МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ
«ФАХОВИЙ КОЛЕДЖ ПРОМИСЛОВОЇ АВТОМАТИКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
ОДЕСЬКОГО НАЦІОНАЛЬНОГО ТЕХНОЛОГІЧНОГО УНІВЕРСИТЕТУ»

Циклова комісія комп'ютерних наук та інженерії програмного забезпечення

КОНСПЕКТ

з навчальної дисципліни

КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

обов'язкова

Освітньо-професійна програма «Комп'ютерні науки»

Код та найменування спеціальності 122 Комп'ютерні науки

Шифр та найменування галузі знань 12 Інформаційні технології

Мова навчання українська

2025

Розроблено та забезпечується: цикловою комісією Комп'ютерних наук та інженерії програмного забезпечення ВСП «Фаховий коледж промислової автоматики та інформаційних технологій Одеського національного технологічного університету»

Розробник:
Юлія БУРМАКІНА, викладач вищої кваліфікаційної категорії ФКПАІТ ОНТУ

Розглянуто та схвалено на засіданні циклової комісії Комп'ютерних наук та інженерії програмного забезпечення

Протокол № 1 від 27. 08. 2025 р.

Голова циклової комісії _____ Тетяна КОСТИРЕНКО
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

Зміст

Змістовий модуль №1

Конструювання програмного забезпечення.

Тема: Вступ до конструювання програмного забезпечення. Задачі конструювання програмного забезпечення.

Мета: Ознайомитись з поняттям конструювання програмного забезпечення, головними компонентами конструювання програмного забезпечення. Дати визначення стилям конструювання програмного забезпечення. Розглянути моделі конструювання, планування та внесення змін.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Конструювання програмного забезпечення.
2. Стилі конструювання.
3. Завдання, пов'язані з конструюванням.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Що описує термін «конструювання програмного забезпечення»?
2. У чому полягає зв'язок конструювання з проектуванням та тестуванням?
3. З чого складається область знань «Конструювання програмного забезпечення»?
4. Які існують стилі конструювання ПЗ? Чим характеризується кожен стиль?
5. Наведіть конкретні завдання, пов'язані з конструюванням. Чому конструювання ПЗ так важливо?

Розробка програмного забезпечення (ПЗ) - непростий процес, який може включати безліч компонентів. Ось які складові розробки ПЗ визначили вчені за останні 25 років:

- визначення проблеми;
- вироблення вимог;

- створення плану конструювання;
- розробка архітектури ПЗ, або високорівневе проектування;
- детальне проектування;
- кодування та налагодження;
- блочне тестування;
- інтеграційне тестування;
- інтеграція;
- тестування системи;
- коригуючий супровід.

Термін **конструювання програмного забезпечення (ПЗ)** (softwareconstruction) описує детальне створення робочої програмної системи за допомогою комбінації кодування, верифікації (перевірки), модульного тестування (unittesting), інтеграційного тестування і налагодження. На рис.1.1 показано місце конструювання серед інших процесів розробки ПЗ.

Рисунок 1.1 - Місце конструювання серед інших процесів розробки ПЗ

Головними компонентами конструювання є кодування та налагодження, однак воно включає і детальне проектування, тестування по блокам, інтеграційне тестування та інші процеси.

Область «Конструювання ПЗ» (рис. 1.2) тісно пов’язана з іншими областями. Найбільш сильний зв'язок існує з проектуванням (Software Design) і тестуванням (Software Testing). Причиною цього є те, що сам по собі процес конструювання програмного забезпечення зачіпає важливі аспекти діяльності з проектування та тестування. Крім того, конструювання відштовхується від результатів проектування, а тестування (у будь-якій своїй формі) передбачає роботу з результатами конструювання. Досить складно визначити межі між проектуванням, конструюванням і тестуванням, так як всі вони пов'язані в єдиний комплекс процесів життєвого циклу і, залежно від обраної моделі життєвого циклу і застосовуваних методів (методології), такий поділ може виглядати по різному.

Хоча ряд операцій з проектування детального дизайну може відбуватися до стадії конструювання, великий обсяг такого роду проектних робіт відбувається паралельно з конструюванням або як його частина. Це й є суть зв'язку з областю

знань «Проектування програмного забезпечення».

У свою чергу, протягом всієї діяльності з конструювання, інженери використовують модульне і інтеграційне тестування. Таким чином пов'язана дана область знань з «Тестуванням програмного забезпечення».

У процесі конструювання зазвичай створюється більша частина активів програмного проекту - конфігураційних елементів (configuration items). Тому в реальних проектах просто неможливо розглядати діяльність з конструювання у відриві від галузі знань «конфігураційне управління» (Software Configuration Management).

Так як конструювання неможливо без використання відповідного інструментарію і, ймовірно, дана діяльність є найбільш інструментально-насиченою, важливу роль у конструюванні відіграє область знань «Інструменти і методи програмної інженерії» (Software Engineering Tools and Methods).

Безумовно, питання забезпечення якості значимі для всіх галузей знань та етапів життєвого циклу. У той же час, код є основним результуючим елементом програмного проекту. Таким чином, явно присутній зв'язок обговорюваних питань з областю знань «Якість програмного забезпечення» (Software Quality).

З пов'язаних дисциплін програмної інженерії (Related Disciplines of Software Engineering) найбільш тісний зв'язок даної галузі знань існує з комп'ютерними науками (computer scince). Саме в них, зазвичай, розглядаються питання побудови та використання алгоритмів і практик кодування. Нарешті, конструювання стосується і управління проектами (project management), причому, в тій мірі, наскільки діяльність з управління конструюванням важлива для досягнення результатів конструювання.

Конструювання ПЗ - створення працюючого ПЗ з залученням методів верифікації, кодування і тестування компонентів. До інструментів конструювання ПЗ віднесені мови програмування і конструювання, а також програмні методи та інструментальні системи (компілятори, системи управління базами даних, генератори звітів, системи управління версіями, конфігурацією, тестуванням та ін.) До формальних засобів опису процесу конструювання ПЗ (взаємозв'язків між людиною і комп'ютером з урахуванням середовища оточення) віднесені мови конструювання.

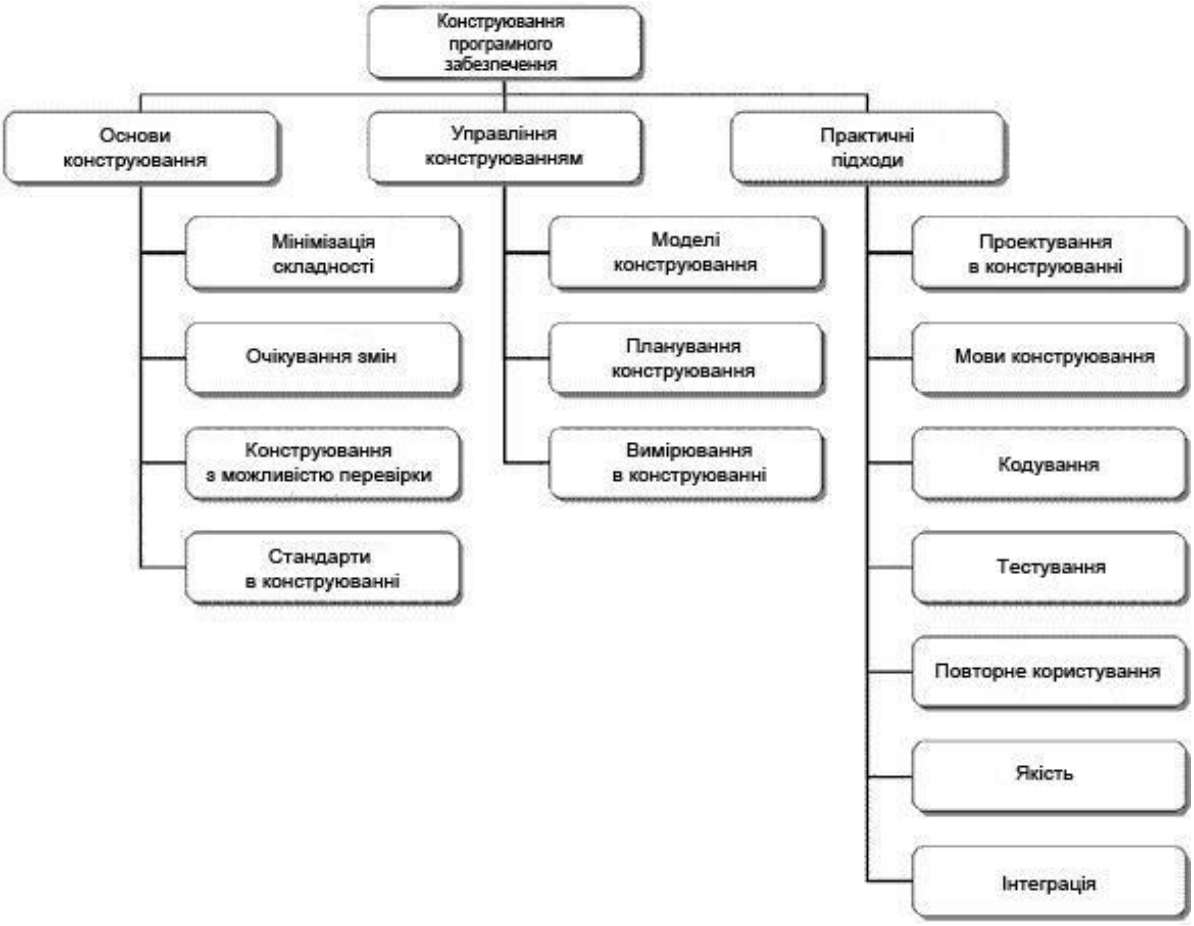


Рисунок 1.2 - Область знань «Конструювання програмного забезпечення»

Область знань «Конструювання ПЗ (Software Construction)» містить наступні розділи:

- зниженняскладності (Reduction in Complexity),
- попередженнявідхиленьвідстилю (Anticipation of Diversity),
- структуризаціядляперевірок (Structuring for Validation),
- використаннязовнішніхстандартів (Use of External Standards).

Основу даної області становлять завдання зниження складності конструювання програмного продукту, попередження відхилень від стилю (лінгвістичного, формального, візуального та ін.), яке забезпечується застосуванням найбільш відповідних стилів конструювання, структуризації ПЗ і використання зовнішніх стандартів.

Лінгвістичний стиль заснований на використанні словесних інструкцій і виразів для перегляду окремих елементів (конструкцій) програм. Він використовується при конструюванні нескладних конструкцій і може бути приведений до виду традиційних функцій і процедур, логічному і функціональному їх програмуванні та ін.

Формальний стиль використовується для точного, однозначного і формального визначення компонентів системи. У результаті його застосування забезпечується конструювання складних систем з мінімальною кількістю помилок, які можуть виникнути у зв'язку з неоднозначністю визначень або узагальнень при конструюванні ПЗ неформальними методами.

Візуальний стиль є найбільш універсальним стилем конструювання ПЗ. Він дозволяє розробникам проекту представляти в наочному вигляді складні програмні конструкції.

Наприклад, графічний інтерфейс користувача звільняє розробника від підбору необхідних координат і властивостей об'єктів інтерфейсу. Візуальна мова проектування UML надає розробнику набір зручних діаграм для завдання статичної та динамічної структури ПЗ.

При застосуванні візуального стилю конструювання створюється текстовий і діаграмний опис структури ПЗ, який виводиться на екран дисплею не тільки для їх розгляду, але і корегування.

У процесі конструювання повинні використовуватися зовнішні стандарти (Ада 95, C++ і ін.), мов опису даних (XML, SQL та ін.), засобів комунікації (COM, CORBA та ін.), інтерфейсів компонентів (POSIX, IDL, APL), сценаріїв UML та ін.

Управління конструюванням базується на моделях конструювання, плануванні та внесенні змін.

Моделі конструювання містять набір операцій, послідовність дій і результати. Види моделей визначаються стандартом життєвого циклу, методологіями та практиками. Основні стандарти орієнтовані на екстремальне програмування і RUP.

Планування полягає у визначенні порядку створення компонентів і методів забезпечення якості. Вимірювання в конструюванні орієнтоване на кількісну оцінку обсягу коду, ступеню використання програмних інформаційних комплексів, ймовірності появи дефектів і кількісних показників якості ПЗ.

Внесення змін проводиться з метою збереження функціональної цілісності системи та рефакторінга коду на основі проведеного метричного аналізу необхідності виконання змін в ПЗ, яке конструюється.

Тестування в конструюванні. Найбільш поширені дві форми тестування створеного коду - модульне і інтеграційне. Види тестування описані в спеціальній галузі знань. При цьому використовуються два стандарти (IEEE 829- 1996 iIEEE 1008-1987)тестування елементів ПЗ і документації. Забезпечення якості конструювання базується не тільки на тестуванні й налагодженні окремих програм, а і на переглядах, інспектуванні, аналізі і оцінках результатів тестування.

Таким чином, розглянуті механізми конструювання дозволяють розробнику проекту прийняти рішення про використання методів конструювання або проектування. Найбільш сучасним вважається метод моделювання UML.

Наведемо деякі конкретні завдання, пов'язані з конструюванням:

1. перевірка виконання умов, необхідних для успішного конструювання;
2. визначення способів подальшого тестування коду;
3. проектування і написання класів та методів;
4. створення і присвоєння імен змінним й іменованим константам;
5. вибір керуючих структур і організація блоків команд;
6. тестування по блокам, інтеграційне тестування і налагодження власного коду;
7. взаємний огляд коду і низькорівневих програмних структур членами групи;
8. «шліфовка» коду шляхом його ретельного форматування

і коментування;

9. інтеграція програмних компонентів, створених окремо;
10. оптимізація коду, спрямована на підвищення його швидкодії, і зниження ступеня використання ресурсів.

Чому конструювання ПЗ так важливо?

Конструювання - крупна частина процесу розробки ПЗ. Залежно від розміру проекту на конструювання зазвичай йде 30-80% загального часу роботи. Все, що займає так багато часу роботи над проектом, неминуче впливає на його успішність.

Конструювання займає центральне місце в процесі розробки ПЗ. Вимоги до додатка і його архітектура розробляються до етапу конструювання, щоб гарантувати його ефективність. Тестування системи (в строгому сенсі незалежного тестування) виконується після конструювання і необхідне для перевірки його правильності. Конструювання - центр процесу розробки ПЗ.

Підвищена увага до конструювання може значно збільшити продуктивність праці. У своєму класичному дослідженні Секман, Еріксон та Грант показали, що продуктивність праці окремих програмістів під час конструювання змінюється в 10-20 разів (Sackman, Erikson, andGrant, 1968).З тих пір ці дані були підтверджені іншими дослідженнями (Curtis, 1981; Mills, 1983; Curtisetal., 1986; Card, 1987; ValettandMcGarry, 1989; DeMarcoandLister,

1999 №; Boehmetal., 2000).

Результат конструювання - вихідний код - часто є єдиним вірним описом програми Найчастіше єдиним видом доступної програмістам документації є сам вихідний код. Специфікації вимог і проектна документація можуть застаріти, але вихідний код актуальний завжди, тому він повинен бути максимально якісним. Послідовне застосування методів поліпшення вихідного коду - ось що відрізняє детальні, коректні і тому інформативні програми.

Конструювання - єдиний процес, який виконується в усіх випадках. Ідеальний програмний проект до початку конструювання проходить стадії ретельної вироблення вимог і проектування архітектури. Після конструювання в ідеалі має бути виконане вичерпне, статистично контрольоване тестування системи. Однак, у реальних проектах нашого недосконалого світу розробники часто пропускають етапи вироблення вимог і проектування, починаючи прямо з конструювання програми. Тестування також часто випадає з розкладу через величезного числа помилок і браку часу. Але яким би терміновим чи погано спланованим не був проект, куди без конструювання подітися? Таким чином, підвищення ефективності конструювання ПЗ дозволяє оптимізувати будь-який проект, яким би недосконалим він не був.

Тема: Планування конструювання програмного забезпечення

Мета: Ознайомитись з основними термінами (потреби, вимога, формалізація постановки завдання, аналіз вимог, специфікація, управління вимогами). Планування конструювання. Блочно-ієрархічний підхід конструювання програмних систем. Визначення рівнів та аспектів проектування. Ознайомитись з процесами технологічного циклу конструювання ПЗ та методами керування і планування проектом. .

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.
II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.
III. Виклад нового матеріалу:

План:

1. Плануванняконструювання програмного забезпечення.
2. Блочно-ієрархічний підхід конструювання програмних систем.
3. Рівні та аспекти проектування.
4. Процеси технологічного циклу конструювання
5. Методи керування і планування проектом.
6. Мережні методи планування і керування. Види планів організації проекту.

IV. Узагальнення та систематизація знань;
V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1.Що таке попередні вимоги? У чому полягає важливість їх виконання?
2.Які існують найпопулярніші типи програмних проектів?
3.Які оптимальні методи слід використовувати при роботі над кожним типом програмного проекту?
4.Що таке попередні умови, пов’язані з визначенням проблеми?
5.Що такепопередні умови, пов'язані з формулюванням вимог?
6. Що таке попередні умови, пов'язані з розробкою архітектури?
7. Скільки часу слід присвятити виконанню попередніх умов?

Розглянемо етапи підготовки до конструювання ПЗ. Як і в будівництві, кінцевий успіх програмного проекту багато в чому визначається до початку конструювання. Якщо фундамент ненадійний або планування виконано недбало, на етапі конструювання ви в кращому випадку зможете тільки звести шкоду до мінімуму.

Важливість виконання попередніх умов

Спільною рисою всіх програмістів, що створюють високоякісне ПЗ, є використання високоякісних методів, що ставлять наголос на якості ПЗ на самому початку, середині і наприкінці проекту.

Якщо ви підкреслюєте якість в кінці проекту, це припадає на етап

тестування системи.

Якщо ви приділяєте підвищену увагу якості всередині роботи над проектом, ви підкреслюєте методи конструювання.

Якщо ви підкреслюєте якість на початку проекту, ви якісно виконуєте планування, визначення вимог і проектування.

Конструювання - середній етап роботи, тому до початку конструювання успіх проекту вже частково зумовлений. І все ж під час конструювання ви хоча б повинні бути в змозі визначити, наскільки вдалою є ваша ситуація, і повернутися назад, якщо це потрібно.

Підготовка до проекту - одна з головних умов ефективного програмування, і це логічно. Обсяг планування залежить від масштабу проекту. З управлінської точки зору, планування передбачає визначення термінів, числа людей і комп'ютерів, необхідних для виконання робіт. З технічної – планування це одержання представлення про створювану систему, що дозволяє не витратити гроші на створення невірної системи.

Дослідження останніх 25 років переконливо довели ефективність правильного виконання проектів з першого кроку і вартість внесення змін, яких можна було уникнути.

Вчені з компаній Hewlett-Packard, IBM, HughesAircraft, TRW та інших організацій виявили, що виправлення помилки до початку конструювання коштує в 10-100 разів дешевше, ніж її усунення в кінці роботи над проектом, під час тестування додатка або після його випуску

Дані про відносну вартість виправлення дефектів в залежності від етапів їх внесення і виявлення (рис. 2.1):

	Час на виявлення дефекту				
Час на внесення дефекту	Формулювання вимог	Проектування архітектури	Конструювання	Тестування системи	Після випуску ПЗ
Формулювання вимог	1	3	5–10	10	10–100
Проектування архітектури	—	1	10	15	25–100
Конструювання	—	—	1	10	10–25

Рисунок 2.1 -Дані про відносну вартість виправлення дефектів в залежності від етапів їх внесення і виявлення

Ці дані говорять, наприклад, про те, що дефект архітектури, виправлення якого при проектуванні архітектури обходиться в 1000 $, може під час тестуванні системи вилитися в 15 000 $ (рис. 2.2).

У більшості проектів основна частина зусиль щодо виправлення дефектів все ще припадає на праву частину (рис. 2.2), а значить, на налагодження і перероблення роботи йде близько 50% часу типового циклу розробки ПЗ. У десятках компаній було виявлено, що політика раннього виправлення дефектів може в два і більше разів знизити фінансові та часові витрати на розробку ПЗ. Це дуже вагомий аргумент на користь раннього знаходження та вирішення проблем.

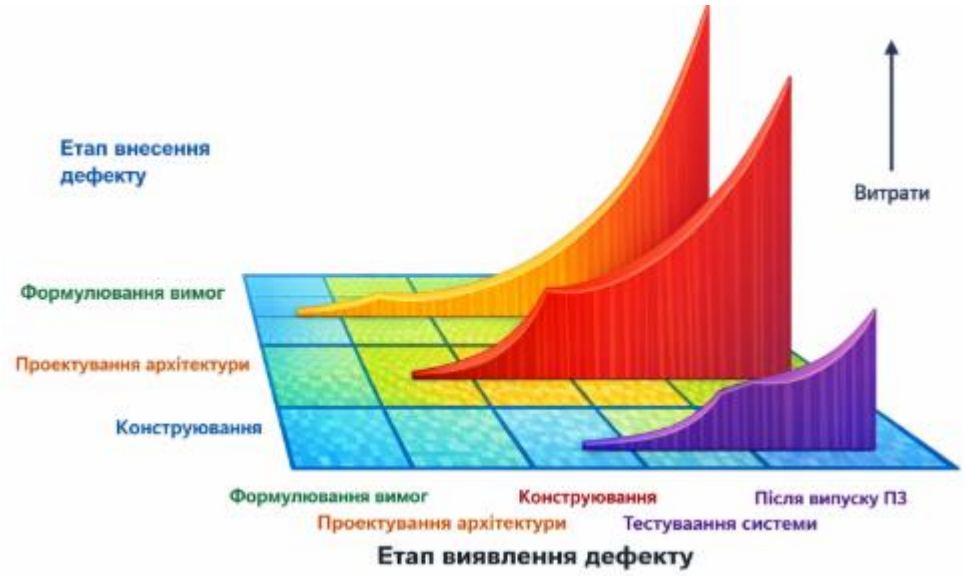


Рисунок 2.2 - Вартість виправлення дефекту в залежності від етапу його виявлення

Визначення типу програмного забезпечення

Узагальнюючи 20 років досліджень розробки ПЗ, Кейперс Джонс, керівник дослідницьких робіт у компанії Software Productivity Research, заявив, що він і його колеги стикалися з 40 різними методами збору вимог, 50 варіантами проектування ПЗ і 30 видами тестування, що застосовувалися в проектах, які реалізуються більш ніж на 700 мовах програмування. Різні типи проектів закликають до різних сполученням підготовки та конструювання. Кожен проект унікальний, проте зазвичай проекти підпадають під загальні стилі розробки. Ось три найпопулярніші типи проектів, а також оптимальні в більшості випадків методи роботи над ними (рис. 2.3).

