **Ubiquitous Language (Єдина мова)** в DDD — набір термінів, які використовують і бізнес-експерти, і програмісти у своєму коді.
- Аналіз бізнес-процесів та виявлення "меж" між відділами (наприклад, склад і бухгалтерія) сьогодні використовується для визначення **Bounded Contexts (Обмежених контекстів)**. Саме по цих межах сучасні архітектори розбивають велику систему на незалежні мікросервіси.

Тому розуміння предметної області (Domain) залишається найважливішим кроком перед написанням будь-якого рядка коду, незалежно від того, чи працює команда за RUP, чи використовує гнучкі підходи.

Лекція 10. Управління вимогами (Requirements Management). Збір, класифікація та документування вимог. FURPS+ модель якості. Специфікація програмних вимог (SRS)

Мета лекції: Зрозуміти сутність вимог до програмного забезпечення, вивчити класифікацію за моделлю FURPS+, ознайомитися з методами збору вимог та структурою ключового документа — Специфікації програмних вимог (SRS).

Ціна помилки в управлінні вимогами

Управління вимогами — це систематичний підхід до виявлення, документування, організації та відстеження змін у вимогах до системи. Розробка програмного забезпечення без чітко визначених вимог схожа на будівництво будинку без креслень: результат буде непередбачуваним і, ймовірно, небезпечним.

Статистика інженерії програмного забезпечення демонструє, що понад 60% дефектів, знайдених на стадії тестування або після релізу, виникають не через помилки програмістів у коді, а через помилки аналітиків на етапі збору вимог.

Фінансовий вплив цих помилок описується правилом зростання вартості:

- Якщо помилка у вимогах виявлена і виправлена на етапі ініціалізації (Inception), її вартість умовна і дорівнює 1 одиниці.
- Виправлення цієї ж помилки на етапі проєктування (Elaboration) коштуватиме 5-10 одиниць.
- На етапі кодування (Construction) вартість зростає до 20-50 одиниць.
- Після розгортання системи (Transition) у користувачів виправлення архітектурної або логічної помилки, закладеної у невірній вимозі, може коштувати у 100-200 разів дорожче.

Класифікація вимог: Модель якості FURPS+

У методології RUP вимоги не обмежуються лише описом того, "що програма має робити". Для забезпечення комплексної якості продукту використовується модель **FURPS+**, розроблена компанією Hewlett-Packard. Ця модель поділяє вимоги на функціональні та нефункціональні.

- **F (Functionality) — Функціональність:** Описує дії, які система повинна вміти виконувати. Це бізнес-правила, алгоритми обчислень, обробка даних, безпека та аутентифікація. Наприклад: "Система повинна розраховувати податок на додану вартість для кожного замовлення". У RUP функціональні вимоги здебільшого фіксуються через прецеденти (Use Cases).
- **U (Usability) — Зручність використання:** Визначає ергономіку системи, вимоги до інтерфейсу користувача (UI/UX), час, необхідний на навчання нового співробітника, наявність довідкових матеріалів та відповідність стандартам доступності (наприклад, для людей з вадами зору).
- **R (Reliability) — Надійність:** Вимоги до стабільності роботи. Включає частоту допустимих збоїв, здатність системи відновлюватися після критичних помилок бази даних, точність обчислень та гарантії цілісності даних.
- **P (Performance) — Продуктивність:** Жорсткі кількісні показники. Сюди належить час відгуку системи (наприклад, "сторінка повинна завантажуватися не довше 2 секунд"), пропускна здатність ("система повинна обробляти 10 000 транзакцій на хвилину"), ефективність використання оперативної пам'яті та дискового простору.
- **S (Supportability) — Придатність до супроводу:** Наскільки легко систему тестувати, оновлювати, локалізувати іншими мовами, розширювати новим функціоналом та конфігурувати під різних клієнтів.
- **«+» (Плюс):** Додаткові обмеження (Constraints), які не вписуються в основні категорії, але є критичними для архітектора:
  - *Проєктні обмеження (Design constraints):* Вимога використовувати конкретну систему управління базами даних (наприклад, PostgreSQL).

- *Обмеження реалізації (Implementation constraints):* Жорстка вказівка щодо стека технологій (наприклад, бекенд виключно на Java, фронтенд на React).
- *Юридичні обмеження (Legal constraints):* Відповідність законодавству, ліцензійним угодам та стандартам захисту даних (наприклад, GDPR або HIPAA).

Методи виявлення (елісітації) вимог

Вимоги ніколи не даються замовником у готовому, структурованому вигляді. Системний аналітик повинен застосовувати різні методи для їх виявлення (елісітації):

- **Інтерв'ювання:** Прямі структуровані або неструктуровані бесіди з ключовими стейкхолдерами. Найкращий спосіб зрозуміти бачення керівництва.
- **Анкетування (Опитування):** Збір інформації від великої групи майбутніх користувачів, коли неможливо поговорити з кожним особисто (наприклад, опитування 500 співробітників відділу продажів).
- **Спостереження ("Тінь"):** Аналітик присутній на робочому місці користувача і мовчки фіксує, як той виконує свої щоденні завдання в існуючій системі. Це дозволяє виявити неявні процеси, про які користувач забуває розповісти на інтерв'ю.
- **Семінари з вимог (JAD-сесії):** Joint Application Design — спільні мозкові штурми за участю команди розробки, експертів предметної області та спонсорів для швидкого вирішення конфліктів у вимогах.
- **Прототипування:** Створення чорнових макетів інтерфейсу (Wireframes) або клікабельних прототипів. Користувачам набагато легше формулювати вимоги та вказувати на помилки, коли вони бачать візуалізацію, а не читають сухий текст.

Специфікація програмних вимог (SRS)

Усі виявлені, проаналізовані та узгоджені вимоги компілюються у єдиний формальний документ — **Software Requirements Specification (SRS)**. Цей документ часто розробляється на основі міжнародного стандарту IEEE 830 (або його сучасних аналогів).

Усі зібрані вимоги компілюються у головний документ — **Software Requirements Specification (SRS)** (часто базується на стандарті IEEE 830).

- *Функціональні вимоги* у RUP описуються у вигляді прецедентів (Use Cases) і є додатком до SRS.

• *Нефункціональні вимоги* (U, R, P, S та "+") жорстко прописуються в самому тексті SRS. Вони стають законом для архітектора та тестувальника навантаження.

Специфікація є юридичним та технічним контрактом між замовником і командою розробки. Вона містить:

1. **Вступ:** Мета системи, глосарій термінів, посилання на інші документи (наприклад, Vision).
2. **Загальний опис:** Характеристики користувачів, операційне середовище, загальні обмеження.
3. **Специфічні вимоги:** Детальний опис функціональних (часто через посилання на модель прецедентів) та нефункціональних (URPS+) вимог.

Матриця трасування (Traceability Matrix) Управління вимогами в RUP вимагає забезпечення трасування. Це механізм, який пов'язує вимогу з усіма артефактами життєвого циклу. Кожна вимога в документі SRS отримує унікальний ідентифікатор (наприклад, REQ-012).

Матриця трасування дозволяє простежити ланцюжок: бізнес-правило → вимога в SRS → конкретний Use Case → клас в архітектурі системи → модуль у вихідному коді → тест-кейс, який перевіряє цю вимогу. Якщо замовник вирішить змінити або видалити REQ-012, аналітик за допомогою матриці миттєво визначить, який код потрібно переписати і які тести оновити, що є основою ефективного аналізу впливу змін (Impact Analysis).

Лекція 11. Прецеденти (Use Cases) на практиці. Деталізація Use-case моделі. Написання сценаріїв прецедентів, альтернативні потоки, відносини include та extend

Мета лекції: Навчитися переходити від абстрактних графічних моделей до детальних текстових специфікацій прецедентів. Опанувати техніку написання основного та альтернативних потоків подій, а також розібрати правила застосування відносин include та extend у проєктуванні складних систем.

Від діаграми до специфікації: Анатомія прецеденту

Найпоширеніша помилка студентів та розробників-початківців полягає у переконанні, що UML-діаграма з овалами (прецедентами) та "чоловічками" (акторами) — це і є кінцева мета етапу управління вимогами. Насправді ж діаграма — це лише «зміст» вимог, візуальна мапа системи.

Справжня інженерна цінність закладена у **Специфікації прецеденту** — структурованому текстовому документі, який детально описує кожен крок взаємодії між актором та системою.

Стандартна структура специфікації прецеденту (за шаблоном RUP):

1. **Назва прецеденту:** Має бути виражена дієсловом (наприклад, *«Опублікувати статтю»*).
2. **Короткий опис:** 1-2 речення про суть процесу.
3. **Актори:**
   - *Основний:* ініціює прецедент (наприклад, *Редактор новинного порталу*).
   - *Другорядний:* зовнішні системи, до яких звертаються під час виконання (наприклад, *Зовнішній сервіс перевірки унікальності тексту*).
4. **Передумови (Pre-conditions):** Стан, у якому повинна перебувати система *до* початку виконання сценарію (наприклад, *«Користувач авторизований у системі управління контентом (CMS)»*).
5. **Постумови (Post-conditions):** Стан системи *після* успішного завершення (наприклад, *«Стаття збережена в базі даних та доступна для перегляду читачам»*).
6. **Потоки подій (Flow of Events):** Покроковий опис взаємодії (найважливіша частина).

Потоки подій: Основний, Альтернативні та Виключення

Потік подій описує діалог між Актором та Системою. Він завжди пишеться у форматі "пінг-понгу": Актор робить дію -> Система відповідає.

А) Основний потік (Basic Flow / Happy Path) Це ідеальний сценарій, коли все йде за планом, користувач вводить правильні дані, а система працює без збоїв.

Приклад для прецеденту «Опублікувати статтю на порталі»:

1. *Редактор* обирає опцію «Створити нову публікацію».
2. *Система* генерує та відображає форму текстового редактора.
3. *Редактор* вводить заголовок, текст, додає медіафайли та натискає «Опублікувати».
4. *Система* перевіряє заповненість обов'язкових полів.
5. *Система* генерує URL-адресу статті, зберігає дані в базу та оновлює кеш сторінки.
6. *Система* виводить повідомлення про успішну публікацію. Прецедент завершено.

Б) Альтернативні потоки (Alternative Flows) Це передбачені бізнес-логікою відхилення від основного шляху, які все одно призводять до успішного виконання або логічного призупинення задачі. Вони прив'язуються до конкретного кроку Основного потоку.

- *Приклад (Альтернатива до кроку 3): Редактор* натискає «Зберегти як чернетку» замість «Опублікувати». *Система* зберігає статтю зі статусом "Чернетка" без оновлення кешу публічної сторінки. Прецедент завершено.

В) Потоки виключень (Exception Flows) Це ситуації, коли нормальне виконання прецеденту стає неможливим через помилки вводу або збої. Прецедент переривається (завершується невдало).

- *Приклад (Виключення до кроку 4): Система* виявляє, що поле «Заголовок» порожнє. *Система* підсвічує поле червоним і виводить повідомлення: "Заголовок є обов'язковим". Прецедент призупиняється до виправлення помилки Редактором.

Відносини (Зв'язки) між прецедентами: include та extend

У складних системах (наприклад, корпоративних CMS або ERP-системах) багато процесів мають спільні кроки. Щоб не дублювати текст у десятках специфікацій, UML та RUP пропонують механізми структурування прецедентів через зв'язки include та extend. Це одна з найскладніших тем для розуміння на етапі проєктування.

Відношення Включення (<>) Використовується для виділення **обов'язкової, повторюваної** поведінки. Якщо базовий прецедент А має зв'язок <<include>> до прецеденту Б, це означає, що А *не може* бути виконаний без Б. Базовий прецедент на певному кроці передає управління включеному прецеденту, а після його завершення — продовжує свою роботу.

- **Мета:** Уникнення дублювання коду та вимог (аналог виклику підпрограми/функції у програмуванні).
- **Приклад:** Прецеденти *«Опублікувати статтю»* та *«Видалити коментар»* обов'язково включають прецедент *«Авторизуватися в системі»*. Без авторизації жодна з цих дій неможлива.

Відношення Розширення (<>) Використовується для моделювання **необов'язкової, специфічної або умовної** поведінки. Прецедент, що розширює (Б), додає нові кроки до базового прецеденту (А), але тільки якщо виконується певна умова (Точка розширення - Extension Point). Базовий прецедент може бути

успішно виконаний і без цього розширення; він "не знає" про існування розширювального прецеденту.

- **Мета:** Відокремлення складного опціонального функціоналу від основного сценарію, щоб не перевантажувати його.
- **Приклад:** Базовий прецедент *«Опублікувати статтю»*. Розширювальний прецедент — *«Налаштувати SEO-параметри»* (прописати мета-теги, ключові слова). Редактор може просто опублікувати текст (базовий сценарій), але за бажанням може викликати розширення для оптимізації статті для пошукових систем.

Правила якісної специфікації (Best Practices)

Для створення документації, яка буде зрозумілою як замовнику, так і розробникам, слід дотримуватися таких правил:

- **Пишіть з точки зору користувача, а не бази даних.** Уникайте технічного жаргону на кшталт *"Система виконує запит INSERT INTO articles..."*. Замість цього пишіть *"Система зберігає статтю"*.
- **Один крок — одна дія.** Не об'єднуйте дії актора та відповідь системи в одне речення.
- **Не використовуйте прецеденти для опису інтерфейсу (UI).** Прецедент має бути стійким до змін дизайну. Пишіть *"Актор підтверджує вибір"*, а не *"Актор натискає зелену круглу кнопку в правому нижньому куті"*. Інтерфейси змінюються часто, бізнес-логіка — ні.
- **Уникайте надмірного використання include та extend.** Якщо ваша діаграма стає схожою на павутиння і нагадує блок-схему алгоритму — ви робите функціональну декомпозицію, що суперечить об'єктно-орієнтованому підходу RUP. Прецеденти мають відображати цінність для користувача, а не ієрархію функцій.

Лекція 12. Аналіз та проєктування (Analysis & Design) – Частина 1. Перехід від вимог до концептуальної моделі. Виділення класів аналізу (Boundary, Control, Entity)

Мета лекції: Показати механізм переходу від текстових специфікацій прецедентів (вимог) до об’єктно-орієнтованої архітектури системи. Ознайомитися з патерном Аналізу стійкості (Robustness Analysis) та класифікацією класів на Граничні (Boundary), Керуючі (Control) та Сутності (Entity).

Проблема розриву між вимогами та кодом

Головна проблема, з якою стикаються розробники після завершення збору вимог, — це прірва між мовою бізнесу та мовою програмування.

- **Вимоги (Use Cases)** описують систему в термінах дій користувача: «Редактор натискає кнопку, система зберігає статтю».
- **Код (Реалізація)** описує систему в термінах об'єктів: ArticleRepository, UserSession, PostController.

Спроба одразу писати код (або малювати детальні діаграми класів бази даних) на основі тексту прецеденту майже завжди призводить до створення монолітної, негнучкої архітектури. Щоб подолати цю прірву, RUP запроваджує проміжний етап — **Аналіз (Analysis)**, результатом якого є створення концептуальної моделі (моделі аналізу).

Модель аналізу — це ідеалізована архітектура системи. Вона абстрагується від конкретних технологій (баз даних, фреймворків чи мов програмування) і фокусується виключно на тому, *які компоненти* потрібні для виконання бізнес-логіки.

Аналіз стійкості (Robustness Analysis) та Патерн BCE

Для переходу від тексту прецеденту до об'єктів використовується техніка, запропонована Іваром Якобсоном — Аналіз стійкості. Вона допомагає перевірити текст прецеденту на повноту та виділити початковий набір класів.

Згідно з цією технікою, всі об'єкти в системі поділяються на три суворо визначені типи (стереотипи UML). Ця концепція отримала назву **BCE (Boundary, Control, Entity)**.

А) Граничні класи (Boundary Classes)

- **Відповідальність:** Граничні класи є посередниками між зовнішнім світом (Акторами) та внутрішньою логікою системи. Вони відповідають за прийом запитів та видачу результатів.
- **Приклади:** Екрани користувацького інтерфейсу (UI), вебформи, форми авторизації. Якщо взаємодія відбувається між двома системами, граничним класом буде API-шлюз (API Gateway) або адаптер для зовнішнього сервісу.
- **Правило:** Актори можуть взаємодіяти *лише* з граничними класами.

Б) Класи-сутності (Entity Classes)

- **Відповідальність:** Це класи, що зберігають інформацію (дані) та базову поведінку, пов'язану з цими даними. Це довгоживучі об'єкти, які зазвичай зберігаються в базі даних.
- **Приклади:** Користувач, Стаття, Рахунок, Договір.
- **Правило:** Сутності нічого не знають про інтерфейс (Boundary). Їхня задача — зберігати цілісність власних даних (наприклад, сутність Рахунок містить метод перевірки позитивного балансу).