Працюючи над реальними проектами, ви зіткнетесь з незліченними варіаціями на три теми, зазначені в таблиці, однак, можна зробити і деякі загальні висновки. При розробці бізнес-систем переважно використовувати високоітеративні підходи, при яких планування, формулювання вимог і проектування архітектури перемежовуються з конструюванням, тестуванням системи та гарантією якості. Системи, від яких залежить життя людей, вимагають більш послідовних підходів, оскільки стабільність вимог - одна з умов найвищої надійності системи.

Вплив ітеративних підходів на попередні умови. Дехто стверджує, що при використанні ітеративних методів не потрібно займатися формулюванням попередніх умов, але ця точка зору хибна. Ітеративні підходи послаблюють недоліки неадекватної підготовки, але не усувають їх. При ітеративному підході дефекти виправляються під час розвитку проекту, що дозволяє знизити загальні витрати.

	Тип ПЗ		
	Бізнес-системи	**Системи цільового призначення**	**Вбудовані системи, від яких залежить життя людей**
Типові програми	Інтернет-сайти. Сайти в інтрамережах. Системи управління матеріально-технічним постачанням. Ігри. Система управління інформацією. Система виплати заробітної плати.	Вбудоване ПЗ. Ігри. Інтрнет-сайти. Пакетне ПЗ. Програмні інструменти. Web-сервіси.	Авіаційне ПЗ. Вбудоване ПЗ. ПЗ для медичних пристроїв. Операційні системи. Пакетне ПЗ.
Моделі життєвого циклу	Гнучка розробка (екстремальне програмування, методологія Scrum, розробка на основі тимчасових вікон і т.д.). Еволюційне прототипування.	Поетапне постачання. Еволюційне постачання. Спіральна розробка.	Поетапне постачання. Еволюційне постачання. Спіральна розробка.
Планування і управління	Інкрементне планування проекту. Планування тестування і гарантії якості за необхідністю. Неформальний контроль за змінами.	Базове завчасне планування. Базове планування тестування. Планування тестування і гарантії якості за необхідністю. Формальний контроль за змінами.	Вичерпне завчасне планування. Вичерпне планування тестування. Вичерпне планування гарантії якості. Ретельний контроль за змінами.
Формулювання вимог	Неформальна специфікація вимог	Напівформальна специфікація вимог. Огляди вимог у міру потреби.	Формальна специфікація вимог. Формальні інспекції вимог.
	Бізнес-системи	**Призначення**	**Життя людей**
Проектування	Комбінація проектування і кодування.	Проектування архітектури. Неформальне детальне проектування. Огляди проекту за необхідності	Проектування архітектури. Формальні інспекції архітектури. Формальне детальне проектування. Формальні інспекції детального проекту.
Конструювання	Парне або індивідуальне програмування. Неформальна процедура реєстрації.	Парне або індивідуальне програмування. Неформальна процедура реєстрації коду. Огляди коду за необхідності.	Парне або індивідуальне програмування. Формальна процедура реєстрації коду. Формальні інспекції коду.
Тестування і гарантія якості	Розробники тестують власний код. Попередня розробка. тестів. Тестування окремої групи проводиться в невеликому обсязі або не проводиться взагалі.	Розробники тестують власний код. Попередня розробка. тестів. Окрема група тестування.	Розробники тестують власний код. Попередня розробка. тестів. Окрема група тестування. Окрема група гарантії якості.
Впровадження програми	Неформальна процедура впровадження.	Формальна процедура впровадження.	Формальна процедура впровадження.

Рисунок 2.3 - Три найпопулярніші типи проектів

Ітеративний проект із скороченою програмою виконання попередніх умов або без неї відрізняється від аналогічного послідовного проекту

двома аспектами. По-перше, при ітеративному підході витрати на виправлення дефектів зазвичай нижче, тому що дефекти виявляються раніше. І все ж це відбувається в кінці кожної ітерації, і виправлення дефектів вимагає повторного проектування, кодування і тестування фрагментів ПЗ, що робить витрати більшими, ніж вони могли б бути (рис. 2.4).

	Підхід 1: послідовний підхід без виконання виконання попередніх умов		Підхід 2: ітеративний підхід без виконання попередніх умов	
Рівень завершенності проекту	**Витрати на роботу**	**Витрати на виправлення дефектів**	**Витрати на роботу**	**Витрати на виправлення дефектів**
20%	$100 000	$0	$100 000	$75 000
40%	$100 000	$0	$100 000	$75 000
60%	$100 000	$0	$100 000	$75 000
80%	$100 000	$0	$100 000	$75 000
100%	$100 000	$0	$100 000	$75 000
Витрати на виправлення дефектів в кінці проекту	$0	$500 000	$0	$0
СУМА	$500 000	$500 000	$500 000	$375 000
ЗАГАЛЬНА СУМА	$1 000 000		$875 000	

Рисунок 2.4 – Витрати на розробку проекту при послідовному та ітеративному підходах без виконання попередніх умов

По-друге, при ітеративних підходах витрати розподіляються по всьому проекту, а не групуються в його кінці. Зрештою і при ітеративних, і при послідовних підходах загальна сума витрат виявиться схожою, але в першому випадку вона не здаватиметься настільки великої, бо буде сплачена по частинах.

Адекватна увага до виконання попередніх умов дозволяє знизити витрати незалежно від типу підходу, що використовується (рис. 2.5). Ітеративні підходи зазвичай за багатьма параметрами краще послідовних, однак ітеративний підхід, при якому нехтують попередніми умовами, може в підсумку виявитися набагато дорожчим, ніж належним чином підготовлений послідовний проект.

	Підхід 3: послідовний підхід з виконанням попередніх умов		Підхід 4: ітеративний підхід з виконанням попередніх умов	
Рівень завершенності проекту	**Витрати на роботу**	**Витрати на виправлення дефектів**	**Витрати на роботу**	**Витрати на виправлення дефектів**
20%	$100 000	$20 000	$100 000	$10 000
40%	$100 000	$20 000	$100 000	$10 000
60%	$100 000	$20 000	$100 000	$10 000
80%	$100 000	$20 000	$100 000	$10 000
100%	$100 000	$20 000	$100 000	$10 000
Витрати на виправлення дефектів в кінці проекту	$0	$0	$0	$0
СУМА	$500 000	$100 000	$500 000	$50 000
ЗАГАЛЬНА СУМА	$600 000		$550 000	

Рисунок 2.5 – Витрати на розробку проекту при послідовному та ітеративному підходах з виконанням попередніх умов

Попередні умови, пов'язані з визначенням проблеми

Перша попередня умова, яку потрібно виконати перед конструюванням, - ясне формулювання проблеми, яку система повинна вирішувати. Це ще іноді називають «баченням продукції», «формулюванням точки зору», «формулюванням завдання» або «визначенням продукції». Визначення проблеми передує виробленню детальних вимог, яке є більш глибоким дослідженням проблеми (рис. 2.6).

Рисунок 2.6 - Визначення проблеми

Проблему слід формулювати мовою, зрозумілою користувачеві, а сама проблема повинна бути описана з користувацької точки зору.

Попередні умови, пов'язані з формулюванням вимог

Вимоги докладно описують, що повинна робити програмна система, а їх формулювання - перший крок до вирішення проблеми. Формулювання вимог також відоме як «розробка вимог», «аналіз вимог», «аналіз», «визначення вимог», «специфікація», «функціональна специфікація».

Важливість явного набору вимог пояснюється кількома причинами:

1. Явні вимоги допомагають гарантувати, що функціональність системи визначається користувачем, а не програмістом.

2. Увага до вимог допомагає звести до мінімуму зміни системи після початку розробки. Виявивши при кодуванні помилку в коді, ви зміните кілька рядків, і робота продовжиться. Якщо ж під час кодування ви знайдете помилку у вимогах, доведеться змінити проект програми, щоб він відповідав зміненим вимогам. Можливо, при цьому доведеться відмовитися від частини старого проекту, а оскільки відповідно до неї вже написаний деякий код, на реалізацію нового проекту піде більше часу, ніж могло б. Ви також повинні будете відмовитися від коду і тестів, на які вплинула зміна вимог, і написати їх наново. Навіть код, що

залишився недоторканим, потрібно буде заново протестувати для гарантії того, що зміна не призвела до появи нових помилок.

Адекватне визначення вимог - одна з найважливіших умов успіху проекту, можливо, навіть більш важливе, ніж використання ефективних методів конструювання.

Дослідження, проведені в IBM і інших компаніях, показали, що при реалізації середнього проекту вимоги під час розробки змінюються приблизно на 25%, на що припадає 70-85% обсягу повторної роботи над типовим проектом.

Наступні дії дозволяють максимально легко перенести зміни вимог під час конструювання:

1. Оцініть якість вимог за допомогою контрольного списку.
2. Переконайтесь, що всім відома ціна зміни вимог.
3. Задайте процедуру контролю змін.
4. Використовуйте ті підходи до розробки, які адаптуються до змін.
5. Пам'ятайте про бізнес-моделі проекту.

Попередні умови, пов'язані з розробкою архітектури

Архітектура - це високорівнева частина проекту програми, каркас, що складається з деталей проекту. Архітектуру також називають «архітектурою системи», «високорівневим проектом» і «проектом високого рівня».

Якість архітектури визначає концептуальну цілісність системи, яка в свою чергу визначає підсумкову якість системи. Якісна архітектура надає структуру, потрібну для підтримки концептуальної цілісності в масштабі системи. Вона надає програмістам керівництво, рівень детальності якого відповідає їх навичкам і виконуваній роботі. Вона дозволяє розділити роботу на частини, над якими окремі розробники і групи можуть працювати незалежно.

Якісна архітектура полегшує конструювання. Погана архітектура робить його майже неможливим.

Внесення змін в архітектуру при конструюванні і на подальших етапах коштує недешево. Час, необхідний для виправлення помилки в архітектурі ПЗ, дорівнює часу, потрібному для виправлення помилки у вимогах, тобто перевищує часові витрати на виправлення помилки в коді. Зміни архітектури схожі на зміни вимог ще й тим, що невеликі зміни можуть мати великі наслідки. Чим би не були обумовлені зміни архітектури - виправленням помилок або внесенням покращень, - чим раніше ви усвідомлюєте їх необхідність, тим краще.

Типові компоненти архітектури:

1. Організація програми. У першу чергу архітектура повинна містити загальний опис системи. Архітектура має містити підтвердження того, що при її розробці були розглянуті альтернативні варіанти, і обґрунтовувати вибір остаточної організації системи. Архітектура має визначати основні компоненти програми. Залежно від розміру програми її компонентами можуть бути окремі класи або підсистеми, що складаються з декількох класів. Кожен компонент є класом або набором класів / методів, які в сукупності реалізують високорівневі функції програми, такі як: взаємодія з користувачем, відображення Web-

сторінок, інтерпретація команд, інкапсуляція бізнес-правил або доступ до даних. За кожну функцію програми, зазначену у вимогах, повинен відповідати хоча б один компонент. Якщо функцію реалізують декілька компонентів, вони повинні співпрацювати, а не конфліктувати.

Архітектура має чітко визначати відповідальність кожного компонента. Компонент повинен мати одну область відповідальності і якомога менше знати про області відповідальності інших компонентів.

Архітектура повинна ясно визначати правила комунікації для кожного компонента. Вона повинна описувати, які інші компоненти даний компонент може використовувати безпосередньо, які побічно, а які взагалі не повинен використовувати.

2. Основні класи. Архітектура має визначати основні класи, додатки, їх галузі відповідальності та механізми взаємодії з іншими класами. Вона повинна описувати ієрархії класів, зміни станів і час існування об'єктів. Якщо система досить велика, архітектура повинна описувати організацію класів у підсистеми. При цьому не всі класи системи потрібно описувати в специфікації архітектури. Орієнтуйтеся на правило 80/20: описуйте 20% класів, якими на 80% визначається поведінка системи).

3. Організація даних. Архітектура повинна описувати основні види формату файлів і таблиць. Вона повинна описувати розглянуті альтернативи і обґрунтовувати підсумкові варіанти. Якщо програма використовує список ідентифікаторів клієнтів і розробники архітектури вирішили реалізувати його за допомогою списку з послідовним доступом, в документації має бути вказано, чому цей вид списку краще, ніж список з довільним доступом, стек або хеш- таблиця. Ця інформація надасть вам неоціненну допомогу під час конструювання й супроводу програми, підказавши, чим керувалися розробники архітектури. Без неї ви будете почувати себе глядачем, який дивиться іноземний фільм без субтитрів.

Прямий доступ до даних звичайно треба надавати тільки одній підсистемі або класу; винятки можливі при використанні класів або методів доступу, що забезпечують доступ до даних контрольованим абстрактним чином.

Архітектура має визначати високорівневу організацію та утримання всіх баз даних (БД), що використовуються.

4. Бізнес-правила. Архітектура, залежна від специфічних бізнес-правил. Вона повинна їх визначати і описувати їх вплив на проект системи. Візьмемо для прикладу бізнес-правило, згідно з яким інформація про клієнтів повинна застарівати не більше ніж на 30 секунд. В даному випадку в специфікації архітектури має бути зазначено, як це правило вплинуло на вибір методу забезпечення актуальності даних та їх синхронізації.

5. Інтерфейс користувача. Інтерфейс користувача (GUI) часто проектується на етапі формулювання вимог. Якщо це не так, його слід визначити на етапі розробки архітектури. Архітектура повинна описувати головні елементи формату Web-сторінок, GUI, інтерфейс командного рядка і т.д.

Комфортність GUI може в підсумку визначити популярність або провал програми.

Архітектура має бути модульною, щоб GUI можна було змінити, не торкнувшись бізнес-правил і модулів програми, що відповідають за виведення даних. Наприклад, архітектура повинна забезпечувати можливість порівняно легкої заміни групи класів інтерактивного інтерфейсу на групу класів інтерфейсу

командного рядка. Така можливість дуже корисна; багато в чому це пояснюється тим, що інтерфейс командного рядка зручний для тестування ПЗ на рівні блоків або підсистем.

6. Управління ресурсами. Архітектура повинна включати план управління обмеженими ресурсами, такими як з'єднання з БД, потоки і дескриптори. При розробці драйверів, вбудованих систем та інших програм, які будуть працювати в умовах обмеженої пам'яті, архітектура повинна також визначати спосіб управління пам'яттю. Архітектура має містити оцінку ресурсів, що використовуються в номінальному режимі і при екстремальному навантаженні.

7. Безпека. Архітектура має визначати підхід до безпеки на рівні проекту програми та на рівні коду. Якщо модель погроз до сих пір не розроблена, це слід зробити при проектуванні архітектури. Про безпеку потрібно пам'ятати і при розробці принципів кодування, в тому числі методик обробки буферів і ненадійних даних (даних, що вводяться користувачами, файлів «cookie», конфігураційних даних і даних інших зовнішніх інтерфейсів), підходів до шифрування, рівню повідомлень про помилки, захист секретних даних, що знаходяться в пам'яті, та інших питань.

8. Продуктивність. У вимогах слід визначити показники продуктивності. Якщо вони пов'язані з використанням ресурсів, треба визначити пріоритети для різних ресурсів, в тому числі співвідношення швидкодії, використання пам'яті і витрат.

Архітектура повинна включати оцінки продуктивності і пояснювати, чому розробники архітектури вважають ці показники досяжними. Якщо вони можуть бути не досягнуті, це теж повинно бути відображено в архітектурі. Якщо для досягнення деяких показників потрібні специфічні алгоритми або типи даних, також вкажіть це в специфікації архітектури. Крім того, в архітектурі можна вказати обсяг простору і час, що виділяються кожному класу або об'єкту.

9. Масштабованість. Масштабованістю називають можливість системи адаптуватися до зростання вимог. Архітектура має описувати, як система буде реагувати на зростання числа користувачів, серверів, мережевих вузлів, записів у БД, транзакцій і т. д. Якщо розвиток системи не передбачається і її масштабованість не грає ролі, це має бути явно вказано в архітектурі.

10. Взаємодія з іншими системами. Якщо деякі дані або ресурси будуть загальними для розроблюваної системи та інших програм або пристроїв, в архітектурі потрібно вказати, як це буде реалізовано.

11. Інтернаціоналізація / локалізація. «Інтернаціоналізацією» називають реалізацію у програмі підтримки регіональних стандартів. «Локалізацією» називають переклад інтерфейсу програми і реалізацію в ній підтримки конкретної мови.

12. Введення-виведення. Архітектура має визначати схему читання даних: попередній виклик, читання із затримкою або на вимогу. Крім того, вона повинна описувати рівень, на якому визначатимуться помилки введення- виведення: на рівні полів, записів, потоків даних або файлів.

13. **Обробка помилок.**

14. Відмовостійкість. При розробці архітектури системи слід вказати очікуваний рівень її відмовостійкості.

15. Можливість реалізації архітектури. Архітектура має підтверджувати, що система технічно здійсненна. Якщо неможливість реалізації

якогось компонента може зробити проект непрацездатним, в архітектурі має бути відображено, як вивчалися ці питання: за допомогою прототипів, досліджень чи інакше. Ці аспекти ризику слід усунути до початку повномасштабного конструювання.

16. Надлишкова функціональність. Надійністю називають здатність системи продовжувати роботу після виявлення помилки. Дуже часто в специфікації архітектури розробники визначають більш надійну систему, ніж вказано у вимогах. Одна з причин цього полягає в тому, що система, яка складається з багатьох частин, що задовольняють мінімальним вимогам до надійності, в цілому може виявитися менш надійною, ніж потрібно. У специфікації архітектури має бути явно вказано, чи можуть програмісти реалізувати у своїх блоках програми надлишкову функціональність або вони повинні створити найпростішу працездатну систему.

Визначити відношення до реалізації надлишкової функціональності особливо важливо тому, що багато програмісти роблять це автоматично, з почуття професійної гідності. Явно висловивши очікування в архітектурі, ви зможете уникнути феномена, при якому деякі класи виключно надійні, а інші лише відповідають вимогам.

17. Повторне використання. Якщо план передбачає застосування існуючого коду, тестів, форматів даних і т. д., архітектура повинна пояснювати, як повторно використані ресурси будуть адаптовані до інших архітектурних особливостей.

18. Стратегія змін. Архітектура повинна підтверджувати, що можливі покращення розглядалися. Якщо вірогідні зміни форматів введення або виведення даних, стилю взаємодії з користувачами або вимог до обробки, архітектура повинна показувати, що всі ці зміни були передбачені і кожна з них буде обмежена невеликим числом класів. Архітектурний план внесення змін може бути зовсім простим: включити у файли даних номери версій, зарезервувати поля на майбутнє, спроектувати файли так, щоб в них можна було додати нові таблиці і т. д. Якщо застосовується генератор коду, архітектура повинна показувати, що він підтримує можливість внесення передбачуваних змін.

В архітектурі повинні бути відображені стратегії, які дозволяють програмістам не обмежувати наявні у них вибір завчасно. Так, архітектура

може визначати, що замість жорстко закодованих тестів буде застосовуватися метод, заснований на перевірці таблиць. Дані таблиць можна зберігати в зовнішньому файлі, а не включати в програму, що дозволить вносити до неї зміни без перекомпіляції.

Скільки часу слід присвятити виконанню попередніх умов

Час, що йде на визначення проблеми, формулювання вимог і проектування архітектури ПЗ, залежить від особливостей проекту. Як правило, якщо проект розвивається без проблем, робота над вимогами, архітектурою проекту і попереднім плануванням поглинає 10-20% зусиль і 20_30% часу. Ці показники не включають витрати на детальне проектування - воно є частиною конструювання.

Тема: Життєвий цикл програмного забезпечення. Моделі життєвого циклу програмного забезпечення.

Мета: ознайомитись з поняттями життєвого циклу програмного забезпечення. Знати стадії створення програмного забезпечення. Вивчити поняття моделі життєвого циклу програмного забезпечення та типи моделей життєвого циклу програмного забезпечення.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Життєвий цикл програмного забезпечення.
2. Стадії створення програмного забезпечення.
3. Моделі життєвого циклу програмного забезпечення

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Що таке життєвий цикл програмного забезпечення?
2. Які виділяють стадії створення програмного забезпечення?
3. Які існують моделі життєвого циклу програмного забезпечення

У процесі створення ПЗ можна виділити 4 базових етапи/стадії (рис. 2.1):

- *специфікація* – визначення основних вимог.
- *розроблення* – створення ПЗ відповідно до специфікацій.
- *тестування* – перевірка ПЗ на відповідність вимогам клієнта.
- *супровід/модернізація* – розвиток ПЗ відповідно до змін потреб замовника.

Рисунок 2.1 - Схема процесу створення ПЗ

Перехід від ручних засобів розроблення ПЗ до промислового виробництва програм потребував розвитку теоретичних основ розроблення ПЗ. Постійна необхідність внесення змін у програми як спричинена помилками, так і розвитком вимог до них, є принциповою властивістю програмного забезпечення. Діяльність, пов’язана з рішенням широкого ряду завдань для постійного розвитку, отримала назву супроводу програмного забезпечення. Якщо зусилля, спрямовані на модернізацію ПЗ, перевищують вигоду від його використання, говорять про моральне старіння програм.

Життєвий цикл програмного забезпечення - період часу, що починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації. Цей цикл - процес побудови і розвитку ПЗ.

Модель ЖЦ ПЗ - це структура, що визначає послідовність виконання і взаємозв'язок процесів, дій, задач протягом ЖЦ, яка залежить від специфіки, масштабу і складності проекту та особливостей умов, за яких система створюється та функціонує.

Стадія створення ПЗ - це частина процесу створення ПЗ, що обмежена певними часовими рамками і завершується випуском конкретного продукту (моделей ПЗ, програмних компонентів, документації).

Модель ЖЦ абстрактно представляє реальний процес, в ній відсутні деталі, несуттєві з точки зору призначення моделі.

Поняття ЖЦ виникло під впливом потреби у систематизації робіт у процесі розробки ПЗ. Систематизація була першим етапом на шляху до автоматизації процесу розроблення ПЗ.

Наступними кроками переходу до автоматизації процесу розроблення ПЗ

були такі: встановлення технологічних маршрутів діяльності розробників ПЗ, визначення можливості їх автоматизації та виявлення ризиків, розробки інструментів для автоматизації.

Коротко розглянемо етапи розвитку програмування, щоб краще зрозуміти рушійні сили і перспективи подальшої еволюції технології програмування.

Перший етап - «Стихійне» програмування.

Перший етап охоплює період від моменту появи перших обчислювальних машин до середини 60-х рр. XX ст. У цей період практично були відсутні сформульовані технології, і програмування фактично було мистецтвом.

Перші програми мали найпростішу структуру. Вони складалися з власне програми на машинній мові і оброблюваних нею даних (рис. 2.2). Складність програм в машинних кодах обмежувалася здатністю програміста одночасно подумки відстежувати послідовність виконуваних операцій і місцезнаходження даних при програмуванні.

У 50-ті роки потужність комп'ютерів першого покоління була невелика, а програмування для них велося переважно в машинному коді. Головним чином вирішувалися науково-технічні завдання оборонного характеру, при цьому завдання на програмування містило, як правило, досить точну математичну постановку задачі. Наприклад, розрахунок траєкторії польоту ракети з урахуванням безлічі супутніх чинників. Використовувалася інтуїтивна технологія програмування, коли відразу ж приступали до складання програми по вихідному завданням, при цьому часто саме завдання кілька разів змінювалося, що сильно збільшувало час розробки програми. Мінімальна документація оформлялася вже після того, як програма починала працювати. Поява асемблерів дозволила замість довічних або шістнадцяткових кодів використовувати символічні імена даних і мнемоніки кодів операцій. В результаті програми стали більш чіткими та зручними для «читання».

Проте, саме в цей період народилася фундаментальна для технології програмування концепція модульного програмування, орієнтована на подолання труднощів програмування в машинному коді. З'явилися перші мови програмування високого рівня.

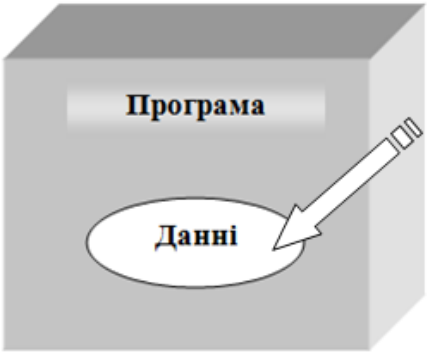


Рисунок 2.2 - Структура перших програм

Створення мов програмування високого рівня, таких як ФОРТРАН і ALGOL, істотно спростило програмування обчислень, знизивши рівень деталізації операцій. Це, в свою чергу, дозволило збільшити складність програм.

Революційною була поява в мовах засобів, що дозволяють оперувати

підпрограмами. Ідея написання підпрограм з'явилася набагато раніше, але відсутність засобів підтримки в перших мовних засобах істотно знижувало ефективність їх застосування. Підпрограми можна було зберігати і використовувати в інших програмах. В результаті були створені величезні бібліотеки розрахункових і службових підпрограм, які в міру потреби викликалися з розроблюваної програми.

В наслідок підвищення потужності комп'ютерів і накопичення досвіду програмування на мовах високого рівня швидко росла складність розв'язуваних на комп'ютерах завдань, в результаті чого виявилася обмеженість мов, які проігнорували модульну організацію програм. Крім того, стало зрозуміло, що важливо не тільки те, якою мовою ми програмуємо, але і те, як ми програмуємо. Поява в комп'ютерах другого покоління переривань призвело до розвитку мультипрограмування і створення великих програмних систем. Широко стала використовуватися колективна розробка, яка, однак, розкрила ряд серйозних технологічних проблем.

Типова програма того часу складалася з основної програми, області глобальних даних і набору підпрограм (в основному бібліотечних), що виконують обробку всіх даних або їх частини (рис. 2.3).

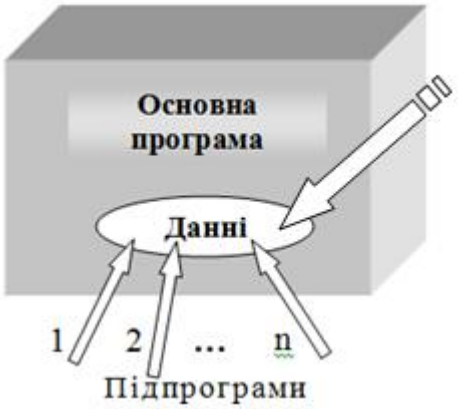


Рисунок 2.3 - Архітектура програми з глобальною областю даних

Слабким місцем такої архітектури було те, що при збільшенні кількості підпрограм зростала ймовірність спотворення частини глобальних даних будь-якої підпрограмою. Наприклад, підпрограма пошуку коренів рівняння на заданому інтервалі за методом ділення, відрізка навпіл змінює величину інтервалу. якщо при виході з підпрограми не передбачити відновлення початкового інтервалу, то в глобальній області виявиться невірне значення інтервалу.

Щоб скоротити кількість таких помилок, було запропоновано в підпрограмі розміщувати локальні дані (рис. 2.4).

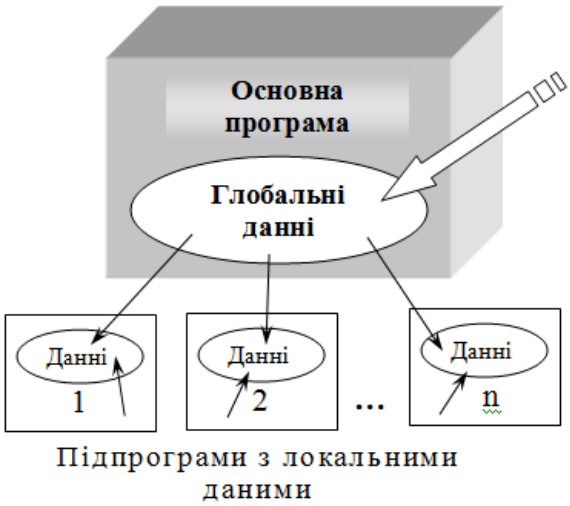


Рисунок 2.4 Архітектура програми, що використовує підпрограми з локальними даними

У відсутності чітких моделей опису підпрограм і методів їх проектування створення кожної підпрограми перетворювалося в непросту задачу, інтерфейси підпрограм виходили складними, і при складанні програмного продукту виявлялося велика кількість помилок узгодження. Виправлення таких помилок, як правило, вимагало серйозної зміни вже розроблених підпрограм, що ще більш ускладнювало ситуацію, так як при цьому в програму часто вносилися нові помилки, які також необхідно було виправляти. Аналіз причин виникнення більшості помилок дозволив сформулювати новий підхід до програмування, який був названий «структурним».

Другий етап - структурний підхід до програмування (60 - 70-ті рр. XX ст.)

Структурний підхід до програмування являє собою сукупність рекомендованих технологічних прийомів, що охоплюють виконання всіх етапів розробки програмного забезпечення.

В основі структурного підходу лежить декомпозиція (розбиття на частини) складних систем з метою подальшої реалізації у вигляді окремих невеликих (до 40 - 50 операторів) підпрограм. З появою інших принципів декомпозиції (об'єктного, логічного і т.д.) даний спосіб отримав назву процедурної декомпозиції.

У 70-ті роки набули широкого поширення інформаційні системи і бази даних. Почався інтенсивний розвиток технології програмування, перш за все, в таких напрямках:

- обґрунтування і широке впровадження низхідній розробки та структурного програмування;
- розвиток абстрактних типів даних і модульного програмування, зокрема, виникнення ідеї поділу специфікації і реалізації модулів і використання модулів, що приховують структури даних;
- дослідження проблем забезпечення надійності та мобільності ПЗс;
- створення методики управління колективною розробкою ПЗс;
- поява інструментальних програмних засобів підтримки технології програмування.

На відміну від використовуваного раніше процедурного підходу до декомпозиції, структурний підхід вимагав уявлення завдання у вигляді ієрархії

підзадач найпростішої структури. Проектування, таким чином, здійснювалося «зверху-донизу» і мало на увазі реалізацію загальної ідеї, забезпечуючи опрацювання інтерфейсів підпрограм. Одночасно вводилися обмеження на конструкції алгоритмів, рекомендувалися формальні моделі їх опису, а також спеціальний метод проектування алгоритмів - метод покрокової деталізації.

Підтримка принципів структурного програмування була закладена в основу так званих процедурних мов програмування. Як правило, вони включали основні «структурні» оператори передачі управління, підтримували вкладення підпрограм, локалізацію та обмеження області

«видимості» даних. Серед найбільш відомих мов цієї групи варто назвати PL/1, ALGOL-68, Pascal, C.

Одночасно зі структурним програмуванням з'явилася величезна кількість мов, які базуються на інших концепціях, але більшість з них не витримало конкуренції. Якісь мови були просто забуті, ідеї інших були в подальшому використані в наступних версіях розвиваються мов.

Подальше зростання складності і розмірів розроблюваного програмного забезпечення зажадав розвитку структурування даних. Як наслідок цього в мовах з'являється можливість визначення користувацьких типів даних. Одночасно посилилося прагнення розмежувати доступ до глобальних даних програми, щоб зменшити кількість помилок, що виникають при роботі з глобальними даними. В результаті з'явилася і почала розвиватися технологія модульного програмування.

Використання модульного програмування суттєво спростило розробку програмного забезпечення декількома програмістами. Тепер кожен із них міг розробляти свої модулі незалежно, забезпечуючи взаємодію модулів через спеціально домовлені міжмодульні інтерфейси. Крім того, модулі в майбутньому без змін можна використовувати в інших розробках, що підвищило продуктивність праці програміста.

Практика показала, що структурний підхід разом із модульним програмуванням дозволяє отримувати достатньо надійні програми, розмір котрих не перевищує 100000 операторів. Вузьким місцем модульного програмування є те, що помилка в інтерфейсі при викликанні підпрограми виявляється тільки при виконанні програми (із-за роздільної компіляції модулів знайти помилки раніше неможливо). При збільшенні розміру програми звичайно виростає складність модульних інтерфейсів, і з певного моменту передбачити взаємодію окремих частин програми стає практично неможливо. Для розробки програмного забезпечення великого об’єму було запропоновано використовувати об’єктний підхід.

Третій етап – об’єктний підхід до програмування (з середини 80-х до кінця 90-х років XX ст.).

У 80-х роках було характерно широке впровадження персональних комп'ютерів в усі сфери людської діяльності і, тим самим, створення великого і різноманітного контингенту користувачів ПЗс. Це призвело до бурхливого розвитку користувальницьких інтерфейсів і створенню чіткої концепції якості ПЗс. Розвиваються методи та мови специфікації ПЗс [11]. Виходить на передові позиції об'єктний підхід до розробки ПЗс [12]. Створюються різні інструментальні середовища розробки і супроводу ПЗс. Бурхливо розвиваються концепції

комп'ютерних мереж.

Об'єктно-орієнтоване програмування визначається, як технологія створення складного програмного забезпечення, яка базується на уявленні програми у вигляді сукупності об'єктів, кожен з яких є екземпляром певного типу (класу), а класи утворюють ієрархію з спадкуванням властивостей. Взаємодія програмних об'єктів в такій системі здійснюється шляхом передачі повідомлень

У 90-ті роки міжнародна комп'ютерна мережа широко охопила все людське суспільство, персональні комп'ютери стали підключатися до неї як термінали. Гостро постала проблема захисту комп'ютерної інформації та переданих по мережі повідомлень. Стали бурхливо розвиватися комп'ютерна технологія розробки ПС і пов'язані з нею формальні методи специфікації програм. Можна сказати, що в цей період починається вирішальний етап повної інформатизації і комп'ютеризації суспільства.

Основною перевагою об'єктно-орієнтованого програмування в порівнянні з модульним програмуванням є «більш природна» декомпозиція програмного забезпечення, яка істотно полегшує його розробку. Це призводить до більш повної локалізації даних та інтегруванню їх з підпрограмами обробки, що дозволяє вести практично незалежну розробку окремих частин (об'єктів) програми. Крім цього, об'єктний підхід пропонує нові способи організації програм, засновані на механізмах успадкування, поліморфізму, композиції, наповнення. Ці механізми дозволяють конструювати складні об'єкти з порівняно простих.

Бурхливий розвиток технологій програмування, заснованих на об'єктному підході, дозволило вирішити багато проблем. Так були створені середовища, підтримують візуальне програмування, наприклад, Delphi, C ++ Builder, Visual C ++ і т. д. При використанні візуального середовища у програміста з'являється можливість проектувати деяку частину, наприклад, інтерфейси майбутнього продукту, із застосуванням візуальних засобів додавання і налаштування спеціальних бібліотечних компонентів.

Таким чином, при використанні цих мов програмування зберігається залежність модулів програмного забезпечення від адрес експортованих полів і методів, а також структур і форматів даних. Ця залежність об'єктивна, так як модулі повинні взаємодіяти між собою, звертаючись до ресурсів один одного. Зв'язки модулів можна розірвати, але можна спробувати стандартизувати їх взаємодія, на чому і заснований компонентний підхід до програмування.

Компонентний підхід і CASE-технології (з середини 90-х років XX ст. до нашого часу).

Компонентний підхід пропонує побудову програмного забезпечення з окремих компонентів фізично окремо існуючих частин програмного забезпечення, які взаємодіють між собою через стандартизовані двійкові інтерфейси. На відміну від звичайних об’єктів об’єкти-компоненти можна збирати в динамічні бібліотеки або файли які виконуються, розповсюджувати в двійковому коді (без початкових текстів) і використовувати в будь-якій мові програмування, що підтримує відповідну технологію. На сьогоднішній день ринок об’єктів став реальністю, так, в Інтернеті існують вузли, які представляють велику кількість компонентів, реклами компонентів в журналах. Це дозволяє програмістам створювати продукти, які хоча б частково складаються з повторно використаних частин, тобто використати

технологію, яка добре зарекомендувала себе в області проектування апаратури (рис. 2.5).

Компонентний підхід лежить в основі технологій, розроблених на базі COM (Component Object Model – компонентна модель об’єктів), і технології створення розподілених додатків CORBA (Common Object Request Broker Architecture – загальна архітектура з посередником обробки запитів об’єктів). Ці технології використовують схожі принципи і розрізняються лише особливостями їх реалізації.

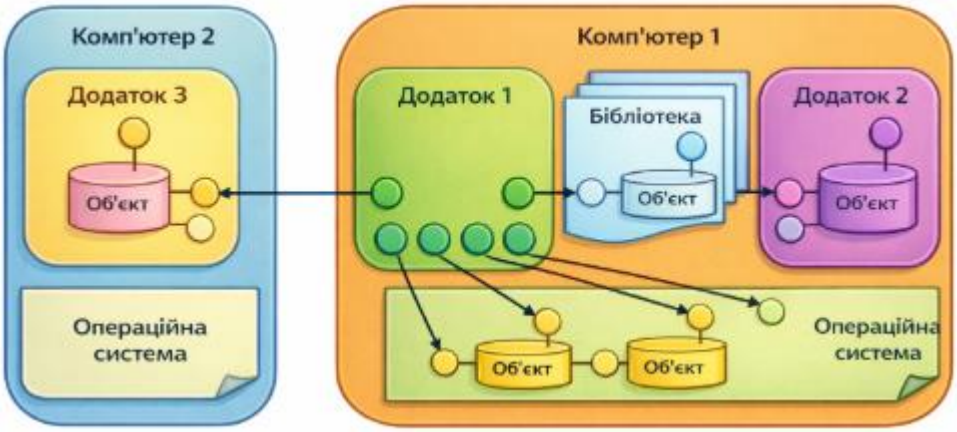


Рисунок 2.5 - Взаємодія програмних компонентів різних типів

Об’єкт завжди функціонує у складі сервера – динамічної бібліотеки або виконуваного файлу, які забезпечують функціонування об’єкта. Розрізняють три типи серверів:

1) внутрішній сервер; реалізується динамічними бібліотеками, які підключаються до додатка-клієнта і працюють в спільному адресному просторі, найбільш ефективний сервер, крім того він не вимагає спеціальних ресурсів;

2) локальний сервер; створюється окремим процесом (модулем, *.exe), який працює на одному комп’ютері з клієнтом;

3) віддалений сервер; створюється процесом, який працює на іншому комп’ютері.

Наприклад, Microsoft Word є локальним сервером. Він включає багато об’єктів, які можуть використовуватись іншими додатками.

Для звернення до служб клієнт повинен отримати вказівник на відповідний інтерфейс. Перед першим зверненням до об’єкта клієнт посилає запит до бібліотеки COM, що містить інформацію про всі зареєстровані в системі класи COM об’єктів, і передає їй ім’я класу, ідентифікатор інтерфейсу і тип сервера. Бібліотека запускає необхідний сервер, створює необхідні об’єкти і повертає вказівники на об’єкти та інтерфейси. Отримавши вказівники, клієнт може викликати необхідні функції об’єкта.

Особливістю сучасного етапу розвитку технології програмування, крім зміни підходу, є створення та супроводження програмного забезпечення, які були названі CASE-технології (Computer-Aided Software/System Engineering – розробка програмного забезпечення програмних систем з використанням комп’ютерної підтримки). Без засобів автоматизації розробка достатньо складного програмного забезпечення на сучасний момент стає тяжко реалізованою задачею: пам’ять

людини вже не в стані фіксувати всі деталі, які необхідно враховувати при розробці програмного забезпечення. На сьогодні існують CASE-технології, які підтримують як структурний, так і об'єктний (у тому числі і компонентний) підходи до програмування.

Поява нового підходу не означає, що все програмне забезпечення буде створюватись з програмних компонентів, але аналіз існуючих проблем розробки складного програмного забезпечення показує, що він буде використовуватись досить широко.

Моделі життєвого циклу програмного забезпечення

Поняття життєвого циклу програмного забезпечення з'явилося, коли програмістське співтовариство усвідомило необхідність переходу від кустарних ремісничих методів розробки програм до технологічного промислового їх виробництва. Як зазвичай відбувається в подібних ситуаціях, програмісти спробували перенести досвід інших індустріальних виробництв в свою сферу. Зокрема, було запозичене поняття життєвого циклу.

Життєвий цикл програмного забезпечення - період часу, що починається з моменту прийняття рішення про необхідність створення програмного продукту і закінчується в момент його повного вилучення з експлуатації [13]. Цей цикл - процес побудови і розвитку ПЗ.

Поняття ЖЦ виникло під впливом потреби у систематизації робіт у процесі розроблення ПЗ. Систематизація була першим етапом на шляху до автоматизації процесу розроблення ПЗ. Наступними кроками переходу до автоматизації процесу розроблення ПЗ були такі: встановлення технологічних маршрутів діяльності розробників ПЗ, визначення можливості їх автоматизації та виявлення ризиків, розроблення інструментів для автоматизації.

Використання поняття життєвого циклу дозволяє обрати підходи, які найбільш ефективні для завдань певного етапу життя ПЗ. Залежно від особливостей процесів розроблення та супроводу програм існують різні моделі ЖЦ.

Використання певної моделі ЖЦ дозволяє визначитися з основними моментами процесу замовлення, розроблення та супроводу ПЗ навіть недосвідченому програмісту. Також використання моделей дозволяє чітко зрозуміти, в який період переходити від версії до версії, які дії з удосконалення виконувати, на якому етапі. Знання про закономірності розвитку програмного продукту, які відбиваються в обраній моделі ЖЦ, дозволяють отримати надійні орієнтири для планування процесу розроблення та супроводу ПЗ, економно витрачати ресурси та підвищувати якість управління усіма процесами.

Також моделі життєвого циклу є основою знань технологій програмування та інструментарію, що їх підтримує. Будь-що, технологія базується на певних уявленнях про життєвий цикл та організує свої методи та інструменти навколо фаз та етапів ЖЦ.

Розвиток методалогій програмування у 70-х рр. XX ст. привів до формування потреби вивчення життєвого циклу ПЗ. До цього часу моделі ЖЦ розвиваються і модифікуються, уточнюючи та доповнюючи дві базові моделі – каскадну та ітеративну. Ці зміни обумовлені потребою організаційної та технологічної підтримки проектів з розроблення ПЗ.

Найбільшого поширення набули дві моделі: каскадна (водоспадна), створена в 1970/85 рр., і спіральна, створена в 1986/1990 рр.

Каскадна модель (waterflow model).

Каскадна модель життєвого циклу (модель водоспаду - Waterfall model) була запропонована в 1970 р. В. Ройсом.

Каскадна модель ЖЦ ПЗ виникла для задоволення потреби у систематизації робіт ще на ранніх етапах розроблення програм. Згідно з цією моделлю програмні системи проходять в своєму розвитку дві фази:

1. розроблення;
2. супровід.

Фази розбиваються на ряд етапів, представлених на рис. 2.6.