В) Керуючі класи (Control Classes)

- **Відповідальність:** Це координатори. Вони містять бізнес-логіку прецеденту. Керуючий клас отримує команду від Граничного класу, вирішує, які класи-сутності потрібно задіяти, і керує їхньою взаємодією.
- **Приклади:** МенеджерПублікацій, КонтролерАвторизації, ПроцесорТранзакцій. Зазвичай на один Use Case створюється один керуючий клас.
- **Правило:** Вони є «клеєм», що з'єднує інтерфейс та дані, забезпечуючи розв'язання (decoupling) системи.

Правила взаємодії об'єктів BCE

Щоб архітектура була «стійкою» (гнучкою до змін), RUP вводить жорсткі правила взаємодії між цими трьома типами класів. Ці правила легко запам'ятати:

1. **Актори** можуть спілкуватися тільки з **Boundary**.
2. **Boundary** можуть спілкуватися тільки з **Control** (інтерфейс не має прямого доступу до бази даних).
3. **Control** можуть спілкуватися з іншими **Control**, **Boundary** та **Entity**.
4. **Entity** можуть спілкуватися тільки з **Entity** (дані не можуть керувати логікою або відображати інтерфейс).

Якщо аналітик малює діаграму і бачить, що Граничний клас (форма на сайті) безпосередньо звертається до Сутності (таблиці в БД) — це грубе порушення архітектури, яке призведе до вразливостей і проблем при зміні дизайну.

Еволюція патерну: Від класичного RUP до сучасної архітектури

При оновленні навчальних програм та переході від класичних методологій розробки до сучасних практик Software Architecture, DevOps та AI Engineering,

надзвичайно важливо продемонструвати студентам, що концепція класів аналізу нікуди не зникла, а лише адаптувалася до нових реалій.

Патерн BCE є прямим попередником найпопулярніших сучасних архітектурних рішень:

- **Модель-Вигляд-Контролер (MVC):** Це найочевидніша еволюція. Boundary трансформувалися у *View* (Вигляд), Control стали *Controllers* (Контролерами), а Entity — це моделі даних (*Model*).
- **Чиста архітектура (Clean Architecture / Onion Architecture):** У сучасних багатошарових архітектурах поділ зберігається ще чіткіше. Boundary — це шар Presentation та зовнішні адаптери. Control — це Use Case Interactors у шарі Application. Entity — це Domain Entities у самому серці архітектури (Domain Layer).
- **Мікросервіси:** При проєктуванні мікросервісів межі компонентів часто визначаються навколо агрегацій сутностей (Entity), а API-шлюзи (API Gateways) виконують роль сучасних граничних класів (Boundary), приховуючи керуючу логіку мікросервісу.

Перехід від прецедентів до моделі аналізу BCE створює надійний "ескіз" системи. У другій частині теми «Аналіз та проєктування» цей ескіз буде перетворено на точне креслення з використанням патернів GoF (Gang of Four) та детальним проєктуванням підсистем.

Лекція 13. Аналіз та проєктування (Analysis & Design) – Частина 2. Проєктування підсистем та інтерфейсів. Застосування патернів проєктування (GoF) та відображення архітектури через UML-діаграми

Мета лекції: Навчитися переходити від ідеалізованої концептуальної моделі до реального технічного проєкту. Зрозуміти принципи декомпозиції системи на підсистеми, роль інтерфейсів як контрактів, а також освоїти застосування класичних патернів проєктування (GoF) та UML-діаграм для документування архітектурних рішень.

Від Аналізу до Проєктування (Від "Що" до "Як")

На попередній лекції ми створили модель аналізу (класи Boundary, Control, Entity). Ця модель відповідала на запитання: *«Які логічні блоки потрібні для виконання бізнес-задачі?»*, при цьому вона повністю ігнорувала технології.

Етап **Проєктування (Design)** бере цю ідеальну модель і "зіштовхує" її з реальністю. На цьому етапі архітектор повинен врахувати:

- Обрану мову програмування та фреймворки (наприклад, Java + Spring або C# + .NET).
- Тип бази даних (реляційна PostgreSQL чи NoSQL MongoDB).
- Мережеві обмеження, безпеку та вимоги до продуктивності (URPS+).

Граничний клас ФормаАвторизації з етапу аналізу тепер перетворюється на конкретний LoginController з методами обробки HTTP-запитів, а клас-сутність Користувач отримує ORM-анотації для мапінгу в таблицю бази даних.

Проєктування підсистем та Інтерфейсів

Для складних систем наявність сотень окремих класів створює хаос. Щоб цього уникнути, RUP вимагає групувати класи у більші будівельні блоки — **Підсистеми (Subsystems)**.

Правила формування підсистем: Усі класи всередині однієї підсистеми повинні бути об'єднані єдиною метою. Тут застосовуються два фундаментальні принципи програмної інженерії:

- **High Cohesion (Висока згуртованість):** Усередині підсистеми класи тісно взаємодіють один з одним і вирішують одну спільну задачу (наприклад, підсистема "Біллінг" містить усі класи, що стосуються оплат, інвойсів та податків).
- **Low Coupling (Низька зв'язність):** Підсистеми повинні мінімально залежати одна від одної. Зміна логіки в "Біллінгу" не повинна зламати підсистему "Управління складом".

Інтерфейси як контракти: Щоб забезпечити низьку зв'язність, підсистеми взаємодіють між собою *виключно* через інтерфейси.

- **Інтерфейс** — це контракт (набір публічних методів), який підсистема зобов'язується виконувати, повністю приховуючи свою внутрішню реалізацію (інкапсуляція).
- Якщо підсистемі "Кошик" потрібно дізнатися ціну товару, вона не звертається до таблиці БД підсистеми "Каталог товарів". Вона викликає метод getPrice() через публічний інтерфейс ICatalog. Це дозволяє в майбутньому повністю переписати "Каталог", не зачіпаючи "Кошик", за умови, що інтерфейс залишається незмінним.

Застосування патернів проєктування (GoF)

На етапі проєктування архітектори стикаються з типовими проблемами: як створити об'єкт, як об'єднати об'єкти в структуру, як організувати комунікацію між ними. Замість того, щоб щоразу "винаходити велосипед", використовуються перевірені часом рішення — **Патерни проєктування**.

Найвідомішим каталогом є патерни **«Банди чотирьох» (Gang of Four - GoF)**. У контексті RUP вони ідеально лягають на класи аналізу:

А) Твірні патерни (Creational Patterns): Вирішують проблему безпечного створення об'єктів.

- *Factory Method (Фабричний метод) / Abstract Factory:* Часто використовується для створення складних класів-сутностей (Entity), коли їх конфігурація залежить від бізнес-логіки.
- *Singleton (Одинак):* Використовується для класів, які повинні існувати в єдиному екземплярі на всю систему (наприклад, з'єднання з базою даних або глобальний конфігуратор середовища).

Б) Структурні патерни (Structural Patterns): Допомагають будувати великі системи з окремих класів.

- *Facade (Фасад):* Ідеально підходить для проєктування підсистем. Фасад слугує єдиною "точкою входу" (тим самим Інтерфейсом) до складної підсистеми, приховуючи її заплутану внутрішню структуру.
- *Adapter (Адаптер):* Типовий патерн для Граничних класів (Boundary), яким потрібно спілкуватися із зовнішніми, застарілими (legacy) системами або сторонніми API.

В) Поведінкові патерни (Behavioral Patterns): Описують, як об'єкти розподіляють між собою обов'язки.

- *Observer (Спостерігач):* Основа для проєктування інтерфейсів користувача (Boundary) та систем сповіщень. Якщо стан Сутності змінився, усі підписані Граничні класи автоматично оновлюються.
- *Strategy (Стратегія):* Дозволяє інкапсулювати різні алгоритми у Керуючих класах (Control). Наприклад, різні алгоритми розрахунку знижки можна підміняти "на льоту" без зміни основного коду контролера.

Відображення архітектури через UML-діаграми

Архітектура має бути задокументована. У класичному RUP для цього використовуються три ключові діаграми UML, які формують *Логічний вигляд* (Logical View) архітектурної моделі "4+1":

1. **Діаграма класів (Class Diagram):** Це статичне креслення системи. На відміну від моделі аналізу, проєктна діаграма класів деталізована до рівня коду. Вона відображає:
   - Атрибути об'єктів (з їх типами даних: String, int).
   - Методи (із вхідними параметрами та типом повернення).
   - Модифікатори доступу (+ public, - private, # protected).
   - Типи зв'язків: Асоціація, Агрегація, Композиція та Успадкування.
2. **Діаграма послідовності (Sequence Diagram):** Це динамічний вигляд системи. Вона перекладає текстовий "Сценарій прецеденту" на мову викликів методів.
   - Діаграма читається зверху вниз (шкала часу).
   - Показує, як Актор викликає метод Граничного класу, як той передає повідомлення (виклик функції) Керуючому класу, і як Керуючий клас виконує CRUD-операції з Сутністю. Ця діаграма доводить, що архітектура дійсно здатна виконати Use Case.
3. **Діаграма компонентів (Component Diagram):** Використовується для візуалізації архітектури на найвищому рівні — рівні підсистем, бібліотек та виконуваних файлів (.dll, .jar). Вона ілюструє залежності між великими блоками коду.

Еволюція концепцій: Від RUP до API-First Design

Сучасні підходи до розробки багато в чому базуються на цих фундаментальних концепціях етапу Design:

- **API-First Design:** Принцип проєктування "інтерфейси як контракти" еволюціонував у сучасний підхід розробки API (наприклад, через Swagger/OpenAPI специфікації). Спочатку проєктується контракт взаємодії (API), і лише потім фронтенд та бекенд команди починають паралельну реалізацію.
- **Мікросервісна архітектура:** Модель декомпозиції системи на підсистеми з жорсткими інтерфейсами — це прямий прототип сучасних мікросервісів. Тільки сьогодні підсистеми розгортаються не як класи в одному моноліті, а як окремі контейнери (Docker), що спілкуються через мережу (REST або gRPC).

• **SOLID:** Патерни GoF та правила проєктування класів у RUP є практичним втіленням принципів SOLID (особливо принципу єдиної відповідальності та інверсії залежностей).

Лекція 14. Дисципліна реалізації (Implementation). Організація вихідного коду, стандарти кодування, збірка системи. Взаємозв'язок з архітектурною моделлю

Мета лекції: Вивчити принципи та завдання дисципліни Реалізації у життєвому циклі RUP. Зрозуміти, як ідеальна модель проєктування трансформується у фізичний вихідний код, ознайомитися з правилами організації кодової бази, важливістю стандартів кодування та процесами збірки (інтеграції) системи.

Сутність та завдання дисципліни «Реалізація»

У багатьох неформальних методологіях розробка починається безпосередньо з написання коду. У методології RUP процес програмування (Implementation) є логічним продовженням і наслідком дисципліни «Аналіз та проєктування».

Реалізація — це дисципліна, що відповідає за перетворення абстрактних архітектурних моделей (UML-діаграм, класів, інтерфейсів) на фізично існуючі компоненти: файли з вихідним кодом, скомпільовані бібліотеки, виконувані файли та бази даних.

Основні цілі дисципліни:

1. Організувати структуру вихідного коду (ієрархію файлів та каталогів).
2. Реалізувати класи та об'єкти мовою програмування відповідно до проєктної моделі.
3. Провести модульне тестування (Unit Testing) розроблених компонентів.
4. Інтегрувати результати роботи окремих розробників або команд у єдину виконувану систему (збірка системи).

Відповідно до графіка «Гірба» (RUP Hump Chart), ця дисципліна починає активно виконуватися на фазі *Elaboration* (створення базової виконуваної архітектури) і досягає свого абсолютного максимуму на фазі *Construction* (масове нарощування функціоналу).

Взаємозв'язок з архітектурною моделлю (Traceability)

Одним із ключових принципів інженерії програмного забезпечення є трасування (Traceability) — здатність простежити зв'язок між вимогами, архітектурою та кодом. Код не повинен існувати окремо від архітектури.

Правила відображення (Mapping) проєкту на код:

- **Підсистеми проєктування (Design Subsystems)** перетворюються на фізичні директорії (папки), пакети (Packages у Java), простори імен (Namespaces у C#/C++) або окремі репозиторії.
- **Класи проєктування (Design Classes)** перетворюються на фізичні файли з вихідним кодом (наприклад, UserController.java або Article.cs).
- **Інтерфейси (Interfaces)** зберігаються як контракти в коді (наприклад, ключове слово interface в ООП) і використовуються для зв'язку між створеними пакетами.

Якщо розробник під час кодування розуміє, що проєктна модель була неповною або неефективною, він не має права просто "зламати" архітектуру в коді. У RUP зміни повинні бути двоспрямованими: зміна в коді повинна відобразитися в UML-моделі (цей процес підтримується сучасними CASE-засобами через механізми Reverse Engineering).

Організація вихідного коду

Коли над проєктом працюють десятки програмістів, структура файлів не може бути хаотичною. Організація кодової бази (Codebase) повинна відображати логічну архітектуру системи.

Принципи організації коду:

1. **Шарувата структура (Layering):** Код фізично розділяється на шари (Presentation, Business Logic, Data Access). Забороняється зберігати SQL-запити до бази даних у файлах, що відповідають за відображення інтерфейсу (UI).
2. **Принцип ізоляції компонентів:** Кожен розробник працює у своєму «робочому просторі» (Workspace). Він змінює лише ті компоненти, за які відповідає, а інші компоненти використовує лише через їхні публічні інтерфейси.
3. **Управління залежностями:** Фізичні залежності між файлами (наприклад, директиви import або include) мають жорстко контролюватися. Не повинно бути циклічних залежностей (коли пакет А залежить від пакета Б, а пакет Б імпортує класи з пакета А).

Стандарти кодування та забезпечення якості

Професійна розробка відрізняється від аматорської тим, що код пишеться не лише для того, щоб його виконав комп'ютер, але й для того, щоб його могли читати інші програмісти.

"Код читають у 10 разів частіше, ніж пишуть" (Роберт Мартін, «Чистий код»).

Елементи стандартизації:

- **Угоди про іменування (Naming Conventions):** Затверджені правила найменування змінних, методів та класів (наприклад, класи записуються у PascalCase, а змінні у camelCase).
- **Коментування та документація:** Обов'язкове використання інструментів генерації документації з коду (наприклад, Javadoc, Doxygen або Swagger для API).
- **Рефакторинг:** Регулярне покращення внутрішньої структури коду без зміни його зовнішньої поведінки для усунення «технічного боргу» (Technical Debt).
- **Code Review (Огляд коду):** Жоден рядок коду не потрапляє до основної гілки без перевірки іншим (часто більш досвідченим) розробником. Це головний інструмент підтримки стандартів на практиці.

Збірка системи (Build) та Безперервна інтеграція

Написання коду окремими розробниками — це лише половина справи. Головний виклик фази Побудови — змусити всі ці розрізнені шматки коду працювати разом. Цей процес називається **Інтеграцією.**

Традиційна (каскадна) модель передбачала розробку ізольованих модулів протягом місяців, і лише перед фінальним релізом намагалася зібрати їх разом. Це завжди призводило до «Інтеграційного пекла» (Integration Hell) — фази, коли нічого не працює і ніхто не знає чому.

Підхід RUP до інтеграції: RUP запровадив принцип **поступової, безперервної інтеграції.**

1. Розробник пише код та модульні тести (Unit Tests).
2. Розробник інтегрує свій код з кодом колег (наприклад, щодня або кілька разів на тиждень).

3. Відбувається **Збірка (Build)** — автоматизований процес компіляції вихідного коду, зв'язування бібліотек та створення виконуваного артефакту (системи).

4. Після кожної збірки автоматично запускаються інтеграційні тести. Якщо збірка «зламана» (Broken Build), уся команда зупиняє розробку нових функцій і фокусується на виправленні помилки.

Регулярні збірки (Daily Builds) дозволяють виявляти конфлікти інтерфейсів та логічні помилки на найраніших стадіях, роблячи процес розробки передбачуваним.

Еволюція дисципліни: Від RUP до CI/CD

Дисципліна Реалізації в RUP заклала фундамент для сучасних практик інженерії програмного забезпечення. Те, що у 2000-х роках вимагало написання складних ручних скриптів (Makefile або Ant), сьогодні повністю автоматизовано:

- **Контроль версій:** Механізми робочих просторів еволюціонували у розподілені системи контролю версій (насамперед **Git**) з їхніми гілками розробки (Feature branches).
- **Статичний аналізатор коду:** Перевірка стандартів кодування тепер виконується не лише людьми на Code Review, а й автоматичними лінтерами (ESLint, SonarQube), які блокують збірку, якщо код не відповідає конвенціям.
- **DevOps та CI/CD:** Концепція регулярної збірки перетворилася на пайплайни (Pipelines) **Безперервної інтеграції та Безперервного розгортання (Continuous Integration / Continuous Delivery)**. Сьогодні код збирається, тестується і пакується у Docker-контейнери автоматично при кожному git push, що є прямим технологічним спадкоємцем ідей інтеграції, закладених у Rational Unified Process.