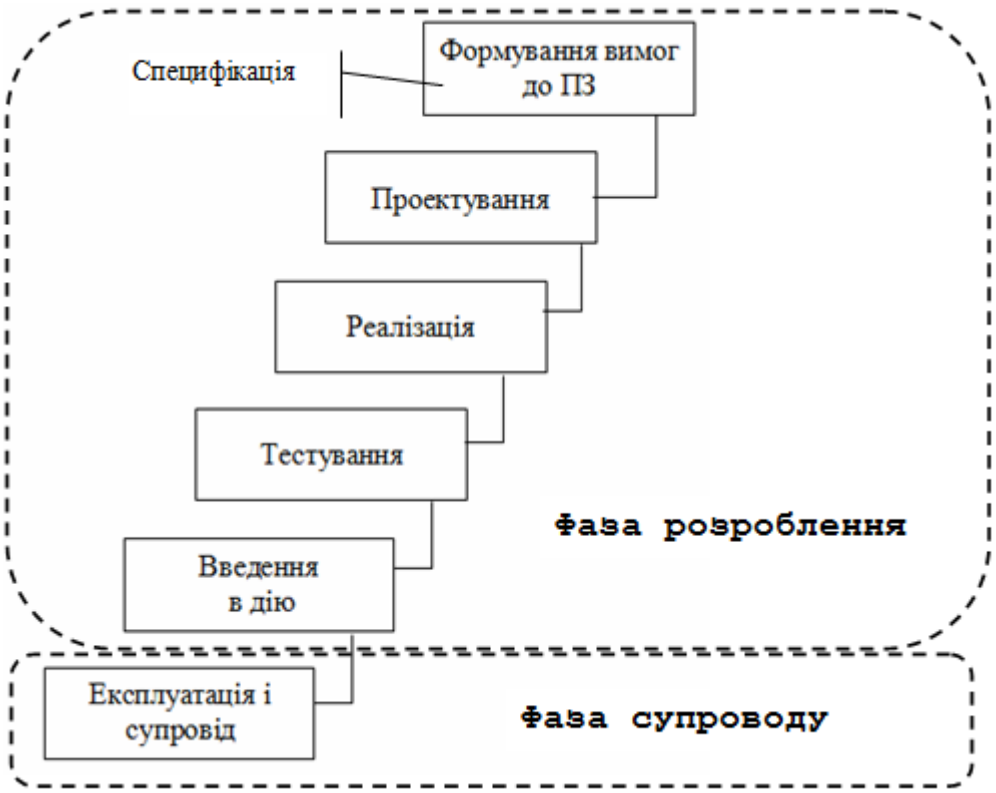


Рисунок 2.6 - Каскадна модель ЖЦ

Каскадна модель передбачає послідовне виконання всіх етапів проекту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі.

Розробка починається з ідентифікації потреби в новому додатку, а закінчується передачею продукту розробки в експлуатацію. Усі етапи розробки програмного забезпечення регламентуються стандартами підприємства та державним стандартом ГОСТ 34.601-90.

Першим етапом фази розробки є специфікація (Requirements Specification) – постановка завдання і визначення вимог. На етапі постановки завдання замовник спільно з розробниками приймають рішення про створення системи. Визначення вимог включає опис загального контексту задачі, очікуваних функцій системи та її обмежень. Особливо важливим є цей етап для нетрадиційних додатків. У разі

позитивного рішення починається аналіз системи відповідно до вимог. Розробники програмного забезпечення намагаються осмислити висунуті замовником вимоги і зафіксувати їх у вигляді специфікацій системи. Призначення специфікацій – описувати зовнішню поведінку системи, а не її внутрішню організацію.

Перш ніж розпочати створювати проект за специфікаціями, вимоги повинні бути ретельно перевірені на відповідність вихідним цілям, повноту, сумісність (несуперечність) та однозначність. Завдання етапу аналізу полягає в тому, щоб вибудувати опис програми у вигляді логічної системи, зрозумілої як для замовника, майбутніх користувачів, так і для виконавців проекту. На етапі специфікації обов'язково формується технічне завдання на розроблення ПЗ [15]. Завдання етапу аналізу полягає в тому, щоб вибудувати опис програми у вигляді логічної системи, зрозумілої як для замовника, майбутніх користувачів, так і для виконавців проекту. На етапі специфікації обов'язково формується технічне завдання на розробку ПЗ.

Отже, на етапі Специфікація виконується *постановка завдання,* коли замовник спільно з розробниками приймають рішення про створення системи, *визначення вимог*, а саме опис загального контексту задачі, очікуваних функцій системи та її обмежень. Розробники ПЗ намагаються осмислити вимоги, висунуті замовником, і зафіксувати їх у вигляді специфікацій системи.

Призначення Специфікації - описати зовнішню поведінку системи.

Розробка проектних рішень, що відповідають на питання, як повинна бути реалізована система, щоб вона могла задовільняти певні вимоги, виконується на етапі проектування. Оскільки складність системи в цілому може бути дуже великою, головним завданням цього етапу є послідовна декомпозиція системи до рівня очевидно реалізованих модулів або процедур. Результати виконання цього етапу оформляються як технічний проект.

Розроблення проектних рішень, що відповідають на питання, як повинна бути реалізована система, щоб вона могла задовольняти визначені вимоги, виконується на етапі проектування (Design).

Головне завдання Розроблення проектних рішень - послідовна декомпозиція системи до рівня очевидно реалізованих модулів або процедур

Результат етапу - технічний проект, вимоги до документів якого встановлені стандартом. Фаза розробки закінчується тестуванням - автономним і комплексним та передачею системи в експлуатацію.

Етапи *Експлуатація* та *Супровід* включають в себе діяльність щодо:

- забезпечення нормального функціонуванняпрограмних систем;
- фіксування помилок, пошук їх причин та виправлення;
- підвищення експлуатаційних характеристик системи;
- адаптацію системи до довкілля.

Принципова особливість каскадної моделі - перехід на наступну стадію здійснюється тільки після повного завершення роботи на поточній стадії, повернення на пройдені стадії не передбачається. Кожна стадія закінчується отриманням результатів, які є вхідними даними для наступної стадії, і випуском повного комплекту документації. Вимоги до програмного забезпечення, певні на стадії формування вимог, документуються у вигляді технічного завдання і фіксуються на всій розробці. Критерієм якості розробки при такій моделі є точність

виконання специфікацій технічного завдання.

Каскадний підхід добре зарекомендував себе при побудові відносно простих ІС, коли на самому початку розробки можна досить точно і повно сформулювати всі вимоги до системи. Основним недоліком цього підходу є те, що реальний процес створення системи ніколи повністю не вкладається в таку жорстку схему, постійно виникає потреба в поверненні до попередніх етапах і уточнення або перегляд раніше прийнятих рішень.

На наступному етапі Реалізації (Construction), або кодування, кожен з цих модулів програмується на найбільш підходящій для даного застосування мові. З точки зору автоматизації цей етап традиційно є найбільш розвиненим.

У каскадній моделі фаза розроблення закінчується етапом тестування (Testing and debugging), автономного і комплексного, та передачею системи в експлуатацію (Installation).

Фаза експлуатації та супроводу включає в себе всю діяльність щодо забезпечення нормального функціонування програмних систем, у тому числі фіксування розкритих під час виконання програм помилок, пошук їх причин та виправлення, підвищення експлуатаційних характеристик системи, адаптацію системи до довкілля, а також за необхідності і більш суттєві роботи з удосконалення системи. Фактично відбувається еволюція системи. У ряді випадків на дану фазу припадає більша частина коштів, що витрачаються в процесі життєвого циклу програмного забезпечення.

Зрозуміло, що увага програмістів до тих чи інших етапів розроблення залежить від конкретного проекту. Часто розробнику немає необхідності проходити через усі етапи, наприклад, якщо створюється невелика, добре зрозуміла програма із чітко поставленою метою.

Стисла характеристика:

- фіксований набір стадій;
- кожна стадія закінчується документованим результатом;
- наступна стадія починається лише після закінчення попередньої.

Цінність цієї моделі полягає в тому, що вона фіксує послідовність етапів розробок і можливість повернення до попередніх етапів роботи.

Основна увага розробників зосереджено на досягненні кращих значень технічних характеристик ПЗ, а саме: продуктивність, обсягу пам'яті і т. п. Переваги застосування каскадної моделі: на кожній стадії формується закінчений набір проектної документації, який відповідає критеріям повноти і узгодженості; виконання робіт в логічній послідовності дозволяє планувати терміни завершення всіх робіт і відповідні витрати.

Ця модель добре зарекомендувала себе, коли на самому початку розробки можна досить точно і повно сформулювати всі вимоги. Під цю категорію потрапляють складні системи з великою кількістю завдань обчислювального характеру, системи реального часу і т. п.

Переваги цієї моделі в найбільшій мірі відповідають командній системі керування господарською діяльністю. Але реальний процес життєвого циклу не вміщується у таку жорстко регламентовану модель. На всіх етапах життєвого циклу неодноразово виникає необхідність повернення до попередніх етапів не тільки через наявність помилок, й через причину впливу факторів часу та інших

факторів. Постійно виникає необхідність уточнити прийняті раніше рішення.

Недоліками каскадної моделі є:

- негнучкість;
- фаза повинна бути завершена до переходу до наступної;
- набір фаз фіксований;
- важко реагувати на зміни вимог.

Недоліком каскадної системи є також те, що система не розвивається, вона зупиняється на тому рівні, який був закладений на початкових етапах аналізу і проектування.

Використання цієї моделі - там, де вимоги добре зрозумілі та стабільні.

Каскадна модель життєвого циклу є ідеальною, оскільки лише дуже прості завдання проходять всі етапи без будь-яких ітерацій (повернень на попередні кроки процесу). Наприклад, при програмуванні може виявитися, що реалізація деякої функції неефективна і вступає в протиріччя з вимогами до продуктивності системи. У цьому випадку потрібні зміни проекту, а можливо, і переробка специфікацій. Для обліку повторюваності етапів процесу розробки створювалися альтернативи каскадної моделі. З таких альтернатив утворилася ітеративна модель. Ця модель передбачає розбиття життєвого циклу проекту на послідовність ітерацій, кожна з яких нагадує "міні-проект" з усіма фазами життєвого циклу.

Ітеративна модель (Iterative and incremental development)

Каскадна модель життєвого циклу є ідеальною, оскільки лише дуже прості завдання проходять усі етапи без будь-яких ітерацій (повернень на попередні кроки процесу). Наприклад, при програмуванні може виявитися, що реалізація деякої функції неефективна і вступає у протиріччя з вимогами до продуктивності системи. У цьому випадку потрібні зміни проекту, а можливо, і переробка специфікацій. Для врахування повторюваності етапів процесу розроблення створювались альтернативи каскадної моделі.

Із таких альтернатив утворилася ітеративна модель. Ця модель передбачає розбиття життєвого циклу проекту на послідовність ітерацій, кожна з яких нагадує "міні-проект" з усіма фазами життєвого циклу.

Класична ітераційна модель абсолютизує можливість повернень на попередні етапи (рис. 2.7). Ця обставина відбиває істотний аспект програмних розробок: прагнення заздалегідь передбачати всі ситуації використання системи та неможливість у переважній більшості випадків досягти цього. Усі традиційні технології програмування спрямовані лише на те, щоб мінімізувати повернення. Але суть від цього не змінюється: при поверненні завжди доводиться повторювати побудову того, що вже вважалося готовим.

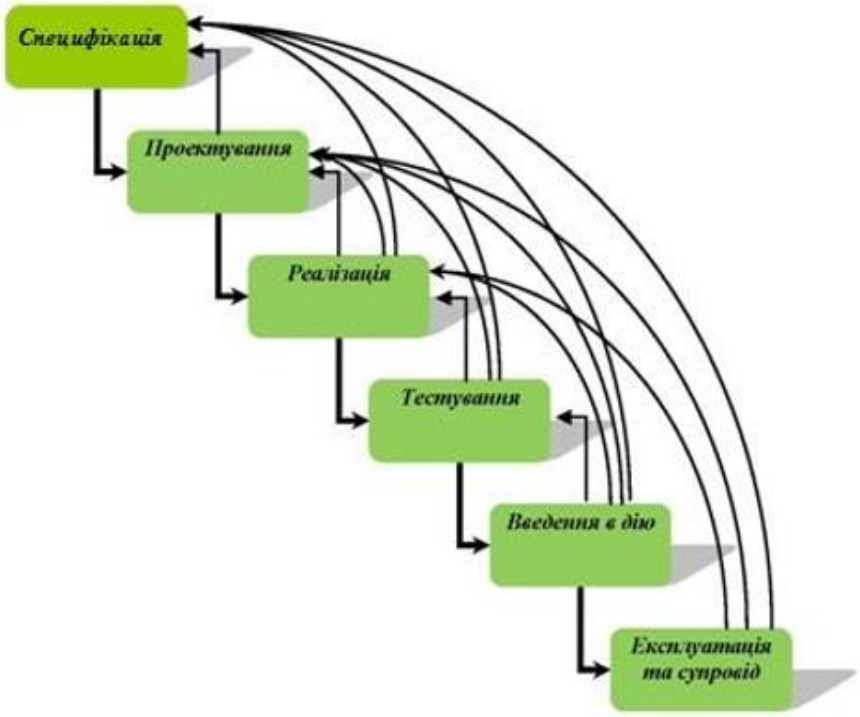


Рисунок 2.7. Класична ітераційна модель

Класична ітераційна модель абсолютизує можливість повернень на попередні етапи. Ця обставина відображає істотний аспект програмних розробок: прагнення заздалегідь передбачити всі ситуації використання системи і неможливість в більшості випадків досягти цього. Всі традиційні технології програмування спрямовані лише на те, щоб мінімізувати повернення. Але суть від цього не змінюється: при поверненні завжди доводиться повторювати побудову того, що вже вважалося готовим.

Мета кожної ітерації в розробці ПЗ - отримання працюючої версії програмної системи, що включає функціональність, певну інтегрованим змістом усіх попередніх і поточної ітерації. Результат фінальної ітерації містить всю необхідну функціональність продукту. Таким чином, із завершенням кожної ітерації розвивається інкрементальний продукт (нарощується функціональність).

Прагнення розробника ПЗ на цьому етапі - заздалегідь передбачати всі ситуації використання системи, але в дійсності неможливо цього досягти. При ітерації доводиться повторювати побудову того, що вважалося готовим.

Мета ітерації - отримати працюючу версію програмної системи, що включає функціональність усіх попередніх і поточної ітерації.

Результат фінальної ітерації - функціональність продукту.

Основна характеристика - неодноразове повторення стадій. Переваги ітераційної моделі:

- на кожному кроці маємо працюючу систему: можна на ранній стадії проекту почати тестування користувачами;
- можна прийняти стратегію розробки відповідно до бюджету,
- повністю захищає від перевитрат часу або коштів (зокрема, за рахунок скорочення другорядної функціональності).

Недоліки:

- система часто погано структурована, проект «не прозорий»;
- після уточнення вимог відкидається частина раніше виконаної роботи;
- потрібні засоби для швидкого розроблення.

Використання: ітераційна модель підходить для малих та середніх проектів.

Спіральна модель (Spiral model).

Природним системам більш характерна спіральна модель життєвого циклу. Вони проходять послідовно етапи зародження, розвитку, відмирання, ці етапи багаторазово повторюються. Розвиток у природі здійснюється по спіралі. Спіральна модель життєвого циклу прийнята у вигляді стандарту і для розробки інформаційних та інших технічних систем [17].

Спіральна модель ЖЦ була запропонована для подолання проблем каскадної та ітераційної моделі.

Спіральна модель була розроблена в середині 1980-х років Барі Боем. Вона ґрунтується на класичному циклі Демінга PDCA (*plan-do-check-act* планувати-ні-перевірити-дія). Відмінною особливістю цієї моделі є спеціальна увага ризикам, що впливає на організацію життєвого циклу.

У спіральній моделі розроблення програми має вигляд серії послідовних ітерацій. На перших етапах уточнюються специфікації продукту, на наступних - додаються нові можливості і функції. Мета цієї моделі - по закінченні кожної ітерації заново здійснити оцінку ризиків продовження робіт.

При використанні цієї моделі система створюється в кілька ітерацій (витків спіралі) методом прототипування.

Прототип - це програмний компонент, який реалізує окремі функції і зовнішні інтерфейси ПЗ [18].

Існує два базових варіанти використання прототипів. У першому варіанті створення прототипу використовується для кращої специфікації вимог до ПЗ, після розробки яких сам прототип стає непотрібним.

Другий варіант передбачає ітераційний розвиток прототипу в готовий для експлуатації програмний продукт. Ітерації розробки прототипу включають створення/модифікацію прототипу, його демонстрацію користувачеві, узгодження, розробку нових вимог до ПЗ, нову модифікацію, поки не буде створено готовий додаток.

Створення прототипів здійснюється декількома ітераціями. Кожна ітерація відповідає створенню фрагмента або версії ПЗ, уточнюються цілі і характеристики проекту, оцінюється якість отриманих результатів та плануються роботи наступної ітерації. На кожній ітерації проводиться ретельна оцінка ризику перевищення термінів і вартості проекту, щоб визначити необхідність проведення ще однієї ітерації, ступінь повноти і точності розуміння вимог до системи, а також доцільності припинення проекту. Спіральна модель позбавляє користувачів і розробників ПЗ від необхідності повного і точного формулювання вимог до системи на початковій стадії, оскільки вони уточнюються на кожній ітерації. Т.ч. уточнюються і послідовно конкретизуються деталі проекту і в кінці кінців вибирається обґрунтований варіант, який і реалізується.

Модель має ітераційний характер і рухається по спіралі, проходячи стадії, де

на кожному витку уточнюються характеристики майбутнього інформаційного продукту.

На етапах аналізу і проектування реалізація технічних рішень і ступінь задоволення потреб замовника перевіряється шляхом створення прототипів. Кожен виток спіралі відповідає створенню працездатного фрагмента або версії системи. Це дозволяє уточнити вимоги, цілі і характеристики проекту, визначити якість розробки, спланувати роботи наступного витка спіралі. Таким чином поглиблюються і послідовно конкретизуються деталі проекту і в результаті вибирається обґрунтований варіант, який задовольняє дійсним вимогам замовника та доводиться до реалізації.

Спіральна модель життєвого циклу також має ряд послідовних етапів, а саме:

- визначення вимог;
- аналіз;
- проектування;
- реалізація та тестування;
- інтеграція.

На цих етапах виконуються операції, як і при каскадній моделі. Етап реалізації здійснюється шляхом створення прототипів, на яких конкретизуються вимоги й деталі наступного циклу. Вказана модель зображена на рис. 2.8.

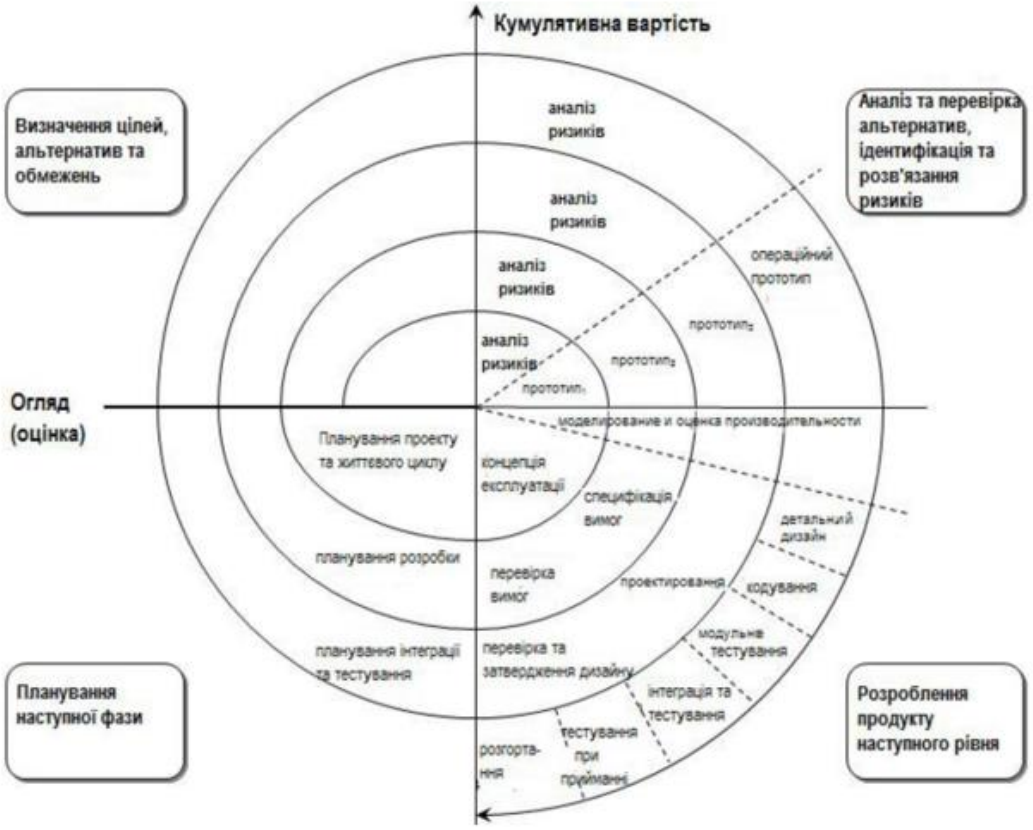


Рисунок 2.8 - Спіральна модель життєвого циклу

Ітераційний процес розробки відображає об'єктивно спіральний цикл

створення системи. Неповне завершення робіт на кожній стадії дозволяє переходити на наступну стадію, не чекаючи повного завершення роботи на поточному. При ітеративному способі розробки відсутню стадію можна буде виконати на наступній ітерації. Головне завдання - якомога швидше показати користувачам системи працездатний продукт, активізуючи процес уточнення і доповнення вимог.

Кожна ітерація відповідає створенню фрагмента або версії ПЗ, на ній уточнюються цілі і характеристики проекту, оцінюється якість отриманих результатів та плануються роботи наступної ітерації.

На кожній ітерації оцінюються:

- ризик перевищення термінів і вартості проекту;
- необхідність виконання ще однієї ітерації;
- ступінь повноти і точності розуміння вимог до системи;
- доцільність припинення проекту.

Спіральна модель не виключає використання каскадного підходу на кінцевих стадіях проекту в тих випадках, коли вимоги до системи стають цілком чіткими.

У спіральної моделі розробки програми має вигляд серії послідовних ітерацій. На перших етапах уточнюються специфікації продукту, на наступних - додаються нові можливості і функції. Мета цієї моделі - після закінчення кожної ітерації заново здійснити оцінку ризиків продовження робіт.

На кожному витку спіралі виконується створення чергової версії продукту, уточнюються вимоги проекту, визначається його якість і плануються роботи наступного витка. Особлива увага приділяється початкових етапах розробки - аналізу і проектування, де реалізація тих чи інших технічних рішень перевіряється і обґрунтовується за допомогою створення прототипів (макетування).

Основна проблема спірального циклу - визначення моменту переходу на наступну стадію. Для її вирішення необхідно ввести тимчасові обмеження на кожну зі стадій життєвого циклу.

Перехід здійснюється відповідно до плану, навіть якщо не вся запланована робота закінчена. План складається на основі статистичних даних, отриманих в попередніх проектах, і особистого досвіду розробників.

Відмінною особливістю спіральної моделі є спеціальна увага, що приділяється ризикам, що впливає на організацію життєвого циклу, і контрольними точками. Барі Боем формулює 10 найбільш поширених (за пріоритетами) ризиків:

1. Дефіцит фахівців.
2. Нереалістичні терміни і бюджет.
3. Реалізація невідповідної функціональності.
4. Розробка неправильного користувальницького інтерфейсу.
5. Перфекціонізм, непотрібна оптимізація і відточування деталей.
6. Безперервний потік змін.
7. Брак інформації про зовнішні компонентах, що визначають оточення системи або залучених в інтеграцію.
8. Недоліки в роботах, виконуваних зовнішніми (по відношенню до проекту) ресурсами.
9. Недостатня продуктивність одержуваної системи.

10. Разрив в кваліфікації фахівців різних галузей.

У сьогоднішній спіральної моделі визначений наступний загальний набір контрольних точок:

- Concept of Operations (COO) - концепція (використання) системи;
- Life Cycle Objectives (LCO) - цілі і зміст життєвого циклу;
- Life Cycle Architecture (LCA) - архітектура життєвого циклу; тут же можливо говорити про готовність концептуальної архітектури цільової програмної системи;
- Initial Operational Capability (IOC) - перша версія створюваного продукту, придатна для дослідної експлуатації;
- Final Operational Capability (FOC) - готовий продукт, розгорнутий (встановлений і налаштований) для реальної експлуатації.

Команда розробників повинна бути групою професіоналів, що мають досвід у проектуванні, програмування та тестуванні ПЗ, здатних взаємодіяти з кінцевим користувачем і трансформувати їх пропозиції в робочі прототипи.

Спіральна модель життєвого циклу має достатньо переваг:

- наявність дій з аналізу ризиків, що забезпечує їх скорочення і завчасне визначення непереборних ризиків;
- розбиття потенційного обсягу робіт на невеликі частини;
- першочерговість реалізації вирішальних функцій з високим ступенем ризику, при необхідності зупинити роботи над проектом на ранніх циклах моделі і зменшити витрати;
- можливість користувачам приймати участь при плануванні, аналізі ризиків, проектуванні, розробці, виконанні оцінних дій;
- удосконалення адміністративного управління процесами життєвого циклу розробки, витратами, дотриманням графіку та кадровим забезпеченням, що досягається шляхом виконання аналізу (огляду) в кінці кожної ітерації;
- підвищення продуктивності за рахунок використання придатних для повторного використання результатів;
- підвищення ймовірності передбачуваної поведінки системи за допомогою уточнення поставлених цілей;
- відсутність необхідності в попередньому розподілі всіх фінансових ресурсів проекту;
- регулярна оцінка загальних витрат і, в результаті, їх скорочення.

При використанні спіральної моделі стосовно невідповідному їй проекту, виявляються такі її недоліки:

- висока вартість моделі за рахунок вартості та додаткових тимчасових витрат на планування, визначення цілей, виконання аналізу ризиків та прототипування при проходженні кожного циклу спіралі;
- невиправдано висока вартість моделі для проектів, що мають низький ступінь ризику або невеликі розміри;
- ускладненість структури моделі, що призводить до складності її використання розробниками, менеджерами і замовниками;

- необхідність у високопрофесійних знаннях для оцінки ризиків;
- можливість віддалення закінчення роботи над проектом у зв'язку з бажанням замовника покращувати кожну створену версію;
- необхідність в обробці додаткової документації за рахунок великої кількості проміжних циклів;
- необхідність у чіткому розподілі робіт між розробниками;
- складність визначення критеріїв для продовження процесу розробки на наступній ітерації;
- необхідність потужних інструментальних засобів і методів прототипування.

Очевидно, що класична спіральна модель є досить складною. З урахуванням цього розроблено ряд її спрощених версій.

Це забезпечує більш високу якість процесу розробки продукту, і спрощує прогнозування термінів і вартості розробки, підвищує задоволеність замовника результатами продукту.

До переваг описаних спрощених спіральних моделей можна віднести:

- Спрощено в порівнянні з базовою моделлю Боема, що робить її більш зрозумілою як розробнику так і замовнику.
- На відміну від інших моделей, увага приділяється діям безпосередньо не пов'язаних з розробкою.

Це забезпечує більш високу якість процесу розробки продукту, і спрощує прогнозування термінів і вартості розробки, підвищує задоволеність замовника результатами продукту.

Використання - для малих і середніх проектів.

Тема: Основні етапи розробки програмного забезпечення.

Мета: ознайомитись з етапами розробки програмного забезпечення. Вивчити питання, які вирішуються на кожному з етапів розробки програмного забезпечення.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.
II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.
III. Виклад нового матеріалу:

План:

1. Етапи розробки програмного забезпечення.
2. Питання, які вирішуються на кожному з етапів розробки програмного забезпечення.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
1. Ознайомитись з теоретичним матеріалом теми.
2. Вивчити основні поняття лекції.
3. Відповісти на контрольні питання.

Контрольні питання:
4. Які виділяють етапи розробки програмного забезпечення?
5. Які виділяють стадії створення програмного забезпечення?
6. Які існують моделі життєвого циклу програмного забезпечення

Створенням програмного забезпечення називають розробку, дизайн та поточне обслуговування програмного забезпечення.

При цьому застосовують різноманітні технології з таких сфер, як інформатика, проектування, управління цифровими даними і дизайн у всіх його проявах. Розробка будь-якої програми, будь-то невелика процедура по обробці надходить на консоль інформації або комплексний програмний продукт, складається з декількох етапів, грамотна реалізація яких є обов'язковою умовою для отримання хорошого результату.

Чітке дотримання вивіреним часом етапах розробки програмного забезпечення стає основним критерієм для компаній, що займаються створенням ПЗ і їх замовників, що зацікавлені в отриманні програми, що чудово виконує свої функції.

Детально розглянемо кожну стадію загальновизнану методологію розробки ПЗ, щоб оцінити їх високу значимість для досягнення поставленої перед виконавцями мети.

Залежно від особливостей проекту порядок розробки програмного забезпечення може відрізнятися, але в загальному вигляді він такий (рис. 1.1):

Рисунок 1.1 - Загальний порядок розробки програмного забезпечення

У процесі створення будь-якої програми можна виділити кілька етапів.

1. **Постановка завдання** - виконується фахівцем в предметної області на природній мові (українській, англійській тощо). Необхідно визначити мету завдання, її зміст і загальний підхід до вирішення. Можливо, що завдання вирішується точно (аналітично), і без комп'ютера можна обійтися. Уже на етапі постановки треба враховувати ефективність алгоритму розв'язання задачі на комп'ютері, обмеження, що накладаються апаратним і програмним забезпеченням (АТ і ПЗ).

2. **Аналіз завдання та моделювання** - визначаються вихідні дані і результат, виявляються обмеження на їх значення, виконується формалізоване опис завдання і побудова (вибір) математичної моделі, придатної для вирішення на комп'ютері.

3. **Розробка або вибір алгоритму розв'язання задачі** - виконується на основі її математичного опису. Багато задач можна вирішити різними способами. Програміст повинен вибрати оптимальне рішення. Неточності в постановці, аналізі завдання або розробці алгоритму можуть привести до прихованої помилки - програміст отримає невірний результат, вважаючи його правильним.

4. **Проектування загальної структури програми** - формується модель рішення з подальшою деталізацією і розбивкою на підпрограми, визначається "архітектура" програми, спосіб зберігання інформації (набір змінних, масивів і т. п.).

5. **Кодування** - запис алгоритму на мові програмування. Сучасні системи програмування дозволяють прискорити процес розробки програми, автоматично створюючи частину її тексту, однак творча робота, як і раніше лежить на програмістові. Для успішної реалізації цілей проекту програмісту необхідно використовувати методи структурного програмування.

6. **Налагодження і тестування програми**. Під налагодженням розуміється

усунення помилок в програмі. Тестування дозволяє вести їх пошук і, в кінцевому рахунку, переконатися в тому, що повністю налагоджена програма дає правильний результат. Для цього розробляється система тестів - спеціально підібраних контрольних прикладів з такими наборами параметрів, для яких рішення задачі відомо. Тестування повинне охоплювати всі можливі розгалуження в програмі, тобто перевіряти всі її інструкції, і включати такі вихідні дані, для яких рішення неможливо. Перевірка особливих, виняткових ситуацій, необхідна для аналізу коректності. Наприклад, програма повинна відмовити клієнту банку в проханні видати суму, відсутню на його рахунку. У відповідальних проектах велика увага приділяється так званої "захисту від дурня" припускає стійкість програми до невмілому зверненням користувача. Використання спеціальних програм-відладчиків, які дозволяють виконувати програму по окремих кроках, переглядаючи при цьому значення змінних, значно спрощує цей етап.

7. **Аналіз результатів** - якщо програма виконує моделювання будь- якого відомого процесу, слід зіставити результати обчислень з результатами спостережень. У разі істотного розбіжності необхідно змінити модель.

8. **Публікація результатів роботи**, передача замовнику для експлуатації.

9. **Супровід програми** - включає консультації представників замовника по роботі з програмою і навчання персоналу. Недоліки і помилки, помічені в процесі експлуатації, повинні усуватися.

Розглянемо ці етапи більш детально.

1. Постановка завдання

При підготовці до проектування вирішуються організаційні питання:

- що клієнт може надати (ТЗ, макети, дизайн), наскільки достатні вихідні та які етапи закривають - таким чином визначається склад робіт;
- бюджет і терміни: на основі наявних матеріалів затверджується приблизна вартість, термін всього проекту, а також термін і точна вартість найближчого етапу.

Тільки тоді можна підписувати контракт, отримувати передоплату і всі необхідні для роботи матеріали.

Під час перших зустрічей щодо замовлення на розроблення ПЗ замовник дуже приблизно уявляє, що йому потрібно. Як правило, він надає кілька сторінок тексту завдання і одразу просить оцінити час виконання замовлення та його вартість. Без чіткого визначення процесів, для автоматизації яких буде використовуватись ПЗ, неможливо навіть приблизно оцінити обсяг робіт. Приблизна оцінка з мінімумом вхідної інформації може призвести до помилки в кілька разів, що негативно відіб’ється на точності визначення строків виконання та вартості робіт.

Потреби (needs) – відображають проблеми бізнесу, персоналій або процесу, що повинні співвідноситися з використанням або придбанням системи.

Щоб тримати уявлення про можливі обсяги робіт, потрібно пропонувати замовнику надати або розробити технічне завдання. Завдяки цьому системні аналітики зможуть розібратися в задачі, за допомогою інструментальних засобів виконати декомпозицію системи на компоненти, приблизно визначити обсяги цих компонентів і відповідно час їх реалізації. Ця початкова стадія ЖЦ ПЗ є "оцінкою здійсненності".

Постановка завдання – найбільш творча частина ЖЦ ПЗ. Потрібно описати поведінку розроблюваної системи. Ця система отримує якісь сигнали з її оточення, тому треба описати поведінку оточення, але оточення само залежить і змінюється під впливом системи, її сигналів, особливо аварійних.

Вирішують це протиріччя ітераційно, поетапно уточнюючи поведінку як системи, так і її оточення. Для відповідальних систем замовник може запропонувати розробити імітаційну модель системи та оточення, що є досить складною.

У поставленому завданні замовник визначає вимоги до створюваної системи, які повинні задовольняти потреби користувачів і бути зрозумілими для розробників.

Вимога (Requirement) – умова або можливість, що визначена користувачем для вирішення проблеми або досягнення мети, та якій повинна відповідати або якою повинна володіти система чи її компонент, щоб задовольняти умови контракту, стандарту, специфікації або іншого формально репрезентованого документа.

Вимоги поділяються на:

- вимоги користувача (User Requirements) – описують цілі/задачі користувачів системи, які повинні досягаються/виконуються користувачами за допомогою створюваної програмної системи;
- функціональні вимоги (Functional Requirements) – вимоги, що конкретизують функції, які система або її компонент повинен виконувати;
- програмні вимоги (Software Requirements) – вимоги до створюваної системи, зрозумілі користувачам (замовникам) і розробникам (виконавцям) стосовно того, що робитиме система і чого від неї не варто чекати.

Досвід показує, що оцінка складності системи, що є сумою оцінок її компонентів, отриманих у результаті декомпозиції, значно точніша, ніж первісна оцінка системи в цілому. Використання засобів формалізації результатів аналізу для їх документального оформлення також підвищує якість початкового опису вимог до системи.

Коли говорять про "формалізацію постановки завдання", мають на увазі розроблення послідовності моделей, кожна з яких описує систему та її оточення з різних точок зору із поступовою деталізацією. Важливо, щоб усі уявлення про систему, отримані в різних моделях, збиралися в єдиному репозитарії (деякій спеціалізованій базі даних). Цей репозитарій полегшить наскрізне проектування, при якому кожна наступна модель використовує результати попередньої і ніяк їм не суперечить. Відповідно всі можливі перевірки повинні бути наскрізними.

Для якісного визначення вимог до ПЗ потрібно спочатку провести аналіз та сформувати їх специфікації.

2. Аналіз завдання та моделювання програмного забезпечення

Ціль аналізу і моделювання - виявлення ясної і відносно простої внутрішньої структури (що іноді називається архітектурою проекту), яка:

- Задовольняє заданим функціональним властивостям і специфікаціям;
- Погоджена з обмеженнями, що накладені апаратним забезпеченням;

- задовольняє явним і неявним експлуатаційним вимогам;
- задовольняє явним і неявним критеріям дизайну;
- задовольняє вимогам до процесу розробки (тривалість, вартість).

До цього етапу розробки програмного забезпечення також належить процедура проведення всебічного аналізу висунутих замовником вимог до створюваного ПЗ, щоб визначити ключові цілі та завдання кінцевого продукту. В рамках цієї стадії відбувається максимально ефективну взаємодію потребує програмному рішенню клієнта і співробітників компанії-розробника, в ході обговорення деталей проекту допомагають більш чітко сформулювати вимоги до ПЗ. Результатом проведеного аналізу стає формування основного регламенту, на який буде спиратися виконавець у своїй роботі - технічного завдання на розробку програмного забезпечення. ТЗ повинно повністю описувати поставлені перед розробником завдання і охарактеризувати кінцеву мету проекту в розумінні замовника.

Продуктом аналізу і проектування є моделі, що дозволяють розробникам і замовникам зрозуміти структуру майбутньої системи, збалансувати вимоги і узгодити схему реалізації.

У процесі створення таких моделей використовуються сучасні методології й інструменти об'єктно-орієнтованого аналізу і проектування. Це дозволяє йти в ногу зі зростаючими вимогами проектування, реалізовувати складні бізнес- і технологічні процеси.

Аналіз вимог (Requirements Analysis) – трансформація інформації, отриманої від користувачів (та інших зацікавлених осіб) у чітко та однозначно визначені програмні вимоги, що передаються інженерам для реалізації у програмному коді. Аналіз вимог включає:

- виявлення і розв'язання конфліктів між вимогами;
- визначення меж задачі, що вирішується створюваним програмним забезпеченням; у загальному випадку – визначення меж (Scope) і змісту програмного проекту;
- деталізацію системних вимог для встановлення програмних вимог.

Специфікація (Specification) – документ, що в закінченій, точній і перевіреній формі описує вимоги, проект, поведінку або інші характеристики компонента або системи, а також процедури, спрямовані на визначення того, чи задовольняються описані характеристики [21]. Для опису комплексних проектів (у частині вимог) використовують три основні специфікації:

- визначення системи (System Definition), або специфікація вимог користувачів (User Requirements Specification);
- системних вимог (System Requirements);
- програмних вимог (Software Requirements).

Специфікація програмних вимог (Software Requirements Specification – SRS) встановлює основні угоди між користувачами (замовниками) і розробниками (виконавцями) стосовно того, що робитиме система і чого від неї не варто чекати. Цей документ може включати процедури перевірки створеного ПЗ на відповідність вимогам, що висуваються (у т.ч. плани тестування), описи характеристик стосовно якості та методів його оцінювання, питань безпеки тощо [20].

Специфікація вимог користувачів (User Requirements Specification) або концепція (concept <of operation>) визначає високорівневі вимоги, часто –

стратегічні цілі, для досягнення яких створюється програмна система. Важливо, що цей документ описує вимоги до системи з позицій предметної області – домену [20].

Специфікація системних вимог (System Requirements) – описує програмну систему в контексті системної інженерії. Зокрема високорівневі вимоги до програмного забезпечення, що містить кілька або багато взаємозв'язаних підсистем і застосувань. При цьому система може бути як цілком програмною, так і містити програмні та апаратні компоненти. У загальному випадку до складу системи може входити персонал, що виконує певні функції системи, наприклад, авторизацію виконання певних операцій з використанням програмно-апаратних підсистем [20].

При постановці завдання потрібно, щоб програмні вимоги були зрозумілі, зв'язки між ними прозорі, а зміст специфікації не допускав різночитань та інтерпретацій, через які програмний продукт не буде відповідати потребам зацікавлених осіб. Тому потрібні інструменти управління вимогами.

Управління вимогами (Requirements Management) – діяльність, виконання якої забезпечує опис вимог, відстежування їх змін, перевірки на несуперечливість і на порушення наперед визначених правил [21].

Від вхідної інформації про майбутній програмний продукт залежить те, яку методологію буде обрано в проекті з розроблення ПЗ. Методологій багато: і дуже формалізованих, і тих, що дають творчу свободу програмістам. Вибір методології обумовлюється досвідом керівника групи розробників та умовами, які встановлюють замовники до документування етапів робіт. У роботі [22] методології розроблення ПЗ класифікуються за кількістю виконавців та критичністю проекту. Чим більше виконавців

та/або вища критичність, тим більш формальна та регульована методика потрібна.

3. Розробка або вибір алгоритму розв'язання задачі

Діяльність під час розроблення ПЗ, як і будь-що, складається з виконання операцій і проектів. Ті й інші мають багато спільного, наприклад, виконуються людьми та на їх виконання виділяються обмежені ресурси.

Головна відмінність операцій від проектів полягає в тому, що операції виконуються постійно і повторюються, тоді як проект тимчасовий і унікальний. Виходячи з цього, проект визначається як тимчасове зусилля, розпочате для створення унікального продукту чи послуги. «Тимчасове» означає, що кожен проект має точно визначені дати початку та закінчення. Говорячи про унікальність продукту, ми маємо на увазі, що вони мають помітні відмінності від усіх аналогічних продуктів або послуг. Таким чином, розроблення ПЗ відповідає визначенню проекту і для організації цього процесу можна застосовувати методи та інструментарій управління проектами. Наприклад, розроблення веб-сайту є проектом, тоді як підтримка його впродовж тривалого часу – це операційна діяльність.

У кожного проекту є чітко визначені початок і кінець. Кінець проекту настає разом із досягненням усіх його цілей або коли стає зрозумілим, що ці цілі не будуть або не можуть бути досягнуті. Тимчасовість не означає короткостроковість проекту – розроблення складної програмної системи може тривати кілька років, хоча, як правило проекти мають обмежені часові рамки для створення ПЗ, оскільки сприятлива для них ситуація на ринку складається на обмежений час. Крім того, проектна команда після його закінчення розпадається, а її члени переходять в інші проекти.

Проект дуже часто плутають із програмою, тобто координованим управлінням групою проектів всередині однієї організації. Управління відразу декількома проектами скоординоване для того, щоб отримати вигоду, яку неможливо одержати від окремого управління кожним із них.

Розробка алгоритму – розробка та представлення алгоритму розв'язування задачі у вигляді певної, точно визначеної послідовності операцій ЕОМ. Це можна зробити словесно-формульно або за допомогою блок-схеми.

Набір даних, що містить всю необхідну і достатню інформацію про досліджувані об'єкти чи процеси називають інформаційною моделлю. Інформаційна модель — це множина тих і тільки тих даних про об'єкт, які необхідні для розв'язання задачі що цікавить дослідника. Інформаційні моделі - це знакові моделі, які описуються з використанням певних систем знаків. Інформаційні моделі можна подавати за допомогою розмовної мови, спеціальних та формальних мов, можна для цього використовувати певні системи графічних позначень (знаків), графи тощо.

Інформаційна модель, яка містить усі необхідні дані для розв'язання певної задачі і поміщена в пам'ять комп'ютера називається комп'ютерною моделлю. Комп'ютерна модель будується на основі певної математичної та інформаційної моделей і реалізується апаратно-програмними засобами.

Складання програми на алгоритмічній мові – здійснюється переклад

алгоритму задачі на мову конкретної машини або відповідну алгоритмічну мову. Розробкою програми завершуються етапи розв'язування задачі, що виконуються людиною без використання ЕОМ. Решта етапів виконуються з використанням комп'ютера.