Лекція 15. Дисципліна тестування (Testing). Рівні тестування в RUP (Unit, Integration, System). Тестування на основі прецедентів. Управління дефектами

Мета лекції: Зрозуміти місце та роль тестування в ітеративному процесі розробки RUP. Дослідити ієрархію рівнів тестування, навчитися генерувати тестові сценарії безпосередньо з прецедентів (Use Cases) та розібрати життєвий цикл управління дефектами.

Філософія тестування у методології RUP

У застарілих каскадних (Waterfall) моделях тестування було окремою фазою, яка починалася лише після того, як програмісти повністю закінчували писати код. Це призводило до катастрофічних наслідків: фундаментальні архітектурні дефекти виявлялися за тиждень до релізу, коли бюджет і час вже були вичерпані.

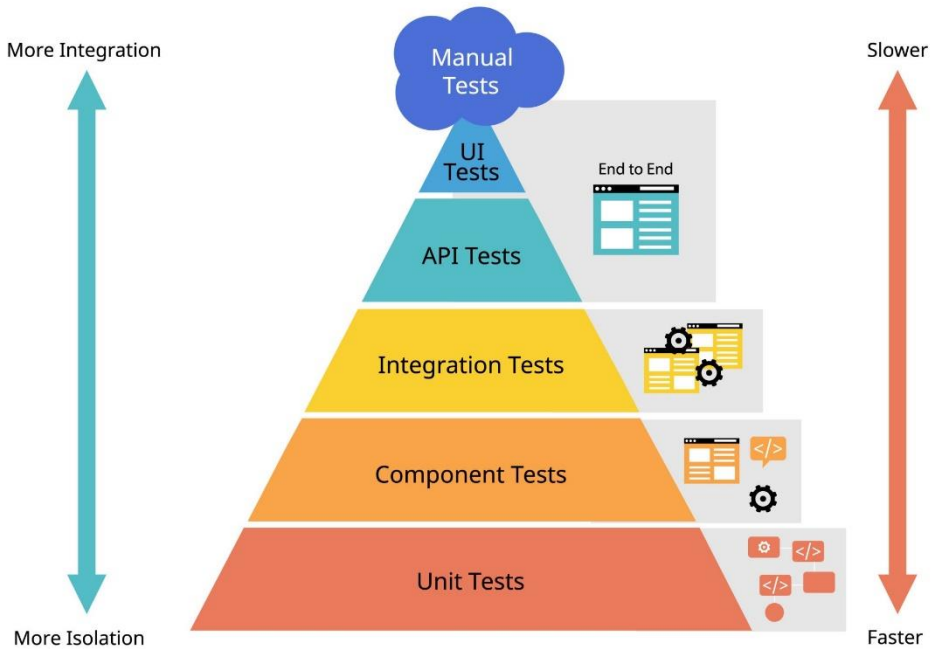
RUP кардинально змінив цю парадигму. **Тестування в RUP — це не фаза, це безперервна дисципліна.** Вона починається ще на етапі Inception (коли тестувальники аналізують вимоги на предмет їхньої "тестованості") і триває до самого релізу.

Головні цілі дисципліни:

1. Знайти та задокументувати дефекти якомога раніше (чим раніше знайдено дефект, тим дешевше його виправити).
2. Об'єктивно оцінити якість продукту та ризики його впровадження.
3. Довести, що розроблена система дійсно відповідає первинним вимогам замовника (верифікація та валідація).

Рівні тестування (Hierarchy of Testing)

RUP пропагує багаторівневий підхід до перевірки якості. Неможливо протестувати всю складну систему за один раз. Перевірка йде знизу вгору — від найдрібніших компонентів до системи в цілому.



А) Модульне тестування (Unit Testing)

- **Хто виконує:** Програмісти (Розробники).
- **Що тестується:** Найменші неподільні частини коду — окремі класи, методи або функції.
- **Специфіка:** Це тестування "білого ящика" (White-box testing). Розробник бачить внутрішню структуру коду. Мета — переконатися, що метод calculateTax() дійсно повертає правильне математичне значення для різних вхідних параметрів. Модульні тести мають бути максимально швидкими та автоматизованими.

Б) Інтеграційне тестування (Integration Testing)

- **Хто виконує:** Розробники разом із QA-інженерами.
- **Що тестується:** Взаємодія між підсистемами, класами або зовнішніми сервісами.
- **Специфіка:** Навіть якщо два модулі ідеально працюють окремо, вони можуть конфліктувати при спробі обмінятися даними. Наприклад, підсистема "Кошик" відправляє ціну у форматі Float, а підсистема "Оплата" очікує формат Integer. Інтеграційне тестування перевіряє контракти (інтерфейси) між компонентами.

В) Системне тестування (System Testing)

- **Хто виконує:** Незалежні QA-інженери (Тестувальники).
- **Що тестується:** Повністю зібрана система в середовищі, максимально наближеному до реального.
- **Специфіка:** Це тестування "чорного ящика" (Black-box testing). Тестувальник не дивиться в код. Він імітує дії кінцевого користувача: натискає кнопки, заповнює форми та перевіряє, чи відповідає поведінка системи документації (SRS). Окрім функціоналу, тут перевіряються нефункціональні вимоги (продуктивність, безпека, навантаження).

Тестування на основі прецедентів (Use-Case Based Testing)

Згадаємо перший принцип RUP — "Керованість прецедентами" (Use-case driven). Прецеденти не просто описують вимоги; вони є прямим джерелом для створення тестових сценаріїв (Test Cases).

Це гарантує **Трасування (Traceability)**: якщо система успішно проходить тест, згенерований з Use Case, це є доказом того, що бізнес-вимога замовника реалізована.

Як перетворити Use Case на Test Case? Аналітик написав прецедент *«Оформити замовлення»*. QA-інженер бере його специфікацію та генерує набір тестів:

1. **Тестування Основного потоку (Happy Path):** Створюється позитивний тест-кейс. Користувач вводить валідні дані картки, система знімає гроші та показує повідомлення про успіх. Якщо цей тест падає, подальше тестування зупиняється — функціонал зламаний.
2. **Тестування Альтернативних потоків:** Створюються тест-кейси для передбачених бізнес-логікою відхилень. *Тест:* Користувач оформлює замовлення, але замість оплати карткою обирає "Оплата при отриманні".
3. **Тестування Потоків виключень (Negative Testing):** Найважливіша частина роботи QA. Перевіряється, як система реагує на помилки. *Тести:* Користувач вводить прострочену картку; користувач вводить літери в поле "Номер телефону"; зникає зв'язок з банківським API під час транзакції. Система не повинна падати (Crash), вона має коректно обробити помилку і вивести зрозуміле повідомлення.

Управління дефектами (Defect Management)

Коли тест провалено, тестувальник створює звіт про помилку (Bug Report). Управління дефектами — це формалізований процес відстеження життєвого циклу помилки від її виявлення до остаточного виправлення.

Типовий життєвий цикл дефекту (Bug Lifecycle):

1. **New (Новий):** Тестувальник знайшов помилку і задокументував її (кроки для відтворення, очікуваний та фактичний результат, скріншоти).
2. **Assigned (Призначений):** Менеджер проєкту або Team Lead перевіряє помилку і призначає її конкретному розробнику.
3. **In Progress (В роботі):** Розробник пише код для виправлення дефекту.
4. **Fixed / Resolved (Виправлено):** Розробник стверджує, що помилку усунуто, і код передано в тестове середовище.
5. **Ready for Retest (Готово до перевірки):** М'яч знову на боці QA. Тестувальник повторює ті самі дії, що й у п.1.
6. **Closed (Закрито):** Якщо помилка дійсно зникла, дефект закривається.
   - *Або* **Reopened (Перевідкрито):** Якщо помилка все ще відтворюється, вона повертається розробнику зі зниженням метрик якості.

Тріаж дефектів (Defect Triage): У великих проєктах неможливо виправити всі помилки. Команда регулярно збирається на процедуру Тріажу (сортування), щоб оцінити кожен баг за двома шкалами:

- **Severity (Серйозність):** Наскільки сильно баг ламає систему технічно (від "Критичного" падіння бази даних до "Мінорного" зміщення кнопки на 2 пікселі).
- **Priority (Пріоритет):** Наскільки швидко бізнес вимагає це виправити. Наприклад, орфографічна помилка на головній сторінці сайту (Severity = Low) матиме найвищий пріоритет (Priority = Blocker) через репутаційні ризики.

Трансформація дисципліни: Від RUP до сучасних практик

Принципи тестування, закладені в RUP, є повністю актуальними сьогодні, але інструментарій та швидкість кардинально змінилися:

- **Автоматизація (Test Automation):** Якщо раніше системне тестування виконувалося переважно вручну ("мавпяча праця"), то сьогодні регресійне тестування (перевірка того, що новий код не зламав старий) на 90% виконується автоматизованими скриптами (Selenium, Cypress, JUnit).
- **Shift-Left Testing:** Сучасна тенденція "зсуву вліво" поширює ідею RUP про раннє тестування ще далі. Тестувальники починають писати автоматизовані тести ще до того, як розробники написали код.
- **Test-Driven Development (TDD):** Практика, за якої програміст спочатку пише Unit-тест (який закономірно падає), а лише потім пише мінімально необхідний код, щоб тест пройшов успішно. Це гарантує 100% покриття архітектури тестами.

Загальні висновки до теми 3:

Тема 3 детально окреслила інженерне серце методології Rational Unified Process (RUP) через призму шести основних робочих дисциплін. Якщо динамічний вимір (Тема 2) визначав часові межі та фази, то статична структура дає чітку відповідь на запитання: **«Які конкретно технічні процеси трансформують абстрактну бізнес-ідею у працюючий програмний продукт на серверах замовника?»**.

Аналіз дисциплін — від бізнес-моделювання до розгортання — дозволяє сформувати кілька фундаментальних висновків, які визначають успіх сучасної інженерії програмного забезпечення:

- **Принцип безперервного конвеєра (Анти-Waterfall):** Жодна з шести інженерних дисциплін не є ізольованим етапом. Вони виконуються паралельно

протягом усього проєкту, взаємодіючи та підживлюючи одна одну. Аналітики продовжують уточнювати вимоги, поки архітектори проєктують, а тестувальники починають роботу одночасно з появою перших чернеток Use Cases, що ламає класичну каскадну лінійність.

- **Наскрізна трасованість (Traceability) як гарантія якості:** RUP створює жорсткий, логічно обґрунтований ланцюжок трансформації артефактів. Бізнес-процес народжує вимогу в SRS; вимога розгортається у текстовий сценарій прецеденту (Use Case); прецедент перекладається мовою архітектурних класів аналізу (Boundary-Control-Entity); класи матеріалізуються у фізичний код, який покривається Unit-тестами та системними тест-кейсами, створеними на основі тих самих Use Cases. Цей наскрізний зв'язок унеможливлює ситуацію, коли команда пише код, який нікому не потрібен, або забуває протестувати критичну бізнес-функцію.
- **Архітектурна стійкість (Robustness):** Поділ системи на класи аналізу (BCE) та подальше проєктування підсистем через інтерфейси-контракти та патерни GoF забезпечують виконання принципів *High Cohesion* та *Low Coupling*. Це робить систему гнучкою до неминучих змін: інтерфейс (Boundary) можна повністю перемалювати, а базу даних (Entity) змінити, не торкаючись при цьому стрижневої бізнес-логіки (Control).

Матриця інженерних дисциплін: Від класичного RUP до сучасних практик

Для забезпечення швидкої сканованості матеріалу, взаємозв'язок між класичними підходами RUP та сучасним індустріальним стеком можна узагальнити у такій таблиці:

Інженерна дисципліна RUP	**Головне завдання (Ціль)**	**Ключовий артефакт**	**Сучасна трансформація в IT-індустрії**
1. Бізнес-моделювання	Зрозуміти предметну область замовника, виявити "вузькі місця" процесу.	Business Use Cases, Domain Model, Glossary	**Domain-Driven Design (DDD)**, Ubiquitous Language, Bounded Contexts.
2. Управління вимогами	Зібрати та класифікувати вимоги до ПЗ, зафіксувати обмеження.	Специфікація вимог (**SRS**), Модель FURPS+	**Product Discovery**, User Stories в Agile (Jira/Confluence).
3. Аналіз та	Створити концептуальну	Виконувана	**Clean Architecture**, Onion

Інженерна дисципліна RUP	Головне завдання (Ціль)	Ключовий артефакт	Сучасна трансформація в IT-індустрії
проєктування	та технічну модель системи, обрати технології.	архітектура, **SAD** (UML-діаграми)	Architecture, Microservices, REST/gRPC API.
4. Реалізація	Перетворити моделі на код, організувати структуру бази коду, зібрати систему.	Вихідний код, Модульні тести, **Build**	Чистий код (SOLID), Рефакторинг, **Git**, Daily Builds еволюціонували в **CI пайплайни**.
5. Тестування	Об'єктивно оцінити якість та знайти дефекти на ранніх стадіях.	Тест-план, Тест-кейси, Bug Reports	**QA Automation** (Selenium, Cypress), TDD (Test-Driven Development), Shift-Left Testing.
6. Розгортання	Доставити ПЗ користувачу, мігрувати дані, навчити персонал.	Дистрибутив (Інсталятор), User & Admin Guides	**DevOps**, Контейнеризація (**Docker**, Kubernetes), **IaC** (Terraform, Ansible), SaaS.

Контрольні запитання до Теми 3

1. Дисципліна «Бізнес-моделювання» (Business Modeling)

1. У чому полягає головна мета дисципліни бізнес-моделювання в RUP? Чому розробка великих систем не може починатися безпосередньо зі збору технічних вимог?
2. Дайте визначення та опишіть практичну цінність моделювання процесів у форматах **"As-Is"** («Як є») та **"To-Be"** («Як має бути»).
3. Опишіть компоненти **Моделі бізнес-прецедентів (Business Use Case Model)**. Чим *Бізнес-актор* відрізняється від *Системного актора*?
4. Які елементи складають **Модель бізнес-об'єктів (Business Object Model)**? Поясніть ролі *Business Worker* та *Business Entity*.
5. Яке призначення *Глосарію (Glossary)* предметної області?
6. Яким чином класичне бізнес-моделювання RUP трансформувалося в сучасний підхід **Domain-Driven Design (DDD)**? Поясніть концепції *Ubiquitous Language* та *Bounded Contexts*.

2. Дисципліна «Управління вимогами» (Requirements Management)

7. Як змінюється вартість виправлення помилки у вимогах залежно від фази життєвого циклу проєкту (від *Inception* до *Transition*)? Опишіть фінансові ризики.

8. Детально розкрийте метрики якості моделі **FURPS+**. Які вимоги належать до нефункціональних, і що саме ховається під символом «+»?

9. Які основні методи елісітації (виявлення) вимог застосовує системний аналітик? У чому перевага *JAD-сесій* та *Спостереження ("Тінь")*?

10. Яка структура та призначення документа **SRS (Software Requirements Specification)** відповідно до стандарту IEEE 830?

11. Як працює **Матриця трасування (Traceability Matrix)**? Яким чином вона забезпечує проведення аналізу впливу змін (*Impact Analysis*)?

12. З яких обов'язкових розділів складається текстова специфікація прецеденту (*Use Case Specification*)?

13. Опишіть різницю між *Основним потоком (Happy Path)*, *Альтернативними потоками* та *Потоками виключень (Exception Flows)*.

14. Поясніть різницю між UML-зв'язками **<<include>>** та **<<extend>>**. Наведіть приклади, коли використання <<extend>> є архітектурно виправданим.

15. Порівняйте інструменти *Use Cases* (RUP) та *User Stories* (Agile). Для яких типів проєктів доцільно використовувати кожен із них?

3. Дисципліна «Аналіз та проєктування» (Analysis & Design)

16. Що таке *Модель аналізу (Analysis Model)* і як вона допомагає подолати прірву між текстом вимог та вихідним кодом?

17. Розкрийте суть техніки *Аналізу стійкості (Robustness Analysis)* та патерну **BCE (Boundary, Control, Entity)**.

18. Сформулюйте суворі правила взаємодії об'єктів у патерні BCE. Які типи зв'язків є архітектурними порушеннями?

19. Прослідкуйте еволюцію патерну BCE у сучасні архітектурні шаблони: **MVC** та **Чисту архітектуру (Clean/Onion Architecture)**.

20. Охарактеризуйте поняття *Підсистеми (Subsystem)*. Які інженерні переваги дає дотримання принципів **High Cohesion** та **Low Coupling** під час декомпозиції системи?

21. Чому на етапі проєктування (Design) підсистеми мають взаємодіяти *виключно через інтерфейси*?

22. Наведіть приклади застосування патернів **GoF (Gang of Four)** у RUP: як патерни *Facade*, *Strategy* та *Adapter* розв'язують задачі структурування підсистем та класів BCE?

23. Які три ключові діаграми UML складають *Логічний вигляд (Logical View)* архітектури ПЗ на етапі проєктування та які задачі вони виконують?

24. Як концепція проєктування інтерфейсів RUP пов'язана із сучасним підходом **API-First Design**?

ТЕМА 4. ДОПОМІЖНІ ДИСЦИПЛІНИ, РОЛІ ТА АРТЕФАКТИ

В інженерії програмного забезпечення написати код — це лише частина справи. Якщо в команді працює 50 фахівців, хтось має гарантувати, що вони не перезапишуть код один одного, що проєкт не вийде за межі бюджету, і що кожен має налаштоване робоче середовище.

Саме для цього в RUP існують 3 допоміжні (підтримуючі) дисципліни. Вони не створюють безпосередньої бізнес-цінності для замовника (замовнику байдуже, яку систему контролю версій ви використовуєте), але без них великий проєкт гарантовано зазнає краху.

Лекція 17. Управління конфігурацією та змінами (Configuration & Change Management)

Мета лекції: Зрозуміти критичну важливість контролю над проєктними артефактами. Вивчити процеси версіонування, створення базових ліній (Baselines) та формального управління запитами на зміни (Change Requests).

Що таке конфігураційне управління?

Уявіть ситуацію: тестувальник знайшов критичну помилку у версії продукту 1.2, яка вже встановлена у клієнта. Розробник відкриває свій код, намагається відтворити помилку, але в нього "все працює", тому що його локальний код вже відповідає майбутній версії 2.0.

Без системи, яка дозволяє в будь-який момент часу точно відтворити стан проєкту на конкретну дату, розробка неможлива.

Елемент конфігурації (Configuration Item - CI) — це будь-який артефакт проєкту, який підлягає контролю версій. Це не лише вихідний код! До CI належать:

- Документ вимог (SRS) та Vision.
- UML-діаграми архітектури.
- Тестові скрипти.
- Сценарії розгортання та конфігурації баз даних.

Базові лінії (Baselines)

Коли команда успішно завершує певну ітерацію або проходить головну віху (наприклад, LCA), стан усіх елементів конфігурації "заморожується". Цей зріз проєкту називається **Базовою лінією (Baseline)**.

- Базова лінія є гарантованою, стабільною точкою відліку.
- Змінити артефакт, який увійшов до базової лінії, просто так неможливо — для цього потрібен формальний дозвіл.

Управління змінами (Change Management)

Зміни у програмному проєкті неминучі. Замовник може захотіти нову функцію, розробник може знайти краще архітектурне рішення, або тестувальник виявить баг. RUP стверджує: зміни — це нормально, але вони мають бути керованими.

Процес обробки Запиту на зміну (Change Request - CR):

1. **Ініціація:** Будь-хто (користувач, тестувальник, аналітик) створює CR. Це може бути повідомлення про дефект (Bug) або запит на нову фічу (Enhancement).
2. **Оцінка впливу (Impact Analysis):** Аналітик та Архітектор використовують *Матрицю трасування*, щоб визначити: якщо ми додамо цю функцію, скільки класів доведеться переписати і скільки це коштуватиме?
3. **Рішення CCB (Change Control Board):** Рада з управління змінами (куди входять менеджер проєкту, представник замовника та головний архітектор) розглядає оцінку. Вони можуть:
   - Схвалити зміну (і виділити на неї бюджет).
   - Відхилити її.
   - Відкласти до наступного релізу (перенести в беклог).
4. **Реалізація:** Тільки після схвалення розробник отримує право змінити код (checkout) і створити нову версію артефакту (check-in).

Еволюція дисципліни: Від ClearCase до Git та Pull Requests

В епоху створення RUP інструменти конфігураційного управління (наприклад, IBM Rational ClearCase) були централізованими, важкими і вимагали ексклюзивного блокування файлів (один розробник редагує — інші чекають).

Сьогодні ця дисципліна еволюціонувала:

- **Git та Розподілені системи:** Дозволяють розробникам працювати паралельно у власних гілках (branches), зливаючи результати лише тоді, коли вони готові.
- **Pull Requests (Merge Requests):** Це сучасний, полегшений аналог засідання CCB (Ради з управління змінами). Замість бюрократичних комітетів зміну коду

розглядають інші розробники (Code Review) та автоматичні системи тестування (CI pipelines) перед тим, як код потрапить до головної гілки.

Лекція 18. Управління проєктом (Project Management). Планування та контроль ризиків

Мета лекції: Розібрати, як ітеративний підхід RUP змінює класичне управління проєктами. Дослідити механізми дворівневого планування, управління ризиками та об'єктивного контролю прогресу. Поглибити розуміння дисципліни управління проєктами в RUP. Розглянути техніки об'єктивної оцінки трудовитрат (зокрема на основі прецедентів), структуру головного керівного документа — Плану розробки ПЗ (SDP), та деталізувати процес управління ризиками.

Дворівневе планування: Фази та Ітерації

Як ми вже зазначали, RUP відкидає ідею створення єдиного детального графіка на весь період проєкту. Планування відбувається на двох рівнях, що дозволяє зберігати гнучкість:

1. **План фаз (Phase Plan / Coarse-Grained Plan):** Це високорівневий розклад. Він визначає тривалість чотирьох фаз (Inception, Elaboration, Construction, Transition) і дати досягнення головних віх. Цей план використовується керівництвом та інвесторами для контролю фінансування. Змінюється рідко.
2. **План ітерації (Iteration Plan / Fine-Grained Plan):** Це детальний план дій команди на найближчі 2–6 тижнів. Він містить конкретні задачі: наприклад, «Написати специфікацію для прецеденту авторизації», «Спроєктувати таблицю користувачів», «Провести модульне тестування класу PaymentGateway». Цей план складається *лише* на одну поточну ітерацію.

Золоте правило RUP: Наприкінці кожної ітерації менеджер оцінює фактичну швидкість команди і на основі цих реальних даних будує план для наступної ітерації.

Оцінка трудовитрат (Effort Estimation) та Конус невизначеності

Одне з найважчих запитань від замовника: *"Скільки це буде коштувати і коли буде готово?"*.

В інженерії ПЗ діє правило **«Конуса невизначеності» (Cone of Uncertainty)**. На фазі *Inception*, коли вимоги ще розмиті, похибка оцінки може становити від -50% до +200%. Наприкінці фази *Elaboration*, коли архітектура протестована, похибка звужується до ±10%. RUP чесно визнає цей конус і не вимагає точних оцінок у перший день.

Як оцінюють трудовитрати в RUP? Оскільки процес керується прецедентами (Use Cases), найпопулярнішим методом оцінки є **Use Case Points (UCP - Бали прецедентів)**.

Кроки оцінки за методом UCP:

1. **Оцінка складності Акторів:** Проста зовнішня система (API) — це 1 бал. Людина, що працює через складний графічний інтерфейс — це 3 бали.
2. **Оцінка складності Прецедентів:** Прецедент з 1-3 транзакціями — це 5 балів. Прецедент з понад 7 транзакціями та складною логікою — 15 балів.
3. **Технічна складність (TCF):** Враховуються вимоги до продуктивності, безпеки, паралелізму обчислень.
4. **Кваліфікація команди (ECF):** Досвідчена команда виконає роботу швидше, новачки вимагатимуть більше часу на навчання.

Перемноживши ці фактори, менеджер отримує загальну кількість "Балів". Знаючи продуктивність команди (наприклад, 20 годин на реалізацію 1 бала), можна досить точно вирахувати необхідний бюджет і час.

Управління ризиками (Risk Management)

Ризики — це потенційні проблеми, які ще не сталися. Якщо ігнорувати ризики, вони перетворюються на реальні катастрофи (Issues). RUP є ризик-орієнтованою методологією: робота в ітерації планується так, щоб усунути найбільші ризики першими.

Реєстр ризиків (Risk List): Кожен виявлений ризик заноситься до реєстру та оцінюється за двома параметрами:

- **Ймовірність (Probability):** Від 0 до 100%.
- **Вплив (Impact):** Наприклад, у грошовому еквіваленті або днях затримки.

Рівень ризику = Ймовірність × Вплив. Ризики з найвищим показником обробляються негайно за однією зі стратегій: *Ухилення, Пом'якшення, Передача* або *Прийняття* (детально розглянуто в попередніх матеріалах).

План розробки ПЗ (Software Development Plan - SDP)

Усі результати планування компілюються у головний керівний артефакт дисципліни — **План розробки ПЗ (SDP)**.

Важливо розуміти, що в RUP SDP — це не статичний документ, написаний один раз і забутий. Це "живий" парасольковий артефакт (Umbrella Artifact), який агрегує інші важливі документи.

Типова структура SDP:

1. **Організація проєкту:** Структура команди, ролі, зони відповідальності.
2. **План фаз (Phase Plan):** Графік досягнення ключових віх (LCO, LCA, IOC, PR).
3. **Бюджет та ресурси:** Оцінка трудовитрат (наприклад, на основі Use Case Points) та план залучення персоналу.
4. **Посилання на інші плани:**
   - *Risk Management Plan* (План управління ризиками та поточний Risk List).
   - *Configuration Management Plan* (Правила роботи з репозиторіями та базовими лініями).
   - *Quality Assurance Plan* (Стратегії тестування та метрики якості).
5. **Плани ітерацій:** Додаються динамічно в міру проходження проєкту.

SDP дозволяє будь-якому новому учаснику або аудитору за 15 хвилин зрозуміти, в якому стані знаходиться проєкт, як він керується і куди рухається.

Трансформація у сучасні практики (Agile / Scrum)

Класичні інструменти планування RUP стали прямими попередниками сучасних гнучких методологій:

- **Use Case Points** трансформувалися у **Story Points**, де команди оцінюють складність задач (User Stories) не в годинах, а у відносних балах за допомогою техніки *Planning Poker*.
- **SDP та План фаз** еволюціонували у **Product Roadmap** та **Product Backlog** (керований Product Owner-ом).
- **План ітерації** перетворився на **Sprint Backlog**.
- Замість складних розрахунків у MS Project, сучасні команди використовують метрику **Velocity (Швидкість команди)** та діаграми згоряння задач **(Burn-down charts)** в Jira для прогнозування термінів.

Лекція 19. Дисципліна «Середовище» (Environment). Підготовка робочого простору. Вибір інструментарію розробки, налаштування процесів під потреби конкретної команди (Tailoring)

Мета лекції: Вивчити призначення допоміжної дисципліни «Середовище». Зрозуміти, чому методологія RUP не може застосовуватися «з коробки», освоїти принципи адаптації процесів (Tailoring) та розібрати підходи до стандартизації інструментарію розробки.

Сутність та завдання дисципліни «Середовище»

В інженерії програмного забезпечення неможливо вимагати від команди високої продуктивності, якщо програмісти витрачають дні на налаштування локальних серверів, аналітики малюють діаграми в несумісних програмах, а тестувальники не мають доступу до бази даних.

Дисципліна **«Середовище» (Environment)** забезпечує організацію програмного проєкту належним інструментарієм та адаптованими процесами. Це створення своєрідного «заводського цеху» до початку масового виробництва.

Основні цілі дисципліни:

1. Забезпечення команди необхідними програмними та апаратними засобами.
2. Адаптація (Tailoring) методології RUP під конкретні реалії проєкту.
3. Розробка внутрішніх стандартів та інструкцій для команди.
4. Технічна підтримка розробників протягом усього життєвого циклу проєкту.

Найвища активність цієї дисципліни спостерігається на фазі **Inception (Початковий етап)**, коли закладається фундамент проєкту. На наступних фазах зусилля зводяться до підтримки та оновлення інструментарію.

Адаптація процесу (Process Tailoring)

Найбільший міф про RUP полягає в тому, що це надзвичайно важка, бюрократизована методологія, яка вимагає створення сотень документів.

Насправді RUP — це **фреймворк процесів (Process Framework)**. Він містить повний набір ролей, артефактів та дисциплін для проєктів будь-якого розміру (навіть для розробки ПЗ для аерокосмічної галузі). Спроба застосувати *всі* елементи RUP для створення невеликого інтернет-магазину призведе до паралічу проєкту.

Процес налаштування RUP під конкретний проєкт називається **Адаптацією (Tailoring)**. За цей процес відповідає спеціальна роль — *Інженер процесу (Process Engineer)*.

Що саме адаптується?

- **Вибір артефактів:** Для невеликого проєкту документ «План розробки ПЗ» може складатися з 3 сторінок, а матриця трасування вестися у звичайній таблиці. Від багатьох додаткових документів (наприклад, плану внутрішнього аудиту) відмовляються взагалі.
- **Вибір ролей:** У команді з 5 осіб одна людина може суміщати ролі Системного аналітика та Тестувальника, а Розробник може бути одночасно Архітектором.
- **Рівень формальності:** Рішення про те, які артефакти потребують офіційного підпису замовника, а які створюються лише для внутрішнього використання.

Результат: Команда отримує власний, полегшений і чіткий регламент роботи — **Процес проєкту (Development Case)**.

Вибір та налаштування інструментарію розробки

Методологія RUP орієнтована на максимальну автоматизацію рутинної праці. За вибір, інсталяцію та підтримку інструментів відповідає *Спеціаліст з інструментарію (Tool Specialist)*.

Комплекс інструментів охоплює всі дисципліни:

1. **CASE-засоби (Computer-Aided Software Engineering):** Програми для бізнес-моделювання та проєктування архітектури через UML (раніше IBM Rational Rose, сьогодні — Enterprise Architect, Visual Paradigm, Draw.io).
2. **Інструменти управління вимогами:** Системи для створення матриць трасування та зберігання прецедентів (наприклад, Jira, Confluence, IBM DOORS).
3. **Середовища розробки (IDE):** Стандартизовані для всієї команди редактори коду з налаштованими плагінами та лінтерами (Visual Studio, IntelliJ IDEA, VS Code).
4. **Інструменти збірки та контролю версій:** Репозиторії (GitLab, GitHub) та системи автоматизації збірки (Maven, Gradle, Webpack).
5. **Інструменти тестування:** Фреймворки для Unit-тестів (JUnit, xUnit), інструменти для автоматизації UI-тестування (Selenium) та навантажувального тестування (JMeter).

Правило єдиного середовища: Кожен розробник у команді повинен мати ідентично налаштоване робоче середовище. Це запобігає виникненню класичної проблеми: *«На моєму комп'ютері код працює, а на сервері — ні»*.

Підготовка робочого простору та розробка інструкцій

Після вибору інструментів Дисципліна «Середовище» вимагає створення внутрішніх інструкцій (Guidelines). Якщо приходить новий розробник (онбординг), він не повинен витрачати тиждень на спроби підключитися до бази даних.

Документація середовища включає:

- **Стандарти кодування (Coding Guidelines):** Правила іменування змінних, форматування коду, глибини вкладеності функцій.
- **Інструкції з проєктування (Design Guidelines):** Правила побудови UML-діаграм, узгоджені архітектурні патерни.
- **Посібники з інструментів (Tool Guidelines):** Покрокові інструкції з інсталяції та налаштування локального середовища, доступи до тестових серверів.

Еволюція дисципліни: Від класичних інструментів до Platform Engineering

Як і інші складові RUP, дисципліна «Середовище» зазнала радикальних змін із розвитком хмарних технологій. Сьогодні задачі цієї дисципліни розв'язуються в межах напрямів **DevOps** та **Platform Engineering**.

- **Контейнеризація (Docker):** Проблема ідентичності середовищ вирішена кардинально. Замість написання довгих інструкцій з інсталяції баз даних та вебсерверів, створюється файл docker-compose.yml. Новий розробник просто запускає одну команду, і все необхідне середовище піднімається на його машині в ізольованих контейнерах за хвилину.
- **Інфраструктура як код (Infrastructure as Code - IaC):** Робочі та тестові сервери більше не налаштовуються вручну системними адміністраторами. Середовище описується у вигляді коду (Terraform, Ansible) і розгортається автоматично.
- **Хмарні середовища розробки (Cloud IDE):** Сучасні проєкти починають відмовлятися навіть від локальних IDE. Середовище розробки (разом із компіляторами та дебагерами) розгортається безпосередньо у веббраузері (наприклад, GitHub Codespaces або AWS Cloud9), що зводить час підготовки робочого простору до кількох секунд.

Розуміння принципів адаптації та стандартизації середовища, закладених у RUP, є критично важливим для створення сучасних, високопродуктивних інженерних команд.

Лекція 20. Рольова модель та ключові артефакти RUP. Детальний розбір зон відповідальності Аналітика, Архітектора, Розробника та Тестувальника

Мета лекції: Зрозуміти концепцію ролей у методології RUP та їхню відмінність від штатних посад. Детально розібрати обов'язки ключових учасників інженерного процесу, а також прослідкувати, як створюються, еволюціонують та затверджуються головні проєктні артефакти.

Концепція ролей у RUP: Роль — це не посада

Фундаментальне правило RUP щодо організації команди полягає в тому, що **Роль (Role)** та **Посада (Job Title)** — це різні поняття.