Вибір або розробка алгоритму й чисельного методу розв'язання задачі мають найважливіше значення для успішної роботи над програмою. Ретельно прооблений алгоритм розв'язання задачі – необхідна умова ефективної роботи зі складання програми.

4. Проектування загальної структури програми

Наступний ключовий етап в розробці програмного забезпечення - стадія проектування, тобто моделювання теоретичної основи майбутнього продукту. Найсучасніші засоби програмування дозволяють частково об'єднати етапи проектування та кодування, тобто технічної реалізації проекту, будучи заснованими на об'єктно-орієнтованому підході, але повноцінне планування вимагає більш ретельного і скрупульозного моделювання.

Якісний аналіз перспектив і можливостей створюваного продукту стане основою для його повноцінного функціонування і виконання всього комплексу покладених на ПЗ завдань. Однією із складових частин етапу проектування, наприклад, є вибір інструментальних засобів і операційної системи, яких сьогодні на ринку присутня дуже велика кількість.

На цьому етапі відбувається «архітектурне» прооблення проекту. Визначаються ті частини алгоритму, які доцільно оформити у вигляді підпрограм, модулів. Визначається й спосіб зберігання інформації – у вигляді набору простих змінних, масивів або інших структур.

В рамках даного етапу сторони повинні здійснити:

- оцінку результатів проведеного спочатку аналізу і виявлених обмежень;пошук критичних ділянок проекту;
- формування остаточної архітектури створюваної системи;аналіз необхідності використання програмних модулів або готових рішень сторонніх розробників;
- проектування основних елементів продукту - моделі бази даних, процесів і коду;
- вибір середовища програмування і інструментів розробки, затвердження інтерфейсу програми, включаючи елементи графічного відображення даних;визначення основних вимог до безпеки розробляється ПЗ.

Отже, етапи і результати проектування:

1. Опис: спільна робота замовника (говорить про користь продукту, вимоги до працездатності та зовнішнім виглядом) і розробника (пропонує технічні та алгоритмічні рішення).
2. Архітектура: затверджується мова програмування, база даних, сервери і фреймворки.
3. Технічне завдання: складається архітектором на підставі опису та відповідей замовника на питання, узгоджується з менеджером проекту, потім передається клієнту, виробляються

правки.

4. Макети (додаються до техзавдання): інтерфейсів, принципові схеми улаштування, діаграми структури бази даних, схеми взаємодії компонентів.
5. Контроль: архітектор усуває зауваження менеджера проектів.
6. Затвердження: замовник перевіряє і змінює ТЗ самостійно або повідомляє список правок проект-менеджеру, зауваження усуваються, ТЗ затверджується і додається до контракту.

Як результат проектування, ми отримуємо технічне завдання зі зрозумілою і однозначною для замовника і виконавця (керівника проекту, програмістів, тестувальників, дизайнерів та інших учасників процесу розробки) ілюстрацією відповідей на питання:

- Що робимо (опис продукту, функціоналу, користувачів)?
- Як робимо (архітектура)?
- Як перевірити, що мета досягнута (тестування, критерії оцінки)?

Методологія проектування поєднує в собі об'єктну декомпозицію, прийоми уявлення фізичної, логічної, а також динамічної та статичної моделей системи.

Під час проектування розробляються проектні рішення щодо вибору платформи, де буде функціонувати система мови або мов реалізації, призначаються вимоги до призначеного для користувача інтерфейсу, визначається найбільш підходяща СКБД. Розробляється функціональна специфікація ПЗ: вибирається архітектура системи, обумовлюються вимоги до апаратного забезпечення, визначається набір організаційних заходів, які необхідні для впровадження ПЗ, а також перелік документів, що регламентують його використання.

Мінімально достатнє ТЗ повинно:

- повністю, чітко (інструкційно, без можливості різночитання) і структуровано описувати майбутній програмний продукт (як повинен виглядати, як і з чим працювати, яким вимогам відповідати) і процес його розробки, щоб у архітектора не виникало питань щодо реалізації;
- виключати суперечливі відомості;
- бути юридично точним, оскільки разом з контрактом та іншими документами ТЗ набуває юридичну силу.

Технічне завдання повинно містити:

- загальні дані про проект (назва продукту, ким і для чого буде використовуватися);загальні вимоги до ПЗ (до структури, функцій, зокрема докласти схему архітектури і описати зв'язок підсистем, види інтерфейсів всіх складових для кожної з ролей користувачів - готовий дизайн або його концепцію);
- докладний план робіт (перелік етапів, терміни по ним);порядок тестування та приймання (види і склад випробувань продукту в цілому і окремих частин);
- перелік дій для запуску продукту;
- вимоги до документування процесу і результату розробки. У складі ТЗ

необхідно приділити увагу опису:

1. **деталей**:
   - користувачі програмного продукту: ролі, права і функції;
   - опис алгоритмів обробки даних;
   - перелік відкритих і закритих протоколів;
   - вимоги до безпеки даних на всьому життєвому циклі;
   - список компонентів (платних, вільних), які будуть використовуватися в розробці.
2. **прикладів**:
   - при наявності аналогів, що інтегруються вказуються посилання на них;
   - в описі роботи системи наводиться опис типових сценаріїв взаємодії з нею користувачів;
   - приклади вхідних даних і формат даних взаємодії підсистем (таблиці, бази, сторінки та ін.);
   - приклади вихідних даних (види звітів і експортованих файлів).
3. **продуктивності і надійності**:
   - вказівка рівнів навантаження системи (день, місяць, максимальний);
   - вимоги до продуктивності, збереження;
   - обґрунтування вибору устаткування запуску програмного забезпечення;
   - вказівка хостингу серверної частини.

5. Кодування

Наступним кроком стає безпосередня робота з кодом, спираючись на обраний в процесі підготовки мову програмування.

Кодування – це запис алгоритму мовою програмування. Якщо алгоритм розв'язання задачі, структура програми й структура даних ретельно продумані й акуратно записані, витрати часу на кодування зменшуються ймовірність помилок на цьому етапі знижується.

Описувати особливості і тонкощі самого трудомісткого і складного етапу навряд чи варто, досить вказати, що успіх реалізації будь-якого проекту прямо залежить від якості попереднього аналізу і оцінки конкуруючих рішень, з якими створюваної програмі має бути «боротися» за право називатися кращою в своїй ніші. Якщо мова йде про написання коду для виконання вузькоспеціалізованих завдань в рамках конкретного підприємства, то від грамотного підходу до етапу кодування залежить ефективність роботи компанії, що замовила розробку. Кодування може відбуватися паралельно з наступним етапом розробки - тестуванням програмного забезпечення, що допомагає вносити зміни безпосередньо по ходу написання коду.

Рівень і ефективність взаємодії всіх елементів, задіяних для виконання сформульованих завдань компанією-розробником, на поточному етапі є найважливішим - від злагодженості дій програмістів, тестувальників і проектувальників залежить якість реалізації проекту.

Від якості коду залежать робота та надійність системи. Якісне кодування потребує постійного навчання та удосконалення. Також код містить інформацію про стан проекту у найбільш стислій та чітко зрозумілій формі.

Програмісти пишуть увесь код у відповідності до правил, які забезпечують комунікацію за допомогою коду.

6. Налагодження і тестування програми

Після досягнення задуманого програмістами в написаному коді слідують не менш важливі етапи розробки програмного забезпечення, часто об'єднуються в одну фазу - тестування продукту і подальше налагодження, що дозволяє ліквідувати огріхи програмування і домогтися кінцевої мети - повнофункціональної роботи розробленої програми. Процес тестування дозволяє змоделювати ситуації, при яких програмний продукт перестає функціонувати. Відділ налагодження потім локалізує і виправляє виявлені помилки коду, «вилизуючи» його до практично ідеального стану. Ці два етапи займають не менше 30% витрачається на весь проект часу, так як від їх якісного виконання залежить доля створеного силами програмістів програмного забезпечення. Нерідко функції тестувальника і відладчика виконує один відділ, проте найоптимальнішим буде розподілити ці обов'язки між різними виконавцями, що дозволить збільшити ефективність пошуку наявних в програмному коді помилок.

Налагодження й верифікація програми являють собою дуже важливу частину процесу розробки програми. Налагодження полягає в усуненні помилок програмування, помилок перекладу алгоритму на мову програмування.

Верифікація – це доказ того, що програма працює «вірно», дає правильний результат. Для цього розробляється система тестів, які можуть являти собою спеціально підібрані набори параметрів, для яких задача може бути вирішена точно. Це можуть бути, наприклад, які-небудь граничні випадки. Якщо результат, отриманий за допомогою програми, збігається (з урахуванням погрішності машинного лічіння) з очікуваним, є підстава думати, що програма працює коректно. Але це усього лише підстава, а не абсолютна впевненість!

Серед починаючих програмістів поширене переконання, що якщо програма успішно відкомпільована й, будучи запущена на виконання, видає на екран ряди цифр, задача вирішена. Насправді ж програма готова, якщо розроблювач спромігся довести замовникові (та й самому собі), що результат роботи програми є рішенням поставленого завдання.

Користуючись висловом Е. Дейкстра «Програма, вільна від помилок — це абстрактне теоретичне поняття» можна зазначити, що тестування виявляє лише наявність, але ніяк не відсутність помилок.

Мало в якому виді діяльності існує стільки можливостей для помилок, як у програмуванні. Одним з критеріїв професійної майстерності програмістів є їх спроможність виявляти та виправляти власні помилки. Програмування – це досить складна задача. Ми намагались описати різні технології програмування, мета яких, в першу чергу, зробити програми структурними і зрозумілими. Але жодна з цих технологій не здатна в корені змінити сумного факту – помилки в програмі зустрічаються завжди. Ми знаходимо їх за допомогою тестування, а усуваємо за допомогою налагодження. Починаючі програмісти не вміють цього робити, досвідчені – вміють, але помилки роблять усі без виключень. Але хороші

програмісти знають, що основний час при програмуванні буде витрачений на тестування та налагодження. Нижче ми обговоримо, як можна скоротити час на налагодження програми і як зробити цей процес більш технологічним. Визначимо зміст ключових слів даного розділу.

Тестування – це виконання комплексу вправ (завдань) для перевірки працездатності програми за будь-яких умов. Тестування може виявити факт наявності помилки, а налагодження виявляє причину помилки, так що ці два етапи розробки “перекриваються”.

Тестування програмного продукту дозволяє протягом усього життєвого циклу ПО гарантувати, що програмні проекти відповідають заданим параметрам якості.

Протягом усього життєвого циклу розробки ПЗ застосовуються різні типи тестування для гарантії того, що проміжні версії відповідають заданим показникам якості (рис. 3.2). При цьому застосовуються автоматичні і ручні тести. Типи тестувань, які використовуються сучасними розробниками ПЗ наведено в табл. 3.1.

Таблиця 3.1. Типи тестувань ПЗ

Тип тестування	Призначення
Модульне тестування	для перевірки правильності функціонування методів класів ПО. Модульні тести пишуться і виконуються розробниками в процесі написання коду. Модульне тестування застосовується як для перевірки якості коду програми, так і для перевірки об'єктів баз даних.
Дослідницьке тестування	для тестування, при якому тестувальник не має заздалегідь визначених тестових сценаріїв і намагається інтуїтивно досліджувати можливості програмного продукту і виявити і зафіксувати невідомі помилки.
Інтеграційне тестування	використовується для перевірки коректності спільної роботи компонентів програмного продукту.
Функціональне тестування	передбачає перевірку конкретних вимог до ПО і проводиться після додавання до системи нових функцій.
Тестування навантаженням	призначене для перевірки працездатності програмного продукту при граничній вхідний навантаженні.
Регресійне тестування	при внесенні змін до програмного забезпечення з метою перевірки коректності роботи компонентів системи, які потенційно можуть взаємодіяти зі зміненим компонентом.
Комплексне тестування	для тестування функціональних і не функціональних вимог всієї системи програмного продукту.
Приймальне тестування	являє собою функціональні випробування, які повинні підтвердити те, що програмний продукт відповідає вимогам і очікуванням користувачів і замовників. Приймальні тести пишуться бізнес-аналітиками, фахівцями з контролю якості та тестувальниками.

Головна мета тестування - визначити відхилення в реалізації функціональних вимог, виявити помилки у виконанні програм і виправити їх якомога раніше в процесі виконання проекту.

Налагодження – це процес, який починається з моменту встановлення існування помилки і закінчується локалізацією цієї помилки в програмі, тобто визначенням її характеру та місцезнаходження. Таким чином, налагодження програми передбачає обов'язкову наявність помилки.

Налагодження програм досить складний процес. По-перше, для виправлення помилки необхідно повністю виявити її причини, які часто далеко неочевидні. По-друге, ця діяльність психологічно носить негативний характер, в тому розумінні, що програміст повинен визнати, що саме його помилка є причиною програмного збою. Крім того, налагодження – це процес, який призупиняється лише тимчасово, поки тестування не виявить наявність чергової помилки.

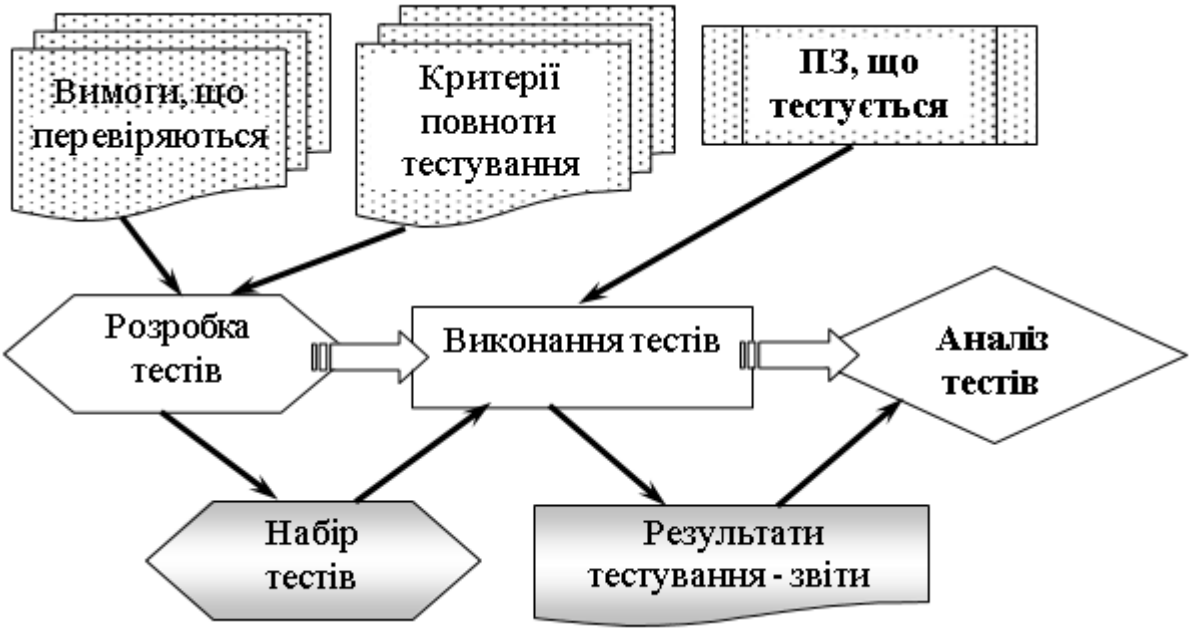


Рисунок 2 - Схема процесу тестування

Якщо програма не працює, або працює, але видає неправильні результати, існують три основні методи налагодження програми, кожен з яких має свої особливості.

1) Використати при створенні програми всі методи, які дозволили б зменшити кількість помилок в програмі, а у випадку їх виявлення використовувати переважно налагодження вручну, тобто перегляд тексту програми та ретельний його аналіз.

2) Переважне використання програмних засобів комп'ютера – так званих налагоджувачів (англ. debuggers) для пошуку помилок. Як варіант цього метода можна розглядати трасування, або можливість покрокового виконання програми і слідкування за значеннями змінних в процесі виконання програми, яке є доступним у деяких програмних середовищах.

Цей метод не гарантує знаходження всіх помилок, тому що програмний налагоджувач може працювати більш коректно, ніж компілятор – наприклад, обнуляти невизначені змінні або інакше розподіляти пам’ять.

3) Поєднує створення програми з одночасним налагодженням та тестуванням її частин. Цей метод вимагає високої самодисципліни програміста і є більш ефективним у випадку аналітичного програмування (програмування згори донизу).

Важко сказати, якому підходу слід надавати перевагу. Скоріше за все це визначається характером програми та особистими прихильностями її розробника.

Тестування тісно пов'язане з такими етапами розробки програмного забезпечення як проектування і реалізація. У систему вбудовуються спеціальні механізми, які дають можливість виробляти тестування системи на відповідність вимог до неї, перевірку оформлення і наявність необхідного пакету документації.

Результатом тестування є усунення всіх недоліків системи і висновок про її якість.

7. *Аналіз результатів*

Одержання результату, його інтерпретація з можливою наступною модифікацією моделі.

Коли програма перевірена, більша частина помилок усунута і є обґрунтована надія на те, що, принаймні в рамках обраної моделі, вона дає правильний результат. Цей результат необхідно проаналізувати. Якщо мова йде про моделювання якогось природного процесу, варто порівняти отримані за допомогою комп'ютера результати й результати спостережень.

Процес такого аналізу називається інтерпретацією результатів розрахунку. Тут програміста може очікувати розчарування – результат може відрізнятися від необхідного. У цьому випадку, можливо, доведеться змінити саму модель, зробивши її більш реалістичною.

Після отриманого нами файла з *.exe (зазвичай), ми можемо запустити його й вкотре перевірити (проаналізувати) чи вірно працює програма. У цьому етапи створення програми закінчено.

8. *Публікація результатів роботи*

Останньою складовою процесу програмування є документування. Воно включає широкий, спектр описів, які полегшують процес програмування і узагальнюючих результуючу програму. Постійне документування має становити невід'ємну частину кожного кроку програмування. Постановка завдання, проектні документи, алгоритми і програми – усе це документи.

Внутрішня документація, включена у програму, полегшує читання коду. Призначення навчального посібника (ще з однією форми документації) – навчити користувача застосовувати нову програму; довідкове керівництво дозволяє ознайомитися з описом команд програмного забезпечення.

Документація програмного забезпечення (англ. software documentation) - супроводжуючі документи до програмного забезпечення, які містять в собі інформацію, що описує загальні положення необхідні для ознайомлення перед тим як використовувати його за призначенням. Така документація дуже важлива і описує не тільки яким чином правильно використовувати поставлене програмне забезпечення, а й пояснює основні використані алгоритми. В залежності від складності кожного окремого програмного забезпечення, його специфіки, а також ліцензії під якою воно створене - документація може варіюватися за обсягом і за

змістом.

Під час розробки програми створюється великий обсяг різноманітної документації. Вона необхідна як передачі між розробниками програми, як управління розробкою програми розвитку й як передачі користувачам інформації, яка потрібна на застосування і супроводження програми.

9. *Супровід програми*

Результат зусиль по розробці програмного забезпечення полягає в передачі в експлуатацію програмного продукту, який задовольняє вимогам користувачів. Відповідно, в процесі експлуатації продукт буде змінюватися або еволюціонувати. Пов'язано це з виявленням при реальному використанні прихованих дефектів, змінами в операційному оточенні, необхідністю покриття нових вимог і т.п.

Фаза супроводу в життєвому циклі, зазвичай, починається відразу після приймання/передачі продукту і діє протягом гарантійного терміну або, частіше, технічної підтримки. Однак, самодіяльність, пов'язана з супроводом, починається набагато раніше.

Програмні засоби є одним з найбільш гнучких видів промислових виробів і епізодично піддаються змінам протягом усього часу їх використання.

Іноді досить при коригуванні програмного забезпечення внести тільки одну помилку для того, щоб різко знизилася його надійність або його коректність при деяких вихідних даних.

Для збереження і підвищення якості програмного забезпечення необхідно регламентувати процес модифікації і підтримувати його відповідним тестуванням і контролем якості. В результаті програмний виріб з часом зазвичай поліпшується як за функціональними можливостями, так і за якістю рішення окремих завдань.

Роботи, що забезпечують контроль і підвищення якості, а також розвиток функціональних можливостей програм, складають процес супроводу.

У процесі супроводу в програмне забезпечення вносяться наступні зміни, значно різняться причинами і характеристиками:

- Виправлення помилок - коригування програм, що видають неправильні результати в умовах, обмежених технічного завдання та документацією. Виправлення помилок вимагають близько 20% загальних витрат на супровід.
- Регламентована документами адаптація програмного забезпечення до умов конкретного використання, з урахуванням характеристик зовнішнього середовища або конфігурації апаратури, на якій має бути функціонувати програмами. Адаптація займає близько 20% загальних витрат на супровід.
- Модернізація - розширення функціональних можливостей або поліпшення характеристик вирішення окремих завдань відповідно до нового або додатковим технічним завданням на програмне виріб. Модернізація займає до 60% загальних витрат на супровід (рис. 3).

Супроводження програми – підтримка працездатності програми, перехід на її нові версії, внесення змін, виправлення помилок, а також процес покращення, оптимізації та виправлення дефектів у програмному забезпеченні після його вводу до експлуатації.

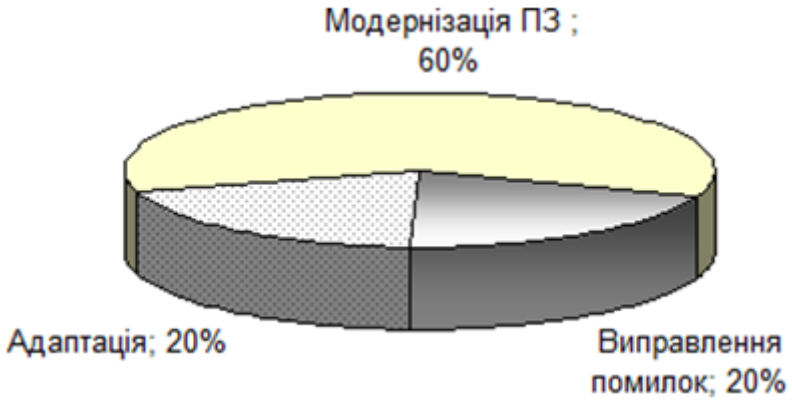


Рисунок 3 - Витрати на супровід програмного забезпечення

Процес супроводження містить у собі моделі процесу супроводу і планування діяльності людей, що проводять запуск ПЗ, перевірку правильності його виконання і внесення в нього змін. Цей процес згідно з стандартом ISO/IEC 14764 проводиться шляхом:

- коригування, вдосконалення продукту для усунення виявлених помилок або нереалізованих задач;
- адаптація, підлаштування продукту до умов експлуатації, що змінилися, або в новому середовищі виконання;
- поліпшення, еволюційна зміна продукту для підвищення продуктивності або рівня супроводу;
- перевірка ПЗ, пошук і виправлення помилок при експлуатації системи.

Ключові питання супроводу ПЗ – це управлінські, вимірювальні і вартісні.

Суть управлінських питань – контроль ПЗ при модифікації й удосконалюванні функцій і недопущення зниження продуктивності системи. Питання вимірювання пов'язане з оцінкою характеристик системи після її модифікації, а також повторного тестування для оцінки показників якості. Вартісні питання пов'язані з оцінкою витрат на супровід залежно від його типу, кваліфікації персоналу, платформи й ін.

В ході підтримки виправляються виявлені дефекти і недоробки. Також додається нова функціональність, вносяться зміни для підвищення зручності використання (юзабіліті) програми.

Послуги з підтримки програмного забезпечення включають в себе такі роботи як:

Виправлення помилок і усунення неполадок, невиявлених раніше.

Оптимізація роботи програми при різних умовах експлуатації.

Оновлення та доопрацювання за вимогами Замовника.

Профілактичні роботи по обслуговуванню баз даних інформаційної системи.

Підготовка технічної і призначеної для користувача документації.

Оновлення модулів програми і використовуваних бібліотек з урахуванням сучасних технологій.

Роботи по супроводу програмного забезпечення проводяться в тісному контакті зі співробітниками замовника, що дозволяє більш динамічно розвивати програмне забезпечення, оперативно змінюючи пріоритети розробки. Також скорочується час, необхідний на узгодження плану робіт, оскільки доповнення та

виправлення зазвичай несуть менш глобальний характер, ніж при розробці ядра програми.

Необхідний пакет послуг з підтримки обмовляється з кожним клієнтом індивідуально.

Створення навіть невеликого і технічно простого ПО залежить від чіткого виконання кожної фази, тобто діяльності всіх відділів, задіяних в процесі розробки. Чіткий план виконання необхідних заходів з зазначенням кінцевої мети стає невід'ємною частиною роботи розробників, які планують залишатися широко затребуваними на ринку праці фахівцями. Тільки правильно складене технічне завдання дозволить домогтися потрібного результату і здійснити розробку по-справжньому якісного і конкурентного ПО для будь-якої платформи - серверної, стаціонарним або мобільним.

Невід'ємною частиною завершального етапу розробки програмного забезпечення також є подальша технічна підтримка створеного продукту в процесі його експлуатації на підприємстві замовника. Грамотно організована служба техпідтримки найчастіше стає ключовим фактором при виборі виконавця в рамках досягнення поставленої мети.

Змістовий модуль №2

Уніфікована мова моделювання UML.

Тема: Мова моделювання UML. Структура і компоненти мови UML.

Мета: Ознайомитись з поняттям UML. Розглянути словник UML. Дати визначення предметам UML.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. UML;
2. Предмети в UML;
   a. Структурні предмети;
   b. Предмети поведінки;
   c. Предмети, що групують;
   d. Предмети, що пояснюють;
3. Відносини в UML;
4. Діаграми в UML.
5. Механізми розширення.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Що таке UML?
2. З чого складається словник UML?
3. Які предмети UML бувають?
4. Що таке структурні предмети? Перелічіть їх.
5. Які існують предмети поведінки?
6. Що таке пакет?
7. Для чого використовують примітку?
8. Що таке діаграма?
9. Які існують діаграми в UML?
10. Перелічіть діаграми взаємодії.

Уніфікована мова моделювання (UML) є стандартним інструментом для створення «креслень» програмного забезпечення. За допомогою UML можна візуалізувати, специфікувати, конструювати і документувати артефакти програмних систем.

UML придатний для моделювання будь-яких систем: від інформаційних систем масштабу підприємства до розподілених Web-додатків і навіть вбудованих систем реального часу.

UML – це мова для візуалізації, специфікації, конструювання та документування артефактів програмних систем. Мова моделювання, подібна до UML, є стандартним засобом для складання «креслень» програмного забезпечення.

Для розуміння будь-якої нетривіальної системи доводиться розробляти велику кількість взаємопов'язаних моделей. У застосуванні до програмних систем це означає, що необхідна мова, за допомогою якої можна з різних точок зору описати уявлення архітектури системи протягом циклу її розробки.

UML – це графічна мова специфікації, що означає побудову точних і повних графічних моделей, що стосуються аналізу, проектування і реалізації, які повинні прийматися в процесі розробки і розгортання системи програмного забезпечення.

UML – це мова конструювання, і моделі, створені за її допомогою, можуть бути безпосередньо переведені на різні мови програмування. Іншими словами, UML-модель можна відобразити на такі мови, як Java, C++, Visual Basic, і навіть на таблиці реляційної бази даних.

Таке відображення моделі на мову програмування дозволяє здійснювати пряме проектування: генерацію коду з моделі UML в якусь конкретну мову. Можна вирішити і зворотну задачу: реконструювати модель за наявною реалізацією.

UML дозволяє вирішити проблему документування системної архітектури і всіх її деталей, пропонує мову для формулювання вимог до системи і визначення тестів і, нарешті, надає засоби для моделювання робіт на етапі планування проекту і управління версіями.

Використання UML ефективне в:

- інформаційних системах масштабу підприємства;
- банківських і фінансових послугах;
- телекомунікаціях;
- на транспорті;
- оборонній промисловості, авіації та космонавтиці;
- роздрібній торгівлі;
- медичній електроніці;
- науці;
- розподілених Web-системах.

Сфера застосування UML не обмежується моделюванням ПЗ. Він дозволяє моделювати, скажімо, документообіг в юридичних системах, структуру і функціонування системи обслуговування пацієнтів в лікарнях, здійснювати проектування апаратних засобів.

Словник UML утворюють три різновиди будівельних блоків: предмети, відносини, діаграми.

Предмети – це абстракції, які є основними елементами в моделі, відносини зв'язують ці предмети, діаграми групують колекції предметів.

Предмети в UML

В UML є чотири різновиди предметів:

- структурні предмети;
- предмети поведінки;
- предмети, що групують;
- предмети, що пояснюють.

Ці предмети є базовими об'єктно-орієнтованими будівельними блоками. Вони використовуються для написання моделей.

Структурні предмети є іменниками в Uml- Моделях. Вони представляють статичні частини моделі - понятійні або фізичні елементи. Перелічимо вісім різновидів структурних предметів.

1. *Клас* - опис безлічі об'єктів, які розділяють однакові властивості, операції, відносини й семантику (зміст). Клас реалізує один або кілька інтерфейсів. Як показано на рис. 1.1, графічно клас відображається у вигляді прямокутника, що звичайно включає секції з іменем, властивостями (атрибутами) і операціями.

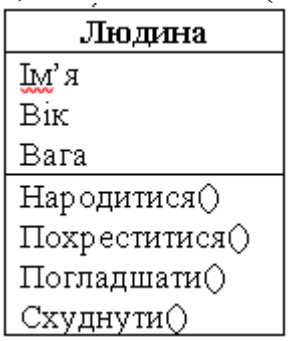


Рисунок 1.1 – Клас

2. *Інтерфейс* - набір операцій, які визначають послуги класу або компонента. Інтерфейс описує поведінку елемента, видиму ззовні. Інтерфейс може представляти повні послуги класу або компонент або частина таких послуг. Інтерфейс визначає набір специфікацій операцій (їх сигнатури), а не набір реалізацій операцій. Графічно інтерфейс зображується у вигляді кружка з іменем, як показано на рис. 1.2. Ім'я інтерфейсу звичайно починається з букви "I". Інтерфейс рідко показують самостійно. Звичайно його приєднують до класу або компонента, який реалізує інтерфейс.

IНавчання

Рисунок 1.2 – Інтерфейс

3. *Кооперація (співробітництво)* визначає взаємодія і є сукупністю ролей і інших елементів, які працюють разом для забезпечення колективної поведінки більш складного, чому проста сума всіх елементів. Таким чином, кооперації мають як структурний, так і поведінковий вимір. Конкретний клас може брати участь у декількох кооперациях. Ці кооперації представляють реалізацію паттернів (зразків), які формують систему. Як показано на рис. 1.3, графічно кооперація зображується як пунктирний еліпс, у який вписується її ім'я.

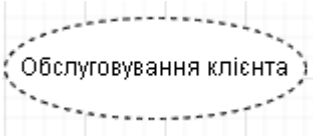


Рисунок 1.3 – Кооперація

4. *Актор* - набір погоджених ролей, які можуть відіграти користувачі при взаємодії із системою (її елементами Use Case). Кожна роль вимагає від системи певного поведінки. Як показано на рис. 1.4, актор зображується як дротовий чоловічок з іменем.

Замовник

Рисунок 1.4 – Актор

5. *Елемент Use Case (Прецедент)* - опис послідовності дій (або декількох послідовностей), виконуваних системою в інтересах окремого актора й виробляючих видимий для актора результат. У моделі елемент Use Case застосовується для структурування предметів поведінки. Елемент Use Case реалізується кооперацією. Як показано на рис. 1.5, елемент Use Case зображується як еліпс, у який уписується його ім'я.

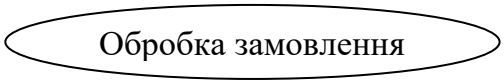


Рисунок 1.5 – Прецедент

6. Активний клас - клас, чиї об'єкти мають один або кілька процесів (або потоків) і тому можуть ініціювати керуючу діяльність. Активний клас схожий на звичайний клас за винятком того, що його об'єкти діють одночасно з об'єктами інших класів. Як показано на рис. 1.6, активний клас зображується як стовщений прямокутник, що звичайно включає ім'я, властивості (атрибути) і операції.

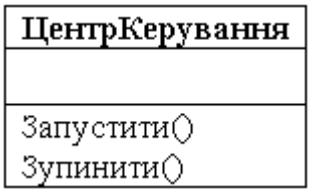


Рисунок 1.6 – Активний клас

7. *Компонент* - фізична й замінна частина системи, яка відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів. У систему включаються як компоненти, що є результатами процесу розробки (файли вихідного коду), так і різні різновиди використовуваних компонентів (COM+-компонента, Java Beans). Звичайно компонент - це фізичне впакування різних логічних елементів (класів, інтерфейсів і співробітництв). Як показано на рис. 1.7, компонент зображується як прямокутник із вкладками, що звичайно включає ім'я.

Рисунок 1.7 – Компонент

8. *Вузол* - фізичний елемент, який існує в період роботи системи й представляє ресурс, що звичайно має пам'ять і можливості обробки. У вузлі розміщається набір компонентів, який може переміщатися від вузла до вузла. Як показано на рис. 1.8, вузол зображується як куб з іменем.

Рисунок 1.8 – Вузол

Предмети поведінки - динамічні частини Uml- Моделей. Вони є дієсловами моделей, виставою поведінки в часі й просторі. Існує два основні різновиди предметів поведінки.

1. *Взаємодія* - поведінка, що містить у собі набір повідомлень, якими обмінюється набір об'єктів у конкретному контексті для досягнення певної мети. Взаємодія може визначати динаміку як сукупності об'єктів, так і окремої операції. Елементами взаємодії є повідомлення, послідовність дій (поведінка, викликуване повідомленням) і зв'язку (з'єднання між об'єктами). Як показано на рис. 1.9, повідомлення зображується у вигляді спрямованої лінії з іменем її операції.

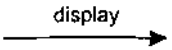


Рисунок 1.9 – Повідомлення

2. *Кінцевий автомат* - поведінка, яка визначає послідовність станів об'єкта або взаємодії, виконувані в ході його існування у відповідь на події (і з урахуванням обов'язків по цих подіях). За допомогою кінцевого автомата може визначатися поведінка індивідуального класу або кооперації класів. Елементами кінцевого автомата є стани, переходи (від стану до стану), події (предмети, що викликає переходи) і дії (реакції на перехід). Як показано на рис. 1.10, стан зображується як закруглений прямокутник, що звичайно включає його ім'я і його під стани (якщо вони є).

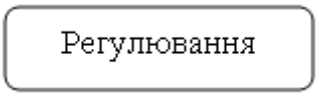


Рисунок 1.10 – Стан

Предмети, що групують, - організаційні частини Uml-моделей. Це скриньки, по яких може бути розкладена модель. Передбачено один різновид предмета, що групує, - пакет.

Пакет - загальний механізм для розподілу елементів по групах. У пакет можуть міститися структурні предмети, предмети поведінки й навіть інші угруповання предметів. На відміну від компонента (який існує в період виконання), пакет - чисто концептуальне поняття. Це означає, що пакет існує тільки в період розробки. Як показано на рис. 1.11, пакет зображується як папка із закладкою, на якій позначено його ім'я й, іноді, його зміст.

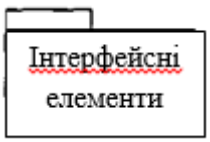


Рисунок 1.11 – Пакет

Предмети, що пояснюють, - частини, що роз'ясняють, Uml-Моделей. Вони є зауваженнями, які можна застосувати для опису, пояснення й коментування будь-

якого елемента моделі. Передбачено один різновид предмета, що пояснює, - примітка.

Примітка - символ для відображення обмежень і зауважень, що приєднуються до елемента або сукупності елементів. Як показано на рис. 1.12, примітка зображується у вигляді прямокутника із загнутим кутом, у який уписується текстовий або графічний коментар.

Рисунок 1.12 – Примітка

Відносини в UML

В UML є наступні різновиди відносин:

1) залежність;
2) асоціація;
3) узагальнення;
4) композиція;
5) агрегація.

Ці відносини є базовими будівельними блоками відносин. Вони використовуються при написанні моделей.

Діаграми в UML

Діаграма - графічне представлення безлічі елементів, найбільше часто зображується як зв'язний граф з вершин (предметів) і дуг (відносин). Діаграми рисуються для візуалізації системи з різних точок зору, потім вони відображаються в систему. Звичайно діаграма дає неповне представлення елементів, які становлять систему. Хоча той самий елемент може з'являтися у всіх діаграмах, на практиці він з'являється тільки в деяких діаграмах. Теоретично діаграма може містити будь-яку комбінацію предметів і відносин, на практиці обмежуються малою кількістю комбінацій, які відповідають п'яти представленням архітектури ПС.

UML включає дев'ять видів діаграм:

1) діаграми класів;
2) діаграми об'єктів;
3) діаграми Use Case (діаграми прецедентів);
4) діаграми послідовності;
5) діаграми співробітництва (кооперації);
6) діаграми схем станів;
7) діаграми діяльності;
8) компонентні діаграми;
9) діаграми розміщення (розгортання).

Діаграма класів показує набір класів, інтерфейсів, співробітництв і їх відносин. При моделюванні об'єктно-орієнтованих систем діаграми класів використовуються найбільше часто. Діаграми класів забезпечують статична проектна вистава системи. Діаграми класів, що включають активні класи, забезпечують статична вистава процесів системи.

Діаграма об'єктів показує набір об'єктів і їх відносини. Діаграма об'єктів представляє статичний "моментальний знімок" з екземплярів предметів, які перебувають у діаграмах класів. Як і діаграми класів, ці діаграми забезпечують

статична проектна вистава або статична вистава процесів системи (але з погляду реальних або фототипових випадків).

Діаграма Use Case (діаграма прецедентів) показує набір елементів Use Case, акторів і їх відносин. За допомогою діаграм Use Case для системи створюється статична вистава Use Case. Ці діаграми особливо важливі при організації й моделюванні поведінки системи, завданні вимог замовника до системи.

Діаграми послідовності й діаграми співробітництва - це різновиди діаграм взаємодії.

Діаграма взаємодії показує взаємодію, що включає набір об'єктів і їх відносин, а повідомлення, що також пересилаються між об'єктами. Діаграми взаємодії забезпечують динамічна вистава системи.

Діаграма послідовності - це діаграма взаємодії, яка виділяє впорядкування повідомлень за часом.

Діаграма співробітництва (діаграма кооперації) - це діаграма взаємодії, яка виділяє структурну організацію об'єктів, що посилають повідомлення, що й ухвалюють. Діаграми послідовності й діаграми співробітництва ізоморфні, що означає, що одну діаграму можна трансформувати в іншу діаграму.

Діаграма схем станів показує кінцевий автомат, представляє стани, переходи, події й дії. Діаграми схем станів забезпечують динамічна вистава системи. Вони особливо важливі при моделюванні поведінки інтерфейсу, класу або співробітництва. Ці діаграми виділяють така поведінка об'єкта, яка управляється подіями, що особливо корисно при моделюванні реактивних систем.

Діаграма діяльності - спеціальний різновид діаграми схем станів, яка показує потік від дії до дії усередині системи. Діаграми діяльності забезпечують динамічна вистава системи. Вони особливо важливі при моделюванні функціональності системи й виділяють потік керування між об'єктами.

Компонентна діаграма показує організацію набору компонентів і залежності між компонентами. Компонентні діаграми забезпечують статична вистава реалізації системи. Вони пов'язані з діаграмами класів у тому розумінні, що в компонент звичайно відображається один або кілька класів, інтерфейсів або кооперацій.

Діаграма розміщення (діаграма розгортання) показує конфігурацію обробних вузлів періоду виконання, а також компоненти, що живуть у них. Діаграми розміщення забезпечують статична вистава розміщення системи. Вони пов'язані з компонентними діаграмами в тому розумінні, що вузол звичайно включає один або кілька компонентів.

Тема: Основи об’єктно-орієнтованого представлення програмних систем. Об’єкти. Загальна характеристика.

Мета: Розглянути поняття декомпозиції. Ознайомитись з принципами об’єктно-орієнтованої декомпозиції: абстрагування, інкапсуляція, модульність, ієрархічна організація. Розглянути "is a" та "part of" – ієрархію. Ознайомитись з поняттям об’єкта. Розглянути поняття індивідуальність, стан та поведінка об’єкта. Розглянути види відношень, що можливі між об’єктами.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Декомпозиція;
2. Принципи об’єктно-орієнтованої декомпозиції:
   a. Абстрагування;
   b. Інкапсуляція;
   c. Модульність;
   d. Ієрархічна організація.
      i. структура із класів ("is a"-ієрархія);
      ii. структура з об'єктів ("part of"-ієрархія).
3. Загальна характеристика об’єктів:
   a. Індивідуальність, стан, поведінка;
   b. Види операцій;
   c. Ролі та обов’язки;
4. Види відношень між об’єктами:
   a. Зв'язок;
   b. Агрегація.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Що таке декомпозиція?
2. В чому різниця між алгоритмічною й об'єктно-орієнтованою декомпозицією?
3. На яких принципах базується об’єктно-орієнтована декомпозиція?
4. Що таке принцип абстрагування?
5. Що таке інкапсуляція?
6. Що таке модульність?
7. Для чого використовується ієрархічна організація?
8. Що таке спадкування?
9. Що таке агрегація?
10. Що таке об’єкт?
11. Що таке стан об’єкта?
12. Що характеризує поведінка об’єкта?

13. Які бувають види операцій об’єкта?
14. Що визначає протокол об’єкту?
15. Що таке роль?
16. В чому різниці між активним та пасивним об’єктом?
17. Що таке зв'язок?
18. В чому відмінність між зв’язком та агрегацією?
19. В чому різниця між агрегацією по величині та агрегацією по посиланню?

Розгляд будь-якої складної системи вимагає застосування техніки *декомпозиції* - розбивки на складові елементи. Відомі дві схеми декомпозиції: алгоритмічна декомпозиція й об'єктно-орієнтована декомпозиція.

В основі *алгоритмічної декомпозиції* лежить розбивку по діях - алгоритмам. Ця схема вистави застосовується у звичайних ПС.

Об'єктно-орієнтована декомпозиція забезпечує розбивку по автономних особах - об'єктам реального (або віртуального) миру. Ці особи (об'єкти) - більш "великі" елементи, кожний з них несе в собі й опису дій, і опису даних.

Об'єктно-орієнтоване представлення ПС ґрунтується на принципах абстрагування, інкапсуляції, модульності й ієрархічної організації. Кожний із цих принципів не новий, але їх спільне застосування розраховане на проведення об'єктно-орієнтованої декомпозиції. Це визначає модифікацію їх змісту й механізмів взаємодії один з одним.

Абстрагування

Апарат абстракції - зручний інструмент для боротьби зі складністю реальних систем. Створюючи поняття в інтересах якого-небудь завдання, ми відволікаємося (абстрагуємося) від несуттєвих характеристик конкретних об'єктів, визначаючи тільки істотні характеристики. Наприклад, в абстракції "годинник" ми виділяємо характеристику "показувати час", відволікаючись від таких характеристик конкретних годин, як форма, колір, матеріал, ціна, виготовлювач.

Отже, абстрагування зводиться до формування абстракцій. Кожна абстракція фіксує основні характеристики об'єкта, які відрізняють його від інших видів об'єктів і забезпечують ясні понятійні границі.

Абстракція зосереджує увагу на зовнішній виставі об'єкта, дозволяє відокремити основне в поведінці об'єкта від його реалізації. Абстракцію зручно будувати шляхом виділення обов'язків об'єкта.

Приклад: фізичний об'єкт - датчик швидкості, встановлюваний на борті літального апарата (ЛА). Сформулюємо обов'язки датчика:

- знати проекцію швидкості ЛА в заданому напрямку;
- показувати поточну швидкість;
- підвергатися настроюванню.