- **Посада** — це адміністративний статус людини в компанії (наприклад, "Senior Java Developer" або "QA Automation Engineer").
- **Роль** у RUP — це набір обов'язків, поведінки та компетенцій, необхідних для виконання певного завдання. Це "капелюх", який працівник одягає для виконання конкретної роботи.

Наслідки такого підходу:

- Одна людина може виконувати кілька ролей одночасно. Наприклад, досвідчений програміст (посада) на початку ітерації може одягнути капелюх *Аналітика* (щоб уточнити вимоги), потім стати *Архітектором* (щоб спроєктувати базу даних), а потім — *Розробником* (щоб написати код).
- Одну роль може виконувати ціла група людей (наприклад, роль *Тестувальника* розподілена між п'ятьма фахівцями).
- Ролі динамічно змінюються залежно від фази життєвого циклу проєкту.

Ключові інженерні ролі та зони їхньої відповідальності

RUP визначає понад 30 різних ролей, але серцевину інженерного процесу складають чотири головні фігури.

А) Системний аналітик (System Analyst) Це "перекладач" між замовником (який мислить бізнес-категоріями) та розробниками (які мислять алгоритмами).

- **Зона відповідальності:** Розуміння предметної області, збір, аналіз та управління вимогами. Саме аналітик визначає межі системи.

- **Ключові дії:** Проведення інтерв'ю зі стейкхолдерами, написання специфікацій прецедентів (Use Cases), структурування моделі вимог.
- **Відповідає за артефакти:** Документ "Бачення" (Vision), Специфікація вимог (SRS), Глосарій.

Б) Архітектор програмного забезпечення (Software Architect) Технічний лідер проєкту. Якщо аналітик відповідає на запитання *"Що робити?"*, то архітектор — *"Як це побудувати, щоб воно не розвалилося?"*.

- **Зона відповідальності:** Прийняття найважливіших технічних рішень. Вибір технологічного стека, розбиття монолітної системи на підсистеми з чіткими інтерфейсами, управління нефункціональними вимогами (продуктивність, безпека).
- **Ключові дії:** Створення моделі аналізу (Boundary-Control-Entity), проєктування виконуваної архітектури (на фазі Elaboration), пом'якшення ключових технічних ризиків.
- **Відповідає за артефакти:** Документ архітектури ПЗ (Software Architecture Document - SAD), Архітектурні моделі (UML-діаграми класів та компонентів).

В) Розробник / Реалізатор (Implementer) Будівельник системи, який матеріалізує архітектурні моделі у вигляді робочого коду.

- **Зона відповідальності:** Написання чистого, оптимізованого вихідного коду, який відповідає проєктній моделі та стандартам кодування.
- **Ключові дії:** Проєктування на мікрорівні (всередині конкретного класу), кодування, написання та запуск модульних тестів (Unit Tests), інтеграція свого коду з кодом колег.
- **Відповідає за артефакти:** Вихідний код, Скомпільовані компоненти, Скрипти модульного тестування.

Г) Тестувальник (Tester) Охоронець якості (Quality Gatekeeper). Його мета — не довести, що система працює, а знайти способи її "зламати" до того, як це зроблять користувачі.

- **Зона відповідальності:** Перевірка системи на відповідність вимогам (як функціональним, так і нефункціональним).
- **Ключові дії:** Написання тестових сценаріїв на основі Use Cases, виконання інтеграційного та системного тестування, документування знайдених дефектів.
- **Відповідає за артефакти:** Тест-план, Тестові сценарії (Test Cases), Звіти про дефекти (Bug Reports).
-

Життєвий цикл ключових артефактів RUP

Артефакт — це будь-який відчутний результат роботи в межах процесу (текстовий документ, UML-модель, фрагмент коду, база даних). Артефакти не створюються за один день; вони мають власний життєвий цикл, який жорстко контролюється дисципліною *Управління конфігураціями*.

Типовий життєвий цикл артефакту:

1. **Створення (Draft):** Документ створюється в чорновому варіанті (наприклад, перші начерки вимог на фазі Inception).
2. **Рецензування (Review):** Артефакт перевіряється іншими членами команди (наприклад, Code Review для коду або вичитка SRS замовником).
3. **Базова лінія (Baseline):** Після схвалення артефакт отримує офіційний статус версії (наприклад, v.1.0). З цього моменту він "заморожується".
4. **Зміна через CR (Change Request):** Якщо потрібно внести зміни в заморожений артефакт, створюється офіційний Запит на зміну. Після його схвалення генерується нова версія документа (v.1.1).

Динаміка розвитку ключових документів за фазами:

- **Документ Vision та Специфікація вимог (SRS):**
  - *Inception:* Створюються чернетки, фіксуються основні межі системи (близько 20% вимог).
  - *Elaboration:* Документи активно доповнюються і досягають 80% повноти.
  - *Construction:* Вимоги заморожуються (Feature Freeze). Вносяться лише дрібні корективи через Запити на зміну (CR).
- **Документ архітектури програмного забезпечення (SAD):**
  - *Inception:* Містить лише загальні ідеї та вибір технологій.
  - *Elaboration:* Досягає свого фінального вигляду. Створюється виконувана архітектура, UML-моделі стабілізуються.
  - *Construction:* Майже не змінюється. Розробники просто дотримуються встановлених правил.
- **Вихідний код:**
  - *Elaboration:* Створюється лише "скелет" (Executable Architecture) для перевірки ризиків.
  - *Construction:* Експоненційне зростання обсягу коду.
  - *Transition:* Написання нового коду заборонено. Допускається лише виправлення критичних багів (Bug Fixing).

Трансформація ролей та артефактів у сучасних Agile-командах

Класична рольова модель RUP була досить формалізованою, що стало об'єктом критики. Сьогодні, у гнучких (Agile) методологіях, ці концепції еволюціонували:

- **Крос-функціональність:** Замість жорсткого поділу на "програмістів" та "тестувальників", Scrum вимагає створення крос-функціональних команд розробки (Development Team), де кожен член команди може підхопити задачу іншого (наприклад, програміст пише автотести, якщо QA не встигає).
- **Злиття ролей:** Роль Аналітика часто бере на себе **Product Owner** (Власник продукту), який керує беклогом (сучасним аналогом Use Case моделі). Окремої посади "Архітектор" у багатьох Agile-командах немає — архітектурні рішення приймаються найдосвідченішими розробниками колективно.
- **Полегшення артефактів:** Важкі текстові документи на кшталт SAD сьогодні замінюються легкими статтями у внутрішніх Wiki-системах (Confluence), інструментами автоматичної генерації документації з коду (Swagger/OpenAPI), а також архітектурними рішеннями, які фіксуються у форматі ADR (Architecture Decision Records).

Загальні висновки до теми 4:

Тема 4 завершує комплексний розгляд методології Rational Unified Process (RUP), зміщуючи фокус із суто інженерних задач (проєктування, кодування, тестування) на управлінський та інфраструктурний каркас проєкту. Допоміжні дисципліни, рольова модель та правила життєвого циклу артефактів дають відповідь на критичне для великих команд запитання: **«Як організувати хаос, забезпечити контроль над змінами та налаштувати робоче середовище під специфіку конкретного продукту?»**.

Опрацювання матеріалів цього розділу дозволяє зробити кілька фундаментальних висновків:

- **Стабілізація проєкту через конфігураційний контроль:** Програмний проєкт — це живий організм, де вимоги, код та архітектурні моделі змінюються щодня. Концепція *Елементів конфігурації* та *Базових ліній (Baselines)* захищає команду від десинхронізації та «інтеграційного пекла». Управління змінами через формальні запити (CR) та раду CCB гарантує, що жодне рішення не приймається наосліп, а кожна зміна проходить ретельний аналіз впливу на бюджет і архітектуру (*Impact Analysis*).
- **Атака на ризики як альтернатива мікроменеджменту:** RUP перевертає класичне уявлення про управління проєктами. Замість спроб сліпо виконувати

жорсткий дворічний план, менеджмент використовує дворівневе планування (стратегічний *План фаз* та тактичний *План ітерації*). Рушійною силою проєкту стає *Список ризиків*. Команда свідомо інвестує ресурси в усунення найбільших технічних та бізнес-загроз на ранніх стадіях, що дозволяє звузити *Конус невизначеності*.

- **Методологія як конструктор (Tailoring):** Тема 4 повністю руйнує міф про RUP як про неповоротку та бюрократизовану систему. Дисципліна «Середовище» чітко вказує: RUP не можна використовувати «з коробки». Процес адаптації (*Tailoring*) та створення *Процесу проєкту (Development Case)* є обов'язковими. Проєкт має бути настільки легким, наскільки це можливо, і настільки формалізованим, наскільки це необхідно для забезпечення якості.
- **Динамічність рольової моделі:** Поділ понять «Посада» та «Роль» забезпечує максимальну гнучкість команди. Роль — це «капелюх» з певними зонами відповідальності. Це дозволяє невеликим командам суміщати ролі (наприклад, Аналітик + Тестувальник), а великим — чітко масштабувати обов'язки без адміністративного перевантаження.

Матриця допоміжних дисциплін: Від класичного RUP до сучасних практик.Для швидкого охоплення матеріалу та розуміння його актуальності, взаємозв'язок між підходами RUP та сучасним станом IT-індустрії узагальнено у таблиці:

Допоміжна дисципліна RUP	Головне завдання (Ціль)	Ключовий артефакт	Сучасна трансформація в IT-індустрії
1. Управління конфігурацією та змінами	Забезпечити збереження та версіонування всіх артефактів, контролювати внесення змін.	Базова лінія (Baseline), Запит на зміну (Change Request)	Розподілені системи контролю версій (Git), процеси Pull / Merge Requests як полегшений аналог рішень ССВ.
2. Управління проєктом	Спланувати ітерації, розрахувати трудовитрати, нівелювати ризики та моніторити метрики прогресу.	План розробки ПЗ (SDP), Список ризиків, План ітерації	Agile / Scrum / Kanban. План ітерації став Sprint Backlog, оцінка UCP перетворилася на Story Points, а SDP замінено на Product Roadmap.
3. Середовище	Підготувати інструменти для команди, адаптувати процеси під проєкт, розробити Guidelines.	Процес проєкту (Development Case), Coding & Design Guidelines	DevOps, Platform Engineering, контейнеризація (Docker), концепції Інфраструктури як коду (IaC) та Cloud IDE.

Контрольні запитання до Теми 4

Дисципліна «Реалізація» (Implementation)

1. У чому полягає суть дисципліни реалізації в RUP? На яких фазах життєвого циклу проєкту її інтенсивність є найвищою?
2. Опишіть правила відображення (*Mapping*) елементів проєктування на фізичний код (підсистеми, класи, інтерфейси).
3. За якими принципами організовується сучасна кодова база (*Codebase*) великого проєкту? Що таке принцип ізоляції робочих просторів?
4. Яку роль відіграють *Стандарти кодування (Coding Guidelines)*, *Code Review* та *Рефакторинг* у мінімізації технічного боргу (*Technical Debt*)?
5. Поясніть різницю між *Поступовою інтеграцією коду* та концепцією **"Integration Hell"** (Інтеграційне пекло).
6. Що таке автоматизована збірка системи (**Build**) та як практика *Daily Builds* трансформувалася у сучасні пайплайни **CI/CD**?

Дисципліна «Тестування» (Testing)

7. Чому RUP розглядає тестування як безперервну дисципліну, а не як фінальну фазу проєкту?
8. Розкрийте ієрархію рівнів тестування в RUP: **Unit, Integration, System**. Які з них належать до тестування "білого", а які — "чорного ящика"?
9. Яким чином здійснюється тестування на основі прецедентів (*Use-Case Based Testing*)?
10. Опишіть класичний життєвий цикл дефекту (**Bug Lifecycle**). У яких випадках баг отримує статус *Reopened*?
11. Що таке **Тріаж дефектів (Defect Triage)**? У чому полягає різниця між параметрами **Severity** (Серйозність) та **Priority** (Пріоритет)? Наведіть приклад бага з низьким Severity, але найвищим Priority.
12. Як дисципліна тестування RUP співвідноситься із сучасними концепціями *QA Automation, TDD (Test-Driven Development)* та *Shift-Left Testing*?

Дисципліна «Розгортання» (Deployment)

13. Які інженерні та логістичні завдання вирішує дисципліна розгортання? На якій фазі RUP вона досягає свого піку?
14. Дайте характеристику трьом основним типам супровідної документації: *Release Notes, Посібник користувача, Посібник адміністратора*.
15. Поясніть, чому класичні методи розгортання (інсталятори та ручне налаштування серверів) еволюціонували у сучасні практики **DevOps**, **Контейнеризацію (Docker/Kubernetes)** та **Infrastructure as Code (IaC)**.

Дисципліна «Управління конфігурацією та змінами» (Configuration & Change Management)

16. Дайте визначення **Елементу конфігурації (Configuration Item - CI)**. Чому конфігураційне управління в RUP поширюється не лише на вихідний код, а й на документацію та моделі?
17. Що таке **Базова лінія (Baseline)** і в які моменти життєвого циклу проєкту вона створюється? Яка її головна інженерна функція?
18. Опишіть покроковий життєвий цикл **Запиту на зміну (Change Request - CR)** від моменту його ініціації до реалізації в коді.
19. Які функції виконує Рада з управління змінами (**Change Control Board - CCB**)? Хто зазвичай входить до її складу?
20. Як аналітики та архітектори використовують матрицю трасування для аналізу впливу змін (*Impact Analysis*) при отриманні нового CR?
21. Порівняйте класичні важкі інструменти конфігураційного управління (на кшталт *ClearCase*) із сучасними розподіленими системами (**Git**). Як концепція *Pull / Merge Requests* замінила традиційні засідання CCB?

Дисципліна «Управління проєктом» (Project Management)

22. У чому полягає суть дворівневого планування в RUP? Чим стратегічний **План фаз (Phase Plan)** відрізняється від тактичного **Плану ітерації (Iteration Plan)**?
23. Розкрийте концепцію **«Конуса невизначеності» (Cone of Uncertainty)**. Як змінюється похибка оцінки трудовитрат від фази *Inception* до фази *Construction*?
24. Опишіть алгоритм оцінки трудовитрат за методом **Use Case Points (UCP)**. Які чинники (актори, транзакції, технічна складність, кваліфікація команди) впливають на кінцевий результат?
25. Як у RUP розраховується показник **Очікуваної величини ризику (Risk Exposure)**? Сформулюйте математичну залежність.
26. Детально опишіть чотири базові стратегії роботи з ризиками: *Ухилення (Avoidance)*, *Пом'якшення (Mitigation)*, *Передача (Transfer)* та *Прийняття (Acceptance)*. Наведіть приклади для кожної.
27. Яка структура та інженерне призначення **Плану розробки ПЗ (Software Development Plan - SDP)**? Чому його називають «парасольковим» артефактом?
28. Чому в RUP категорично заборонено оцінювати прогрес проєкту на основі суб'єктивних звітів типу "код готовий на 90%"? Які об'єктивні метрики використовуються натомість?

29. Прослідкуйте еволюцію ролі *Project Manager* з RUP у гнучкі методології (**Scrum**). Як його обов'язки розділилися між *Product Owner* та *Scrum Master*?

Дисципліна «Середовище» (Environment)

30. Чому методологію RUP неможливо і небезпечно застосовувати на реальних проєктах «з коробки» без попередніх змін?
31. Що таке **Адаптація процесу (Process Tailoring)**? Які критерії (розмір команди, критичність системи, технології) впливають на вибір артефактів та ролей?
32. Яке призначення артефакту **Процес проєкту (Development Case)**? Чим він відрізняється від загальної методології RUP?
33. Які типи внутрішніх інструкцій (**Guidelines**) розробляються в межах цієї дисципліни для забезпечення однаковості роботи команди?
34. Які інструменти автоматизації (CASE-засоби, IDE, системи збірки, трекери) мають бути стандартизовані для команди до початку активної фази розробки?
35. Як сучасні технології контейнеризації (**Docker**) та концепції Інфраструктури як коду (**IaC**) вирішили класичні завдання дисципліни «Середовище» щодо синхронізації робочих просторів?

Рольова модель та життєвий цикл артефактів

36. Поясніть фундаментальне правило RUP: *«Роль — це не посада в штатному розкладі»*. Наведіть приклади суміщення ролей для невеликих інженерних команд. Окресліть зони відповідальності та ключові артефакти для чотирьох базових ролей: **Системний аналітик** (*System Analyst*) **Архітектор програмного забезпечення** (*Software Architect*) **Розробник / Реалізатор** (*Implementer*) **Тестувальник** (*Tester*)
37. Опишіть життєвий цикл класичного документа в RUP за статусами: *Draft (Чернетка) → Review (Рецензування) → Approved/Baselined (Затверджено) → Зміна через CR*.
38. Як змінюється динаміка розвитку та рівень деталізації документів **SRS** (Специфікація вимог) та **SAD** (Документ архітектури) під час переходу проєкту від фази *Inception* до фази *Transition*?
39. Як еволюціонувала рольова модель RUP у сучасних **крос-функціональних** Agile-командах? Як сьогодні документуються архітектурні рішення (концепція *ADR - Architecture Decision Records*)?

Тема 5. Трансформація класичних процесів у сучасні інженерні практики

Сучасна інженерія програмного забезпечення пройшла довгий шлях еволюції від суворих, жорстко регламентованих (предиктивних) процесів до гнучких (адаптивних) та автоматизованих практик. Ця тема розкриває, як фундаментальні принципи класичних процесів, зокрема RUP (Rational Unified Process), були трансформовані, спрощені та інтегровані в сучасні методології (Agile, DevOps).

Обмеження класичних процесів розробки

Класичні методології (наприклад, Каскадна модель або «важкі» версії RUP з надмірною кількістю артефактів) були орієнтовані на детальне планування перед початком розробки.

Основні проблеми класичних підходів у сучасних реаліях:

- **BDUF (Big Design Up Front):** Спроба спроєктувати всю архітектуру та вимоги до написання коду. Це призводило до створення неактуальної документації, оскільки вимоги ринку змінювалися швидше, ніж завершувався етап проєктування.
- **Довгий Time-to-Market:** Від моменту збору вимог до першого релізу могли проходити місяці або роки.
- **Бункерність (Silos):** Розробники, тестувальники та системні адміністратори працювали ізольовано, що ускладнювало комунікацію та інтеграцію продукту.
- **Важковаговість процесів:** Величезна кількість обов'язкових артефактів (документів, звітів, діаграм), які не приносили прямої цінності кінцевому користувачеві.

Вплив Agile-маніфесту на трансформацію процесів

Переломним моментом став 2001 рік (Agile Manifesto), який змістив фокус із процесів та інструментів на людей та працюючий продукт.

Як класичні принципи трансформувалися в Agile:

- *Замість вичерпної документації* $\rightarrow$ **Працююче програмне забезпечення**. Документація залишається (наприклад, архітектурні рішення фіксуються через ADR - Architecture Decision Records), але вона стала мінімалістичною.

• *Замість жорсткого слідування плану* $\rightarrow$ **Готовність до змін**. Ітерації RUP (які могли тривати місяцями) перетворилися на короткі спринти Scrum (1-4 тижні).

Еволюція RUP: від "важкого" фреймворку до Agile Unified Process

RUP від самого початку був ітеративним та орієнтованим на ризики процесом, що робило його концептуально ближчим до сучасних практик, ніж Waterfall. Проте його складність вимагала адаптації.

Похідні "легкі" процеси на базі RUP:

• **AUP (Agile Unified Process):** Спрощена версія RUP, запропонована Скоттом Емблером. Вона об'єднала ітеративний підхід RUP з технічними практиками Agile (наприклад, Test-Driven Development, Refactoring).
• **OpenUP (Open Unified Process):** Мінімалістичний фреймворк, який зберіг основні характеристики RUP (керованість прецедентами, архітектурна орієнтованість, ітеративність), але відкинув зайву бюрократію. Фази залишилися тими ж (Inception, Elaboration, Construction, Transition), але виконуються набагато швидше і з фокусом на мікро-ітерації.

Ключові сучасні інженерні практики (The Modern Engineering Practices)

Трансформація процесів призвела до появи технічних практик, які дозволяють реалізовувати гнучкість на рівні коду та інфраструктури.

CI/CD (Continuous Integration / Continuous Delivery)

Якщо в класичному RUP етап інтеграції (Integration) часто був окремою болючою фазою наприкінці розробки, то зараз це безперервний процес.

• **Continuous Integration (CI):** Код інтегрується в спільний репозиторій кілька разів на день. Кожен коміт супроводжується автоматичною збіркою та тестуванням.
• **Continuous Delivery/Deployment (CD):** Автоматизоване розгортання перевіреного коду на тестових (Delivery) або відразу на продуктових (Deployment) серверах.

DevOps та DevSecOps

Трансформація організаційної культури, що руйнує бар'єри між розробкою (Dev), експлуатацією (Ops) та безпекою (Sec).

- **IaC (Infrastructure as Code):** Інфраструктура описується кодом (Terraform, Ansible), що дозволяє розгортати середовища миттєво, усуваючи необхідність ручного налаштування серверів.

Автоматизація тестування та Shift-Left підхід

У класичних процесах тестування відбувалося наприкінці. Сучасні практики зміщують тестування "вліво" (ближче до початку розробки):

- **TDD (Test-Driven Development):** Написання тестів *до* написання самого коду.
- **BDD (Behavior-Driven Development):** Опис поведінки системи зрозумілою для бізнесу мовою (наприклад, Gherkin), що автоматично трансформується в тести. Це сучасна реалізація принципу RUP "Use-case driven" (керованість прецедентами).

Мікросервісна та Cloud-Native архітектура

Лекція 21. Від важковагових процесів до Agile. Обмеження класичного RUP. Огляд Agile Unified Process (AUP) та OpenUP

На початку 2000-х років індустрія розробки програмного забезпечення зіткнулася з кризою неефективності так званих «важковагових» (heavyweight) методологій. Хоча RUP замислювався як гнучкий та ітеративний фреймворк, на практиці він часто перетворювався на бюрократичний процес. Ця лекція розглядає причини кризи класичного RUP та те, як спрощені процеси (AUP, OpenUP) стали містком до сучасних Agile-фреймворків.

Обмеження класичного RUP («Важковаговість» процесу)

Класичний Rational Unified Process включає 4 фази, 9 дисциплін та понад 100 можливих артефактів (документів, моделей, звітів). Хоча творці RUP наголошували на необхідності його адаптації (tailoring) під потреби конкретного проєкту, більшість організацій намагалися впровадити його «як є».

Основні проблеми та обмеження:

- **Надмірна бюрократія:** Створення великої кількості документації, яка не приносила прямої цінності замовнику, але вимагала багато часу на підтримку її в актуальному стані.
- **Аналітичний параліч (Analysis Paralysis):** Команди витрачали місяці на фазах Початку (Inception) та Проєктування (Elaboration), намагаючись передбачити всі ризики та вимоги до написання коду.

- **Високий поріг входження:** Складність фреймворку вимагала тривалого навчання персоналу та впровадження дорогих CASE-засобів (наприклад, IBM Rational Rose).
- **Негнучкість до постійних змін:** Незважаючи на ітеративність, внесення змін у вимоги на пізніх етапах вимагало оновлення величезного масиву пов'язаних артефактів.

Agile Unified Process (AUP): Спрощення заради швидкості

Розуміючи недоліки класичного підходу, Скотт Емблер (Scott Ambler) розробив **Agile Unified Process (AUP)** у 2005 році. Це була спрощена версія RUP, яка об'єднала його фазову структуру з технічними практиками Agile.

Ключові відмінності AUP від класичного RUP:

1. **Зменшення кількості дисциплін:** Замість 9 дисциплін залишилося 7 (Моделювання, Реалізація, Тестування, Розгортання, Управління конфігураціями, Управління проєктом, Середовище). Бізнес-моделювання та аналіз об'єднали в одну дисципліну.
2. **Інтеграція Agile-практик:** AUP явно підтримує розробку через тестування (TDD), гнучке моделювання (Agile Modeling) та рефакторинг баз даних.
3. **«Серійний у великому, ітеративний у малому»:** Проєкт все ще має 4 класичні фази RUP (Inception, Elaboration, Construction, Transition) для управління життєвим циклом, але всередині фаз робота ведеться короткими ітераціями, що нагадують спринти.

OpenUP: Відкритий міст до сучасних методологій

OpenUP (Open Unified Process) — це ще більш мінімалістичний і відкритий фреймворк, розроблений Eclipse Foundation (як частина Eclipse Process Framework). Він зберіг фундаментальні характеристики RUP, але відкинув усе зайве, ставши ідеальним перехідним етапом для команд, які рухалися в бік Scrum.

Чотири ключові принципи OpenUP:

1. **Співпраця (Collaborate):** Об'єднання інтересів замовників і розробників (що перегукується з цінностями Agile-маніфесту).
2. **Баланс пріоритетів (Balance):** Розуміння того, що архітектура, вимоги та обмеження проєкту повинні балансувати для створення максимальної цінності.

3. **Фокус на архітектурі (Focus on architecture):** Раннє зниження технічних ризиків шляхом створення стабільної архітектури на етапі Elaboration (спадщина RUP).

4. **Еволюційний розвиток (Evolve):** Постійне отримання зворотного зв'язку через часті релізи робочого ПЗ.

Мікро-ітерації: В OpenUP з'явилося поняття мікро-ітерацій (від кількох днів до тижня), що зробило його дуже схожим на сучасний Scrum.

AUP та OpenUP як міст до Scrum та Kanban

Спрощені версії RUP стали необхідним еволюційним кроком для індустрії. Вони дозволили великим корпораціям психологічно та технічно перейти від Waterfall-подібного мислення до Agile.

Трансформація ключових елементів:

- **Від Артефактів вимог до Backlog'у:** В OpenUP з'явився *Work Items List* (Список робочих елементів), який концептуально є тим самим, що й *Product Backlog* у Scrum.
- **Від Ітерацій до Спринтів:** Ітерації тривалістю 3-6 місяців у класичному RUP стиснулися до 2-4 тижнів у AUP/OpenUP, що стало стандартом для Scrum.
- **Мікропланування:** OpenUP запровадив концепцію щоденного планування (Daily meetings), що є прямим аналогом Daily Scrum.
- **Рух до Kanban:** Фокус OpenUP на мінімізації незавершеної роботи (WIP - Work in Progress) та фокусування на архітектурно значущих прецедентах підготували команди до розуміння принципів потоку (Flow), які лежать в основі Kanban-методу.

Класичний RUP заклав міцний фундамент інженерних практик (ітеративність, орієнтація на архітектуру та прецеденти), але його «важковаговість» заважала швидкості. AUP та OpenUP виступили каталізаторами змін: вони довели, що строгу інженерію RUP можна об'єднати з легкістю та адаптивністю Agile, проклавши шлях до домінування таких методологій, як Scrum та Kanban у сучасному IT.

Лекція 22. Еволюція дисципліни розгортання та управління конфігураціями: DevOps. Трансформація принципів RUP у CI/CD та роль контейнеризації

Класичний RUP містив дві окремі, але тісно пов'язані дисципліни: **Управління конфігурацією та змінами** (Configuration & Change Management) та **Розгортання** (Deployment). У традиційному виконанні ці етапи часто супроводжувалися великим обсягом ручної роботи, розбіжностями між середовищами та ізольованістю команд розробки й експлуатації. Ця лекція розглядає еволюцію цих дисциплін у сучасну культуру DevOps, конвеєри CI/CD та технології контейнеризації.

RUP створювався в епоху монолітних застосунків. Сьогодні архітектура трансформувалася:

- Системи розбиваються на незалежні мікросервіси, кожен з яких може розроблятися окремою невеликою командою (відповідно до закону Конвея та принципу "Two-pizza team").
- Використання контейнеризації (Docker) та оркестрації (Kubernetes).

Дисципліни RUP: від ручного управління до автоматизації

У класичному RUP дисципліна управління конфігураціями відповідала за збереження цілісності артефактів проєкту (коду, моделей, документів) і відстеження версій. Дисципліна розгортання фокусувалася на доставці готового ПЗ кінцевому користувачеві (написання інструкцій, пакування, інсталяція).

Проблеми класичного підходу:

- **Синдром «Працює на моєму комп'ютері»:** Програмне забезпечення, яке успішно працювало на комп'ютері розробника, часто ламалося на тестових або продуктових серверах через різницю у версіях бібліотек чи налаштуваннях ОС.
- **Бункерність (Silos):** Розробники (Dev) створювали код і «перекидали його через стіну» системним адміністраторам (Ops), які мали його розгорнути. Це породжувало конфлікти: розробники хотіли швидких змін, а адміністратори — стабільності системи.
- **Ручне розгортання:** Процес перенесення коду на сервери вимагав виконання довгих чеклістів вручну, що підвищувало ризик людської помилки (human error).

DevOps: Культурна та технічна еволюція

DevOps (Development + Operations) — це методологія та корпоративна культура, яка об'єднує розробку програмного забезпечення (Dev) та його експлуатацію (Ops). Це пряма відповідь на проблеми «бункерності», з якими стикалися команди, що використовували важкі процеси.

Як DevOps реалізує цілі RUP:

- RUP завжди пропагував ітеративність і зниження ризиків. DevOps доводить це до максимуму, дозволяючи випускати дрібні зміни по кілька разів на день. Чим менша зміна, тим нижчий ризик того, що вона зламає систему.
- Інфраструктура починає розглядатися як код (**Infrastructure as Code, IaC**). Замість ручного налаштування серверів, конфігурація описується у файлах (дотримуючись ідеалів RUP щодо строгого управління конфігураціями), які версіонуються разом із програмним кодом.

Трансформація принципів RUP у сучасні пайплайни CI/CD

Сучасним втіленням дисциплін інтеграції та розгортання RUP є конвеєри (пайплайни) CI/CD.

Continuous Integration (Безперервна інтеграція)

Автоматизований процес, при якому розробники часто (бажано кілька разів на день) зливають свої зміни до спільної гілки репозиторію.

- **Автоматизація збірки:** Як тільки код потрапляє в репозиторій, автоматично запускаються інструменти (наприклад, Maven, Gradle, Webpack), які компілюють код і збирають його у виконуваний файл або пакет.
- **Автоматизоване тестування:** Одразу після збірки запускаються модульні та інтеграційні тести. Якщо хоча б один тест падає — збірка вважається невдалою (broken build), і команда має негайно це виправити.

Continuous Delivery / Continuous Deployment (Безперервна доставка / розгортання)

- **Delivery (Доставка):** Кожна успішна збірка автоматично розгортається на тестовому середовищі (Staging). Реліз у Production відбувається натисканням однієї кнопки.
- **Deployment (Розгортання):** Повністю автоматизований процес, де кожна зміна, що пройшла всі тести, автоматично та без втручання людини розгортається безпосередньо у Production (продуктове середовище кінцевого користувача).

Роль контейнеризації (Docker) в інженерії ПЗ

Щоб конвеєри CI/CD працювали ідеально, необхідно було вирішити проблему розбіжностей між середовищами розробки, тестування та експлуатації. Рішенням стала контейнеризація, стандартом де-факто для якої є **Docker.**

Що таке контейнер? Це стандартизована одиниця програмного забезпечення, яка містить у собі сам код, середовище виконання (runtime), системні інструменти, системні бібліотеки та налаштування. Контейнер ізолює програму від операційної системи хоста.

Переваги Docker у контексті сучасного процесу розробки:

1. **Гарантована ідентичність середовищ:** Якщо контейнер працює на ноутбуці розробника, він зі 100% гарантією працюватиме точно так само на серверах AWS або Azure. Це остаточно знищує синдром «Працює на моєму комп'ютері».
2. **Легковаговість:** На відміну від традиційних віртуальних машин (VM), які містять повноцінну гостьову ОС, контейнери використовують ядро хостової ОС, що робить їх запуск миттєвим, а споживання ресурсів — мінімальним.
3. **Ідеальна сумісність з мікросервісами:** Кожен мікросервіс пакується в окремий контейнер, що дозволяє незалежно оновлювати, тестувати та масштабувати різні частини великої системи (що є сучасною реалізацією архітектурної орієнтованості RUP).

Дисципліни управління конфігураціями та розгортання, які в класичному RUP були обтяжені рутинними завданнями та бюрократією, сьогодні повністю автоматизовані. Культура DevOps, конвеєри CI/CD та контейнеризація Docker дозволили реалізувати фундаментальні принципи інженерії ПЗ — стабільність, повторюваність, швидкий зворотний зв'язок та мінімізацію ризиків — на технологічному рівні, недоступному в епоху створення RUP.

Лекція 23. Еволюція архітектурного підходу (Architecture-centric). Перехід від RUP-монолітів до Service-Oriented Architecture (SOA) та сучасних мікросервісних ландшафтів

Один із трьох фундаментальних принципів RUP звучить як **«Архітектурна орієнтованість» (Architecture-centric)**. Згідно з ним, стабільна архітектура має бути спроєктована та протестована на ранніх етапах (у фазі Elaboration) для мінімізації технічних ризиків. Однак те, *якою* є ця архітектура, зазнало кардинальних змін за останні десятиліття. Ця лекція розглядає еволюцію архітектурних стилів: від монолітних систем епохи раннього RUP до SOA та сучасних мікросервісів.

Архітектурна орієнтованість у класичному RUP: Епоха монолітів

В епоху створення RUP (кінець 90-х – початок 2000-х років) архітектура зазвичай описувалася за допомогою моделі «4+1 View Model» Філіпа Кручтена і базувалася на компонентному підході (Component-based architecture). Фізично ж ці системи найчастіше реалізовувалися як **моноліти** (або багаторівневі моноліти — n-tier).

Характеристики монолітної архітектури:

- Вся бізнес-логіка, інтерфейс користувача та доступ до даних об'єднані в єдину кодову базу (single codebase).
- Система збирається та розгортається як єдиний виконуваний файл або архів (наприклад, .war або .ear у Java).
- Використовується єдина, централізована реляційна база даних.

Обмеження монолітів, що призвели до еволюції:

- **Складність масштабування:** Якщо один компонент (наприклад, генерація звітів) вимагав більше ресурсів, доводилося масштабувати весь моноліт цілком.
- **Тісна зв'язність (Tight Coupling):** Зміна в одному модулі могла непередбачувано зламати інший, що вимагало тривалих регресійних тестувань (що затягувало фазу тестування в RUP).
- **Технологічне блокування:** Весь застосунок мав бути написаний однією мовою програмування з використанням одного стеку технологій.

Перехідний етап: Service-Oriented Architecture (SOA)

Коли корпоративні системи стали занадто великими, індустрія перейшла до **SOA (Сервіс-орієнтованої архітектури)**. Це була спроба декомпозувати великі

системи на набір незалежних сервісів рівня підприємства, які могли б перевикористовуватися.

Ключові особливості SOA:

- **Enterprise Service Bus (ESB):** Центральна інтеграційна шина. Сервіси не спілкувалися безпосередньо, а відправляли повідомлення в ESB, яка брала на себе маршрутизацію, трансформацію даних та безпеку.
- **Спілкування через важкі протоколи:** Використовувалися технології на кшталт SOAP та XML.
- **Спільне сховище даних:** Незважаючи на розділення логіки, сервіси в SOA часто продовжували працювати зі спільною базою даних.

Чому SOA не стала ідеалом? Центральна шина (ESB) швидко перетворилася на «вузьке місце» та новий моноліт. Логіка інтеграції стала настільки складною, що для внесення змін доводилося залучати окремі команди адміністраторів ESB, що руйнувало гнучкість розробки.

Сучасний етап: Мікросервісна архітектура (Microservices)

Мікросервіси є логічним продовженням SOA, але з виправленням її головних недоліків. Якщо SOA фокусувалася на перевикористанні сервісів на рівні всієї компанії, мікросервіси фокусуються на **незалежному розгортанні та автономності**.

Основні принципи мікросервісного ландшафту:

1. **Децентралізація (Decentralized Governance):** Відмова від центральної шини ESB на користь легковагих протоколів спілкування (REST API через HTTP/HTTPS, gRPC, або асинхронні брокери повідомлень на кшталт Apache Kafka чи RabbitMQ). «Розумні кінцеві точки та дурні канали зв'язку».
2. **База даних для кожного сервісу (Database-per-service):** Кожен мікросервіс має власну БД і повністю контролює свої дані. Це унеможливлює пряме втручання одного сервісу в дані іншого.
3. **Поліглотне програмування (Polyglot Persistence & Programming):** Команда може обрати найкращу мову та БД для конкретної задачі. Наприклад, сервіс аналітики може бути написаний на Python (з NoSQL БД), а ядро транзакцій — на Java (з PostgreSQL).
4. **Стійкість до збоїв (Resilience):** Падіння одного мікросервісу (наприклад, сервісу рекомендацій) не призводить до зупинки всієї системи (користувач все ще може робити покупки).

Як мікросервіси трансформують принципи RUP

Перехід до мікросервісів не скасовує принцип "Architecture-centric", але кардинально змінює підхід до проєктування:

- **Від компонентів до сервісів:** Те, що в RUP називалося «компонентом», тепер стало повністю автономним сервісом.
- **Закон Конвея:** Мікросервіси ідеально лягають на сучасну структуру команд. Згідно з законом Конвея, архітектура системи копіює структуру комунікацій в організації. Мікросервіс створюється невеликою крос-функціональною командою («Two-pizza team»), яка повністю володіє ним від проєктування до розгортання в Production.
- **Еволюційна архітектура замість BDUF:** Сучасні архітектори розуміють, що спроєктувати ідеальну систему на етапі Elaboration неможливо. Тому створюється базова архітектура, яка *готова до змін*. Зв'язки між сервісами проєктуються так, щоб будь-який сервіс можна було переписати з нуля або замінити, не впливаючи на решту системи.

Висновки

Фокус на архітектурі залишається критично важливим для успіху складних проєктів ПЗ. Проте історія довела, що монолітні підходи, які часто генерувалися ранніми інтерпретаціями RUP, не здатні забезпечити сучасні вимоги до масштабованості та швидкості випуску продукту (Time-to-Market). Трансформація через SOA до мікросервісних ландшафтів дозволила реалізувати справжню компонентну незалежність. Сьогодні архітектурна орієнтованість означає проєктування не статичних монолітів, а гнучких, децентралізованих екосистем.

Лекція 24. Майбутнє інженерії програмного забезпечення. Інтеграція підходів AI Engineering у життєвий цикл розробки

Протягом курсу ми розглядали еволюцію процесів розробки: від суворих фаз класичного RUP до гнучких Agile-методологій та автоматизованих пайплайнів DevOps. Наступний еволюційний стрибок індустрії — це інтеграція штучного інтелекту (AI) у сам процес створення програмного забезпечення (SDLC). Ця лекція присвячена тому, як AI Engineering трансформує рутинні завдання, та як зміниться роль інженера-програміста в найближчому майбутньому.

AI Engineering: Нова парадигма розробки

AI Engineering (Інженерія штучного інтелекту) охоплює два взаємопов'язані напрями:

1. Створення програмних систем, що включають AI-компоненти (машинне навчання, великі мовні моделі - LLM).
2. Використання AI як інструменту для проєктування, написання та підтримки традиційного коду.

Перехід до AI Engineering частково змінює природу програмного забезпечення: замість виключно детермінованої логіки (де результат завжди на 100% передбачуваний), системи починають включати імовірнісні моделі, що вимагає нових підходів до архітектури та тестування.

Трансформація аналізу вимог за допомогою AI

У RUP фаза аналізу вимог (Inception) вимагає ідентифікації всіх акторів та прецедентів (Use Cases). Раніше це вимагало десятків годин інтерв'ю та ручного написання специфікацій.

Як AI автоматизує цей процес:

- **Синтез та структурування:** AI може аналізувати неструктуровані нотатки із зустрічей із замовником (розшифровки дзвінків) і автоматично формувати структуровані User Stories або класичні специфікації Use Cases.
- **Виявлення конфліктів:** Сучасні LLM здатні аналізувати сотні сторінок вимог (наприклад, технічних завдань) і підсвічувати логічні суперечності або пропущені граничні сценарії (edge cases) ще до початку розробки.
- **Генерація прототипів:** На основі текстового опису вимог AI-інструменти можуть миттєво генерувати базові UI/UX прототипи (mockups), що значно прискорює отримання зворотного зв'язку від клієнта.

Автоматизація генерації коду та AI Pair Programming

Етап реалізації (Construction у термінології RUP) зазнає найбільших змін завдяки генеративному штучному інтелекту.

- **AI Pair Programming (Парне програмування з ШІ):** Інструменти на кшталт GitHub Copilot, Gemini Code Assist або Tabnine інтегруються безпосередньо в IDE. Вони автодоповнюють цілі блоки коду, генерують функції за текстовим коментарем та пропонують оптимізацію в реальному часі.
- **Від написання коду (Coding) до його направлення (Prompting):** Інженер майбутнього писатиме менше шаблонного (boilerplate) коду. Його головною навичкою стає здатність декомпозувати складну архітектурну задачу на правильні промпти (запити) для AI та критична оцінка згенерованого результату.
- **Автоматичний рефакторинг та модернізація:** AI здатен аналізувати застарілий код (наприклад, написаний на старих версіях мов) і автоматично перекладати його на сучасні фреймворки з дотриманням актуальних патернів проєктування.

Роль AI у тестуванні та контролі якості

Тестування історично було однією з найбільш ресурсоємних дисциплін. ШІ переводить автоматизацію тестування на новий рівень:

- **Генерація Unit-тестів:** AI може миттєво згенерувати повний набір модульних тестів для написаної функції, покриваючи як позитивні, так і негативні сценарії.
- **Self-healing (самовідновлювані) тести:** В автоматизованому тестуванні UI найчастіша проблема — зміна селекторів (наприклад, ID кнопки), через що тести «ламаються». AI-алгоритми здатні автоматично розпізнавати, що елемент просто змінив розташування або атрибут, і самостійно виправляти тестовий скрипт на льоту.
- **Предиктивна аналітика помилок:** Аналізуючи історію комітів та баг-трекер (наприклад, Jira), ML-моделі можуть передбачати, у яких модулях системи найвищий ризик виникнення критичних помилок після нового релізу, дозволяючи команді сфокусувати тестування саме там.

Майбутнє професії Software Engineer

Інтеграція AI не означає зникнення професії програміста. Навпаки, відбувається підвищення рівня абстракції, як це вже було при переході від Асемблера до мов високого рівня (C, Java) і фреймворків.

Нові виклики та компетенції для фахівців 121 спеціальності:

1. **AI-грамотність та критичне мислення:** AI часто «галюцинує» (генерує впевнені, але технічно хибні відповіді) або використовує застарілі бібліотеки. Інженер повинен мати достатню експертизу, щоб робити code review (перевірку) AI-коду.
2. **Фокус на системному проєктуванні (System Design):** Оскільки AI може написати окремий мікросервіс за секунди, головним завданням інженера стає проєктування безпечної, масштабованої та економічно вигідної архітектури всієї системи загалом.
3. **Безпека (Security):** З появою AI з'являються нові вектори атак (наприклад, Prompt Injection) та юридичні ризики, пов'язані з авторським правом на згенерований код.

Висновки

Дисципліна програмної інженерії пройшла шлях від жорстких процесів RUP до швидкісного CI/CD і нині вступає в еру AI-помічників. Проте фундаментальні інженерні принципи залишаються незмінними. Розуміння життєвого циклу розробки, управління вимогами, управління ризиками та проєктування архітектури (те, чого навчає RUP та його нащадки) є тією базою, без якої неможливо ефективно керувати навіть найрозумнішими системами штучного інтелекту. Інструменти змінюються, але інженерне мислення залишається ключовим.

Загальний висновок до Теми 5: Трансформація класичних процесів у сучасні інженерні практики

Підсумовуючи еволюцію методологій розробки програмного забезпечення, можна стверджувати, що індустрія пройшла шлях від суворих, регламентованих підходів (таких як класичний RUP) до гнучких, децентралізованих та високоавтоматизованих екосистем.

Головний урок цієї трансформації полягає в тому, що **фундаментальні інженерні принципи не зникли — вони були оптимізовані та інтегровані в сучасні технології**:

- **Ітеративність та управління ризиками:** Багатомісячні фази та ітерації RUP стиснулися до коротких спринтів у Scrum (через перехідні фреймворки AUP та OpenUP) і досягли свого максимуму в безперервних конвеєрах **CI/CD**, де зміни доставляються користувачеві по кілька разів на день.

- **Архітектурна орієнтованість:** Принцип побудови надійної архітектури на ранніх етапах трансформувався. Від важких монолітів епохи RUP індустрія перейшла до **мікросервісної архітектури**, яка забезпечує незалежність команд, масштабованість та технологічну гнучкість.
- **Автоматизація рутини:** Дисципліни управління конфігураціями та розгортанням, які раніше вимагали значних ручних зусиль, сьогодні повністю автоматизовані завдяки культурі **DevOps, інфраструктурі як коду (IaC)** та контейнеризації (**Docker**).
- **Зміна ролі інженера (AI Engineering):** Впровадження інструментів генеративного штучного інтелекту забирає в розробника левову частку рутинної роботи (написання шаблонного коду, генерація тестів, структурування вимог).

Резюме для майбутніх фахівців (спеціальність 121): Сучасна інженерія програмного забезпечення — це вже не просто вміння писати код. Інструменти та фреймворки (від RUP до Agile, від монолітів до мікросервісів, від ручного тестування до ШІ-помічників) постійно змінюються. Найбільшою цінністю сучасного інженера є **системне мислення**: здатність розуміти весь життєвий цикл створення продукту, проєктувати стійку архітектуру, декомпозувати складні завдання та критично оцінювати результати, згенеровані як людьми, так і штучним інтелектом. База, закладена класичними процесами, залишається тим фундаментом, який дозволяє ефективно керувати цим складним технологічним хаосом.

Перелік контрольних питань до Теми 5

Обмеження класичних процесів та перехід до Agile (за матеріалами Лекції 21)

1. У чому полягала «важковаговість» класичного RUP та які основні проблеми вона створювала для команд розробки?
2. Що таке аналітичний параліч (Analysis Paralysis) і на яких фазах RUP він найчастіше виникав?
3. Як принципи *Agile Manifesto* змінили ставлення до проєктної документації порівняно з предиктивними (каскадними) методологіями?
4. Які ключові відмінності Agile Unified Process (AUP) від класичного Rational Unified Process?
5. Поясніть концепцію «мікро-ітерацій» у фреймворку OpenUP. Як цей процес став мостом до впровадження Scrum?

Еволюція розгортання та DevOps (за матеріалами Лекції 22)

6. Які проблеми класичних дисциплін «Управління конфігураціями» та «Розгортання» вирішує методологія DevOps?

7. Що означає концепція «Інфраструктура як код» (Infrastructure as Code) і як вона допомагає зменшити ризики людської помилки?

8. Розшифруйте абревіатуру CI/CD. Яка різниця між Continuous Delivery та Continuous Deployment?

9. Чому виникає проблема «Працює на моєму комп'ютері» та як технологія контейнеризації (наприклад, Docker) її вирішує?

10. Як пайплайни CI/CD реалізують принцип RUP щодо раннього та постійного зниження ризиків?

Архітектурна еволюція: від монолітів до мікросервісів (за матеріалами Лекції 23)

11. Які основні обмеження монолітної архітектури призвели до необхідності пошуку нових архітектурних стилів?

12. Що таке сервіс-орієнтована архітектура (SOA) та яку роль у ній відіграє інтеграційна шина (ESB)? Чому індустрія частково відмовилася від цього підходу?

13. Назвіть ключові характеристики мікросервісної архітектури (децентралізація, незалежні БД тощо).

14. Як Закон Конвея пов'язаний із мікросервісною архітектурою та концепцією «Two-pizza team»?

15. Як змінилося розуміння принципу RUP «Architecture-centric» (Архітектурна орієнтованість) у сучасних умовах еволюційної архітектури?

AI Engineering та майбутнє розробки (за матеріалами Лекції 24)

16. Як інструменти генеративного штучного інтелекту змінюють підхід до фази збору та аналізу вимог (Inception)?

17. Що таке AI Pair Programming (парне програмування з ШІ) та як воно впливає на етап написання коду?

18. Наведіть приклади, як штучний інтелект автоматизує та покращує процес тестування програмного забезпечення (self-healing тести, генерація unit-тестів).

19. Які нові виклики та ризики (наприклад, безпекові або юридичні) з'являються при використанні AI у життєвому циклі розробки?

20. Які нові компетенції та навички (окрім безпосереднього написання коду) стають критично важливими для інженера-програміста в епоху AI Engineering?

Завдання для самостійного роздуму (для дискусії на семінарі):

- *Оберіть один із чотирьох ключових принципів класичного RUP (ітеративність, архітектурна орієнтованість, керованість прецедентами, управління ризиками) і на конкретному прикладі доведіть, що він не зник, а продовжує існувати в сучасному стеку (Agile + DevOps + AI).*

ЛЕКЦІЯ-СЕМІНАР НА ТЕМУ: «ПРАКТИКА ВИКОРИСТАННЯ ДІАГРАМ У ПРОЄКТУВАННІ АРХІТЕКТУРИ ПЗ»

Вступ

Сьогодні ми розглядаємо тему, яка відрізняє звичайного кодера від Software Architect та Tech Lead — це практика використання діаграм у проєктуванні архітектури програмного забезпечення. Ви вже вмієте писати код. Але на рівні магістратури ми маємо навчитися розуміти, як архітектурні рішення впливають на бізнес, як обґрунтовувати ці рішення перед стейкголдерами та як інтегрувати архітектурну документацію в сучасний процес CI/CD.

1. Проблема невидимості ПЗ та ціна помилки

Давайте почнемо з фундаментальної проблеми нашої індустрії — проблеми невидимості програмного забезпечення. Згадайте відому індійську притчу про сліпих мудреців та слона. Архітектуру програмного забезпечення неможливо «помацати». Коли ви відкриваєте IDE, код показує вам деталі реалізації алгоритму, але приховує загальну картину. В результаті члени команди бачать систему ізольовано: DevOps-інженер бачить її як набір Docker-контейнерів, QA-інженер — як набір тест-кейсів, а розробник бази даних — як набір таблиць.

Чому архітектору важливо створити спільне бачення? Через експоненційне зростання ціни помилки:

- **На рівні написання коду** помилка коштує години вашого часу. Ви написали неправильний SQL-запит або забули звільнити пам'ять — виявили баг, виправили, залили в репозиторій.
- **На рівні архітектури** неправильно обраний патерн (наприклад, синхронний REST там, де потрібен асинхронний брокер) або хибний спосіб комунікації мікросервісів — це місяці рефакторингу та тисячі доларів збитків для компанії.

2. Навіщо ми малюємо діаграми та Головне правило

Раніше, за часів класичних важковагових процесів, намагалися писати велетенські детальні специфікації на сотні сторінок. Зараз, в епоху Agile, документація має бути мінімальною, але вичерпною.

Загалом, ми малюємо діаграми для трьох речей:

1. **Синхронізація команди:** щоб розробник, DevOps та QA розуміли систему однаково.

2. **Документування прийнятих рішень:** фіксація компромісів.
3. **Онбординг:** швидке залучення нових спеціалістів у проєкт.

Але тут ми підходимо до **Головного правила моделювання:** діаграма заради діаграми не має жодного сенсу. Кожна схема повинна відповідати на конкретне запитання певної цільової аудиторії (стейкголдера, розробника, клієнта). Якщо діаграма не дає відповіді на запитання — її не потрібно малювати.

Сьогодні ми фіксуємо архітектуру через **ADR (Architecture Decision Records)** — короткі текстові файли, які пояснюють, *чому* ми прийняли те чи інше рішення (наприклад, чому обрали RabbitMQ замість Kafka). У цих документах діаграми слугують нашою візуальною доказовою базою. Через кілька років найціннішим у такому документі буде не сам факт вибору технології, а описаний контекст та компроміси.

3. Архітектурні фреймворки: від 4+1 до C4

Фахівці рівня Tech Lead мають мислити концептуальними фреймворками, а не просто малювати окремі схеми.

Історично важливою є **Модель «4+1»**, розроблена Філіпом Крученом у 1995 році. Замість того, щоб вмістити всі деталі в одну складну схему, вона розділяє архітектуру на ракурси:

- **Логічне представлення:** функціональність для кінцевого користувача.
- **Представлення процесів:** динаміка, паралелізм, продуктивність під навантаженням.
- **Представлення розробки:** фізична структура коду, організація модулів.
- **Фізичне представлення:** відображення ПЗ на апаратне забезпечення (сервери, мережі).
- **+1 Сценарії (Use Cases):** центральний елемент, який об'єднує всі види.

Проте сьогодні стандартом де-факто для Enterprise-рівня є **Модель C4 (Context, Containers, Components, Code)**. Вона ієрархічна і працює за принципом зумування, як Google Maps:

1. **Рівень 1. Context (Контекст):** Система в оточенні користувачів та зовнішніх систем. Це вид з висоти пташиного польоту, мета якого — спілкування з бізнесом. Тут немає технічних деталей.
2. **Рівень 2. Containers (Контейнери):** З яких додатків, баз даних чи мікросервісів складається система. Це рівень для DevOps. Тут ми показуємо

технології: де крутиться Docker, що використовується для кешу (наприклад, Redis).

3. **Рівень 3. Components (Компоненти):** Внутрішня будова окремого контейнера (модулі, шари). Це рівень розробників та Tech Lead.

4. **Рівень 4. Code (Код):** UML-діаграми класів. На практиці руками їх малюють вкрай рідко, зазвичай вони генеруються автоматично з коду.

4. UML у сучасних розподілених системах

Універсальна мова моделювання (UML) — це класика з чітким синтаксисом. Проте її головний недолік — перевантаженість (14 типів діаграм). У реальних Agile-проєктах ми беремо з UML лише те, що дійсно вирішує проблеми розподілених систем:

- **Діаграма послідовності (Sequence Diagram):** Незамінна для візуалізації патерну Saga у мікросервісах (механізми відкату транзакцій) та документування складних флоу авторизації (наприклад, OAuth 2.0).
- **Діаграма розгортання (Deployment Diagram):** Використовується для планування хмарної інфраструктури, візуалізації відмовостійкості (High Availability) та розподілу навантаження.
- **Діаграма станів (State Machine Diagram):** Ідеальна для моделювання життєвого циклу складних доменних сутностей (наприклад, стан замовлення в e-commerce при асинхронній обробці платежів).

5. Інженерні практики: "Docs as Code"

Як ми документуємо архітектуру на практиці? Графічні редактори (Draw.io, Lucidchart, Miro) класні для швидкого брейнштorму на мітингу. Але фінальна архітектура не повинна малюватися мишкою.

Індустрія перейшла на підхід **Docs as Code (Документація як код)**. Архітектура має жити разом з кодом проєкту. Діаграми пишуться текстом за допомогою спеціального синтаксису, зберігаються в Git, версіонуються і обов'язково проходять Code Review.

- Ми використовуємо **Structurizr DSL** для опису моделі C4. Текстом вказується, що Web App написаний на React, API Gateway — це Nginx, а мікросервіс працює на C++. З цього тексту автоматично генерується схема.
- Ми використовуємо **Mermaid.js** або **PlantUML** прямо у markdown-файлах репозиторію.

Будь-яка зміна архітектурного коду в Pull Request автоматично генерує нову версію діаграми в CI/CD пайплайні. Це гарантує дотримання принципу «Єдиного джерела правди» та рятує від проблеми застарілої документації.

6. Антипатерни та практичні кейси

Проєктуючи документацію, уникайте типових архітектурних помилок:

- **Змішування рівнів абстракції:** не ставте на одну схему бізнес-користувача та технічну таблицю БД з колонками.
- **"Смерть від ліній" (Spaghetti Architecture):** схема не повинна бути перевантажена нечитабельними зв'язками.
- **Відсутність легенди:** кольори та форми повинні бути пояснені.

Розглянемо кілька практичних кейсів для самоперевірки:

1. *Ми додаємо Redis для кешування сесій. На якому рівні C4 ми його відображаємо?* Це Рівень 2 (Containers). Будь-яке сховище, що розгортається окремо — це контейнер.
2. *Ви фіксуєте заміну REST на gRPC в документі ADR. Який розділ буде найціннішим через 3 роки?* "Контекст" та "Наслідки". Без розуміння умов навантаження чи обмежень, саме рішення втрачає сенс.
3. *DevOps-інженер додав новий балансувальник і мишкою домалював його в Confluence. Що він порушив?* Принцип Docs as Code. Зміна мала пройти текстом через Git, щоб схема оновилася синхронно з інфраструктурою.
4. *CEO просить показати UML вашої бази даних. Що ви йому покажете?* Нетехнічному стейкголдеру не потрібні таблиці. Йому потрібен бізнес-контекст і доменні сутності (рівень System Context з C4).

ПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

Ці питання охоплюють ключові аспекти дисципліни та призначені для самоперевірки знань при підготовці до заліку або екзамену.

Основи інженерії ПЗ та RUP:

1. Що таке життєвий цикл програмного забезпечення? Назвіть його основні моделі.
2. Які три основні характеристики (принципи) визначають методологію RUP?
3. Охарактеризуйте мету та основні артефакти фази «Початковий етап» (Inception).
4. У чому полягає різниця між фазою «Уточнення» (Elaboration) та «Побудова» (Construction)?
5. Як RUP підходить до управління ризиками проєкту?

Аналіз та Управління вимогами:

6. Дайте визначення функціональним та нефункціональним вимогам. Наведіть приклади.
7. Що таке модель FURPS+? Розшифруйте кожну літеру абревіатури.
8. Які складові повинна містити якісна текстова специфікація варіанта використання (Use Case Specification)?
9. Для чого використовується Матриця трасування вимог (Requirements Traceability Matrix)?

Проєктування та Архітектура (UML/SOLID/GRASP):

10. У чому полягає різниця між об'єктом і класом в об'єктно-орієнтованому підході? 11. Поясніть сутність патерну BCE (Boundary-Control-Entity).
11. Перелічіть п'ять принципів SOLID та коротко поясніть кожен із них.
12. У чому суть патернів Information Expert та Creator згідно з GRASP?
13. З яких п'яти представлень складається архітектурна модель «4+1» Філіпа Крачтена?
14. Які критерії визначають «хорошу» архітектуру програмної системи (зв'язність, зчеплення, масштабованість)?

Реалізація, Тестування та Впровадження:

15. У чому полягає різниця між верифікацією та валідацією програмного забезпечення?
16. Поясніть концепцію безперервної інтеграції (Continuous Integration).
17. Для чого потрібна діаграма розгортання (Deployment Diagram) та які її основні елементи?
18. Як технології контейнеризації (наприклад, Docker) допомагають уникнути проблеми "На моєму комп'ютері це працює"?

ВИСНОВКИ

Сучасна індустрія інженерії програмного забезпечення розвивається надзвичайно стрімко. Технології, хмарні платформи, фреймворки та мови програмування оновлюються безперервно, проте фундаментальні принципи проєктування надійних, масштабованих та безпечних систем залишаються незмінними.

Дисципліна **«Раціональний уніфікований процес проєктування ПЗ» (RUP)** покликана сформувати у майбутніх інженерів системне архітектурне мислення. Вона вчить керувати вимогами, управляти ризиками та перетворювати хаос бізнес-ідей у чітко структуровані візуальні моделі та якісний програмний код. Базові принципи RUP — *керованість прецедентами, фокус на архітектурі, ітеративність та зниження ризиків* — сьогодні стали генетичним кодом сучасних гнучких методологій (Agile, Scrum) та інженерних практик (DevOps).

Нюанс сучасної індустрії: В епоху, коли генеративний штучний інтелект (Generative AI) здатний миттєво генерувати рутинний вихідний код, найціннішою навичкою інженера стає не синтаксис мови програмування, а **здатність правильно спроєктувати систему, виставити архітектурні обмеження та забезпечити трасованість вимог**.

Протягом перших років роботи розробники зазвичай фокусуються на написанні коду та логіці окремих функцій. Однак із поглибленням професійного досвіду стає очевидним, що понад 60% критичних помилок і "вузьких місць" виникають не через дефекти кодування, а через прорахунки на етапі аналізу вимог та проєктування. Виправлення архітектурних помилок на стадії релізу коштує в сотні разів дорожче, ніж їхнє нівелювання за допомогою вдалого моделювання.

Патерни проєктування (GoF, GRASP) та архітектурні стилі (від класичного MVC до Microservices та Clean/Hexagonal Architecture) — це перевірені часом типові плани інженерних рішень. Проте їхнє успішне застосування неможливе без системного підходу до розробки, який і забезпечує ітеративний життєвий цикл.

Протягом виконання лабораторного практикуму ви пройшли повний шлях створення програмного продукту — від початкового етапу (**Inception**) та архітектурного проєктування (**Elaboration**) до масової побудови (**Construction**) і підготовки до впровадження (**Transition**).

Ви набули практичних навичок у таких напрямах:

- **Управління вимогами:** Проведення аналізу предметної області, ідентифікація стейкхолдерів та класифікація функціональних і нефункциональних вимог (за моделлю **FURPS+**).
- **Візуальне моделювання:** Побудова наскрізних моделей за допомогою мови **UML** та сучасних методологій (включаючи елементи моделі **C4**), документування сценаріїв використання, статичної структури класів та динаміки взаємодії об'єктів.

- **Проєктування стійкої архітектури:** Правильний розподіл обов'язків між компонентами системи із застосуванням принципів **SOLID** та патернів **GRASP** для досягнення високої згуртованості коду (**High Cohesion**) та низької зв'язності підсистем (**Low Coupling**).
- **Робота із сучасним інструментарієм:** Використання CASE-засобів візуального моделювання та середовищ проєктування для автоматизованої генерації базового каркаса коду й забезпечення двоспрямованого трасування (*Forward/Reverse Engineering*).
- **Інтеграція та Розгортання:** Проєктування фізичної топології системи, розуміння ролі контейнеризації (**Docker**) та безперервної інтеграції (**CI/CD пайплайнів**) як сучасної технологічної еволюції дисциплін реалізації та розгортання RUP.
- **Валідація та Ризики:** Ідентифікація та кількісна оцінка технічних ризиків (*Risk Exposure*), побудова матриць трасування та формування стратегії тестування на основі прецедентів для гарантії відповідності продукту бізнес-цілям.

Вектор професійного розвитку

Комплексне бачення життєвого циклу розробки програмного забезпечення та глибокі навички об'єктно-орієнтованого аналізу й проєктування (OOAD) є міцним інженерним фундаментом.

Ці знання дозволять вам упевнено еволюціонувати від позиції рядового програміста-виконавця до рівня **Tech Lead** та **Software Architect** — фахівця, здатного приймати стратегічні технічні рішення, керувати розробкою складних корпоративних систем і створювати програмні продукти, що приносять реальну цінність бізнесу та суспільству.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

Базова література (Foundational & Core Literature)

1. **Kruchten, P.** (2004). *The Rational Unified Process: An Introduction* (3rd ed.). Addison-Wesley Professional. *(Класичне першоджерело, обов'язкове для вивчення філософії та фаз RUP).*
2. **Pressman, R. S., & Maxim, B. R.** (2024). *Software Engineering: A Practitioner's Approach* (10th ed.). McGraw-Hill LLC. *(Найновіше ювілейне видання, що інтегрує класичні процеси із сучасними AI-driven підходами до розробки).*
3. **Sommerville, I.** (2021). *Software Engineering* (11th ed.). Pearson. *(Базовий світовий підручник з інженерії ПЗ, що детально описує управління вимогами та ітеративні моделі).*
4. **Wiegers, K., & Hokanson, C.** (2023). *Software Requirements Essentials: Core Practices for Successful Business Analysis*. Addison-Wesley Professional. *(Актуалізована праця К. Вігерса, яка фокусується на швидкому та ефективному управлінні вимогами в сучасних екосистемах).*
5. **Larman, C.** (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall. *(Найкраще практичне керівництво з аналізу стійкості, патернів GRASP/GoF та декомпозиції BCE).*
6. **Блажко, О. А., & Луценко, С. Г.** (2022). *Інженерія програмного забезпечення: об'єктно-орієнтований аналіз та проєктування систем*. Харків: ХНУРЕ. *(Сучасний вітчизняний підручник, адаптований під стандарти спеціальності 121)*.
7. **Глибовець, М. М.** (2023). *Архітектура програмних систем*. Київ: НаУКМА. *(Навчальний посібник, що розглядає перехід від концептуальних моделей UML до архітектурних стилів)*.

Допоміжна література (Supplementary & Advanced Literature)
Архітектурні стилі та патерни (Design & Architecture)

8. **Martin, R. C. (Uncle Bob)** (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall. *(Обов'язково для вивчення еволюції класів BCE у шари Onion/Clean архітектури).*
9. **Richards, M., & Ford, N.** (2024). *Fundamentals of Software Architecture: An Engineering Approach* (2nd ed.). O'Reilly Media. *(Оновлене видання з чіткими метриками оцінки зв'язності коду та декомпозиції підсистем).*
10. **Brown, S.** (2024). *Visualise, Document and Explore Your Software Architecture with the C4 Model*. Leanpub. *(Практичний посібник від автора моделі C4 для абстрагування над важкими діаграмами UML)*.

11. **Newman, S.** (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.

Процеси, Управління якістю та DevOps (Agile, QA & Deployment)

13. **Farley, D.** (2021). *Modern Software Engineering: Doing What Works to Build Better Software Faster*. Addison-Wesley. *(Фундаментальна праця про безперервну інтеграцію, ітеративність та управління технічним боргом).*
14. **Rubin, K. S.** (2023). *Essential Scrum: A Practical Guide to the Most Popular Agile Process* (2nd ed.). Addison-Wesley. *(Оновлене керівництво, що демонструє трансформацію планів ітерацій RUP у спринти Scrum).*
15. **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. *(Для вивчення сучасної еволюції дисциплін реалізації та розгортання).*

Інформаційні ресурси в мережі Інтернет (Digital Resources & Standards)

16. **Офіційна специфікація UML (Unified Modeling Language)** версії 2.5.1 (та найновіші релізи) від консорціуму OMG (Object Management Group). — Режим доступу: https://www.omg.org/spec/UML/
17. **Офіційний портал методології C4 (The C4 model for visualising software architecture)**. — Режим доступу: https://c4model.com/
18. **ISO/IEC/IEEE 29148:2018 (у чинній редакції)**. *Systems and software engineering — Life cycle processes — Requirements engineering*. Стандарт інженерії вимог. — Режим доступу: https://www.iso.org/standard/72089.html
19. **SWEBOK V4.0 (Software Engineering Body of Knowledge)**. Оновлений звід знань з інженерії програмного забезпечення від IEEE Computer Society, що враховує сучасні практики автоматизації та AI-інженерії. — Режим доступу: https://www.computer.org/volunteering/boards-committees/professional-educational-activities/swebok-v4
20. **Agile Alliance Resource Library**. База знань із гнучких методологій, ретроспектив та ітеративного планування. — Режим доступу: https://www.agilealliance.org/resources/
21. **The Twelve-Factor App**. Методологія проєктування хмароорієнтованих додатків (Cloud-Native Applications), яка є сучасним стандартом для дисципліни розгортання та архітектури підсистем. — Режим доступу: https://12factor.net/uk/ *(доступна українська версія).*