Інкапсуляція

Інкапсуляція й абстракція - взаємодоповнюючі поняття: абстракція виділяє зовнішню поведінку об'єкта, а інкапсуляція містить і приховує реалізацію, яка забезпечує ця поведінка. Інкапсуляція досягається за допомогою інформаційної закритості. Звичайно ховаються структура об'єктів і реалізація їх методів.

Інкапсуляція є процесом поділу елементів абстракції на секції з різною видимістю. Інкапсуляція служить для відділення інтерфейсу абстракції від її реалізації.

Модульність

У мовах C++, Object Pascal, Ada 95 абстракції класів і об'єктів формують логічну структуру системи. При виробництві фізичної структури ці абстракції містяться в модулі. У більших системах, де класів сотні, модулі допомагають управляти складністю. Модулі служать фізичними контейнерами, у яких оголошуються класи й об'єкти логічної розробки.

Модульність визначає здатність системи зазнати декомпозиції на ряд сильно зв'язаних і слабко зчеплених модулів.

Загальна мета декомпозиції на модулі: зменшення строків розробки й вартості ПС за рахунок виділення модулів, які проектують і змінюються незалежно. Кожна модульна структура повинна бути досить простій, щоб бути повністю понятий. Зміна реалізації модулів повинне проводитися без знання реалізації інших модулів і без впливу на їхню поведінку.

Визначення класів і об'єктів виконується в ході логічної розробки, а визначення модулів - у ході фізичної розробки системи. Ці дії сильно взаємозалежні, здійснюються інтерактивно.

В Ada 95 потужним засобом забезпечення модульності є пакет.

Ієрархічна організація

Ми розглянули три механізми для боротьби зі складністю:

1. абстракцію (вона спрощує виставу фізичного об'єкта);
2. інкапсуляцію (закриває деталі внутрішньої вистави абстракцій);
3. модульність (дає шлях угруповання логічно зв'язаних абстракцій).

Прекрасним доповненням до цих механізмів є ієрархічна організація - формування з абстракцій ієрархічної структури. Визначенням ієрархії в проекті спрощуються розуміння проблем замовника і їх реалізація - складна система стає доступною для огляду людиною.

Ієрархічна організація задає розміщення абстракцій на різних рівнях опису системи.

Двома важливими інструментами ієрархічної організації в об'єктно-орієнтованих системах є:

1. структура із класів («is a»-ієрархія);
2. структура з об'єктів («part of»-ієрархія).

Найчастіше *«is a»-ієрархічна* структура будується за допомогою спадкування. *Спадкування* визначає відношення між класами, де клас розділяє структуру або поведінку, певні в одному іншому (одиничне спадкування) або в декілька інших (множинне спадкування) класах.

Суперклас відповідає загальній абстракції, а підклас - спеціалізованої абстракції, у якій елементи суперкласу доповнюються, змінюються й навіть ховаються. Тому спадкування часте називають відношенням *узагальнення-спеціалізація.*

Інший різновид ієрархічної організації – *«part of» -ієрархічна структура* - базується на відношенні агрегації. *Агрегація* не є поняттям, унікальним для об'єктно-орієнтованих систем. Наприклад, будь-яка мова програмування, що дозволяє структури типу «запис», підтримує агрегацію. І все-таки агрегація особливо корисна в комбінації зі спадкуванням:

1) агрегація забезпечує фізичне угруповання логічно зв'язаної структури;

2) спадкування дозволяє легко й багаторазово використовувати ці загальні групи в інших абстракціях.

Цікаво зрівняти елементи ієрархій спадкування й агрегації з погляду рівня складності. При спадкуванні нижній елемент ієрархії (підклас) має більший рівень складності (більші можливості), при агрегації - навпаки.

Розглянемо більш пильно об'єкти - конкретні сутності, які існують у часі й просторі.

Загальна характеристика об'єктів

Об'єкт – це конкретне представлення абстракції. Об'єкт має індивідуальність, стан і поведінку. Структура й поведінка подібних об'єктів визначені в їхньому загальному класі. Терміни "екземпляр класу" і "об'єкт" взаємозамінні. На рис. 2.1 наведений приклад об'єкта по імені Стілець, що має певний набір властивостей і операцій.

Індивідуальність - це характеристика об'єкта, яка відрізняє його від усіх інших об'єктів.

Стан об'єкта характеризується переліком усіх властивостей об'єкта й поточними значеннями кожного із цих властивостей.

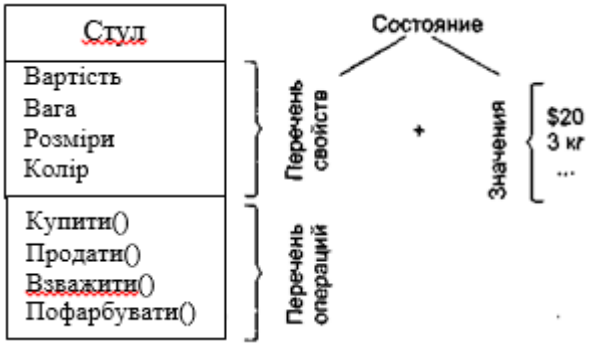


Рисунок 2.1 – Представлення об’єкта «Стул»

Об'єкти не існують ізольовано одне від одного. Вони зазнають впливу або самі впливають на інші об'єкти.

Поведінка характеризує те, як об'єкт впливає на інші об'єкти (або зазнає впливу) у термінах змін його стану й передачі повідомлень. Поведінка об'єкта є функцією як його стану, так і виконуваних їм операцій (Купити, Продати, Зважити, Перемістити, Пофарбувати). Говорять, що стан об'єкта представляє сумарний результат його поведінки.

Операція позначає обслуговування, яке об'єкт пропонує своїм клієнтам. Можливі п'ять видів операцій клієнта над об'єктом:

1) модифікатор (змінює стан об'єкта);
2) селектор (дає доступ до стану, але не змінює його);
3) ітератор (доступ до змісту об'єкта вроздріб, у строго певному порядку);
4) конструктор (створює об'єкт і ініціює його стан);
5) деструктор (руйнує об'єкт і звільняє займану їм пам'ять).

Таблиця 2.1 – Різновиди операцій

Вид операції	Приклад операції
Модифікатор	Поповнити (кг)
Селектор	Яка вага ( ) : integer
Ітератор	Показати асортимент товарів ( ) : string
Конструктор	Створити робота (параметри)
Деструктор	Зруйнувати роботу ( )

У чистих об'єктно-орієнтованих мовах програмування операції можуть оголошуватися тільки як методи - елементи класів, екземплярами яких є об'єкти. Гібридні мови (C++, Ada 95) дозволяють писати операції як вільні підпрограми (поза класами). Відповідні приклади показані на рис. 2.2.

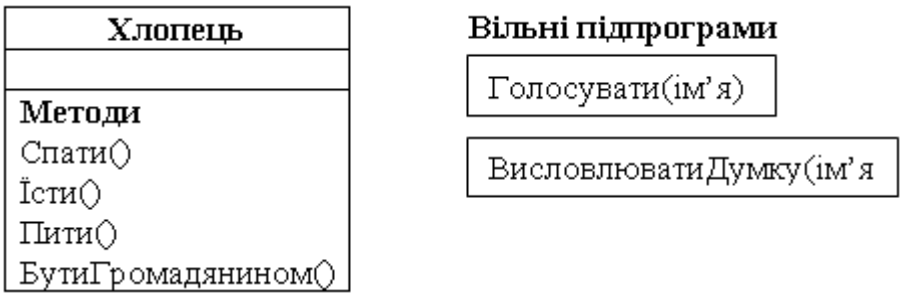


Рисунок 2.2 – Методи та вільні підпрограми

У загальному випадку всі методи й вільні підпрограми, асоційовані з конкретним об'єктом, утворюють його *протокол*. Таким чином, протокол визначає оболонку припустимої поведінки об'єкта й тому містить у собі цільне (статичне й динамічне) представлення об'єкта.

Великий протокол корисно розділяти на логічні угруповання поведінки. Ці угруповання, що розділяють простір поведінки об'єкта, позначають *ролі*, які може відіграти об'єкт. Принцип виділення ролей ілюструє рис. 2.3.

З погляду зовнішнього середовища важливе значення має таке поняття, як обов'язки об'єкта. *Обов'язки* означають зобов'язання об'єкта забезпечити певну поведінку. Обов'язками об'єкта є всі види обслуговування, які він пропонує клієнтам. У світі об'єкт відіграє певні ролі, виконуючи свої обов'язки.

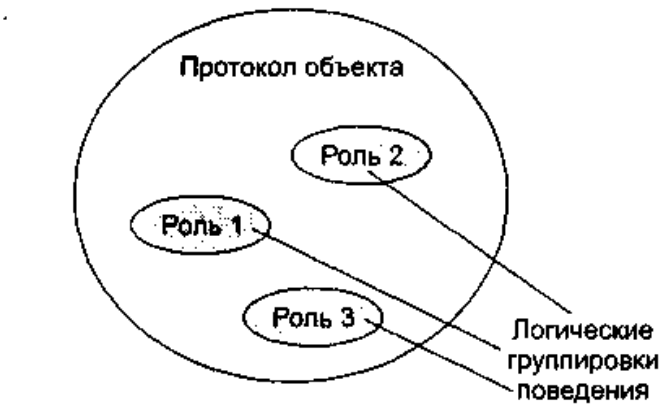


Рисунок 2.3 – Простір поведінки об'єкта

Наявність в об'єкта внутрішнього стану означає, що порядок виконання їм операцій дуже важливий. Інакше кажучи, об'єкт може представлятися як

незалежний автомат. За аналогією з автоматами можна виділяти активні й пасивні об'єкти.

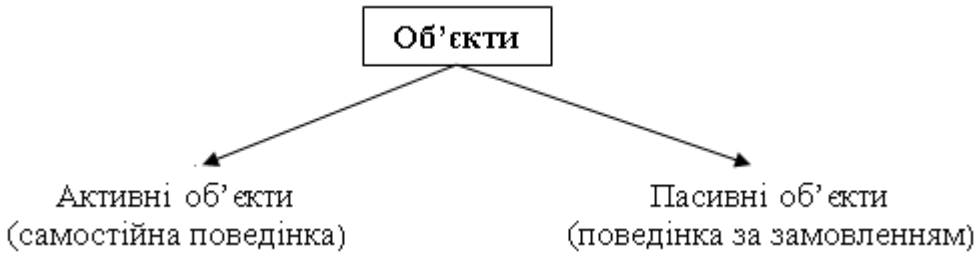


Рисунок 2.4 – Активні та пасивні об’єкти

Активний об'єкт має власний канал (потік) керування, пасивний - немає. Активний об'єкт автономний, він може проявляти свою поведінку без впливу з боку інших об'єктів. Пасивний об'єкт, навпаки, може змінювати свій стан тільки під впливом інших об'єктів.

У поле зору розроблювача ПЗ перебувають не об'єкти-одинаки, а взаємодіючі об'єкти, адже саме взаємодія об'єктів реалізує поведінка системи. У Г. Буча є відмінна цитата з Галла: "Літак - це набір елементів, кожний з яких по своїй природі прагне впасти на землю, але ціною спільних безперервних зусиль долає цю тенденцію". Відносини між парою об'єктів ґрунтуються на взаємній інформації про дозволені операції й очікуваній поведінці. Особливо цікаві два види відносин між об'єктами: зв'язки й агрегація.

Зв'язки

Зв'язок - це фізичне або понятійне з'єднання між об'єктами. Об'єкт співробітничає з іншими об'єктами через з'єднуючі їхні зв'язки. Зв'язок позначає з'єднання, за допомогою якого:

1. об'єкт-клієнт викликає операції об'єкта-постачальника;
2. один об'єкт переміщає дані до іншого об'єкта.

Можна сказати, що зв'язки є рейками між станціями-об'єктами, по яких їздять "трамвайчики повідомлень".

Зв'язки між об'єктами показані на рис. 2.5 за допомогою сполучних ліній. Зв'язки представляють можливі шляхи для передачі повідомлень. Самі повідомлення показані стрілками, що відзначають їхнього напрямку, і позначені іменами викликуваних операцій.

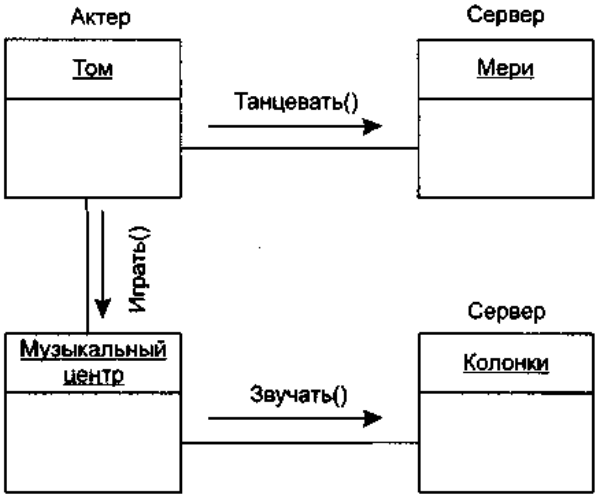


Рисунок 2.5 – Зв’язки між об’єктами

Як учасник зв'язку об'єкт може відіграти одну із трьох ролей:

1. актор - об'єкт, який може впливати на інші об'єкти, але ніколи не підданий впливу інших об'єктів;
2. сервер - об'єкт, який ніколи не впливає на інші об'єкти, він тільки використовується іншими об'єктами;
3. агент - об'єкт, який може як впливати на інші об'єкти, так і використовуватися ними. Агент створюється для виконання роботи від імені актора або іншого агента.

На рис. 2.5 Том - це актор, Мері, Стовпчики - сервери, Музичний центр - агент.

Агрегація

Зв'язки позначають рівноправні (клієнт-серверні) відносини між об'єктами. *Агрегація* позначає відносини об'єктів в ієрархії "ціле/частина". Агрегація забезпечує можливість переміщення від цілого (агрегату) до його частин (властивостям).

Агрегація може позначати, а може й не позначати фізичне включення частини в ціле. На рис. 2.6 наведений приклад фізичного включення (композиції) частин (Двигуна, Сидінь, Коліс) в агрегат Автомобіль. У цьому випадку говорять, що частини включені в агрегат по величині.

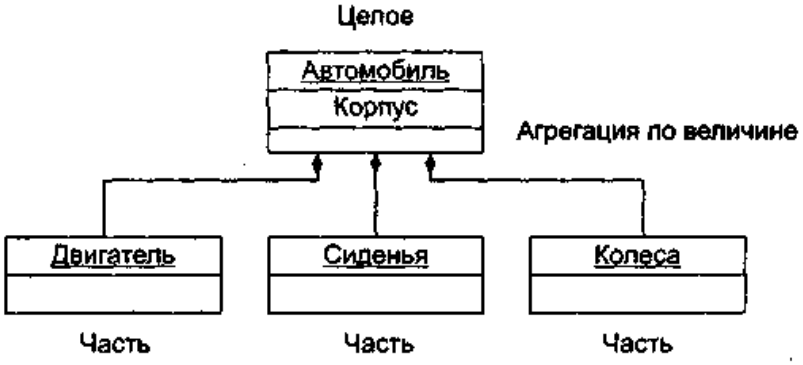


Рисунок 2.6 – Фізичне включення частин в агрегат

На рис. 2.7 наведений приклад нефізичного включення частин (Студента, Викладача) в агрегат ВУЗ. Очевидно, що Студент і Викладач є елементами ВУЗу,

але вони не входять у нього фізично. У цьому випадку говорять, що частини включені в агрегат по посиланню.

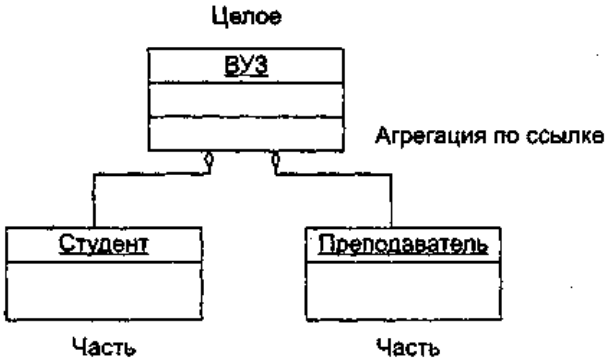


Рисунок 2.7 – Не фізичне включення частин до агрегату

При виборі виду відносини повинні враховуватися наступні фактори:

1. зв'язки забезпечують низьке зчеплення між об'єктами;
2. агрегація інкапсулює частини як секрети цілого.

Тема: Діаграма прецедентів (діаграма варіантів використання або Use case Diagram).

Мета: Розглянути основні поняття діаграми Use Case. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   1. Готовність групи до заняття;
   2. Психоемоційний настрій;
   3. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   1. Повідомлення теми та мети заняття;
   2. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграма варіантів використання
2. Актори
3. Варіанти використання
4. Інтерфейси
5. Примітки
6. Зв’язки між елементами діаграми варіантів використання:
   - відношення асоціації
   - відношення узагальнення
   - відношення включення
   - відношення розширення

IV. Узагальнення та систематизація знань.

IV. Підведення підсумків заняття.

VI. Домашнє завдання:

1. Ознайомитись з теоретичними відомостями лекції
2. Вивчити основні поняття лекції.
3. Дати відповіді на контрольні питання.

Контрольні питання:

1. Для чого призначена діаграма варіантів використання?
2. З яких елементів складається діаграма Use Case?
3. Що таке «актор» в термінології UML? Як позначається «актор» на діаграмі варіантів використання?
4. Що на діаграмі варіантів використання зображується еліпсом?
5. Перелічіть відношення, що можуть використовуватися на діаграмі Use Case?
6. Чим відрізняються друг від друга відношення включення й розширення з погляду керування?
7. Які відношення можуть використовуватися для поєднання тільки акторів, акторів та варіантів використання, тільки варіантів використання?

Проектування - один із важливих кроків під час розробки програми, який дуже часто ігнорується розробниками-початківцями. Зазвичай вони намагаються утримати все в голові або, у кращому разі, записати деякі важливі відомості на аркуші паперу. Як результат, у них немає чіткого плану подальших дій, і проект може бути відкладений у довгий ящик.

Зазвичай під час проектування розробники зображують систему графічно, оскільки людині легко розібратися в такому поданні. Саме тому замість написання громіздких текстів про кожну можливість майбутньої програми розробники будують різні діаграми для опису своїх систем. Це допомагає їм не забувати, що потрібно реалізувати в програмі, і швидко вводити в курс справи своїх колег.

Сьогодні ми розберемося з тим, як використовувати діаграму варіантів використання UML (англ. "Unified Modeling Language") - стандартизована мова моделювання під час проектування програм.

Коли розробник створює свій додаток, він насамперед замислюється над двома питаннями:

- *Що робитиме додаток?*
- *Хто буде користуватися цим додатком?*

Деякими програмами може користуватися безліч людей, тому часто необхідно виділяти різні групи користувачів системи. У кожної такої групи можуть бути свої права і можливості в системі.

Для того щоб описати різні групи користувачів та їхні можливості в майбутній програмі, створюється так звана діаграма варіантів використання.

Діаграма варіантів використання (англ. use-case diagram) - діаграма, що описує, який функціонал розроблюваної програмної системи доступний кожній групі користувачів.

На сьогоднішній парі ми розберемо елементи цієї діаграми, які найчастіше застосовуються під час побудови, на безлічі невеликих прикладів діаграм і на прикладі однієї великої діаграми. Ця велика діаграма буде використовуватися під час проектування якоїсь програмної системи. В якості прикладу такої системи

виберемо інформаційну систему для коледжу (можна розглядати її як сайт або як окремий додаток).

У цій системі можна виділити такі групи користувачів:

- Студенти
- Викладачі
- Класні керівники
- Заступники директора

Кожна з груп користувачів може користуватися нашою системою по-своєму.

Студенти можуть:

- Дивитися розклад
- Переглядати свої оцінки

Викладачі можуть:

- Розміщувати матеріали для пар зі своїх дисциплін
- Виставляти оцінки в електронний журнал

Класні керівники можуть робити все те ж саме, що і викладачі плюс:

- Складати розклад батьківських зборів

Заступники директора можуть:

- Складати розклад
- Публікувати пости з важливою інформацією

Крім того, у системи є функціонал, який доступний усім групам користувачів. У розроблюваній системі актуально буде додати месенджер, у якому можна буде швидко зв'язуватися з людиною, що цікавить. Отже, ця функціональність має бути доступна кожному користувачеві. Тобто всі користувачі можуть:

- Надсилати повідомлення

Вийшло багато пунктів, які може бути складно укласти в голові. Для того щоб швидко орієнтуватися в цих пунктах, треба навчитися будувати діаграми варіантів використання.

А чому ми описуємо так мало можливостей?

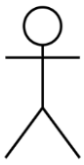
Зауважте, що на діаграмі треба відображати тільки ключовий функціонал системи. Наприклад, дії "увійти в систему", "вийти з системи" або "відновити пароль" можуть бути присутніми в будь-якій системі, і їхню наявність не варто додатково описувати, оскільки це забруднює діаграму несуттєвими елементами.

Взагалі додавання деяких дій на діаграмі залежить від глибини деталізації. Якщо вам усе ж таки потрібно зобразити деякі стандартні дії, ніщо не завадить швидко це зробити.

А тепер, коли були виділені групи користувачів і функціональність системи, можна починати будувати діаграму, щоб зафіксувати та структурувати отримані дані.

Побудова діаграми

Кожна група користувачів на діаграмі варіантів використання позначається чоловічком, під яким записується ім'я групи людей, яку він позначає. Давайте зобразимо групу користувачів "Викладачі":



Викладач

Цей чоловічок позначає всіх викладачів, які користуватимуться системою.

Зверніть увагу, що ім'я групи записується в однині. Символ чоловічка вже позначає групу користувачів, тому не потрібно додатково відображати це в імені.

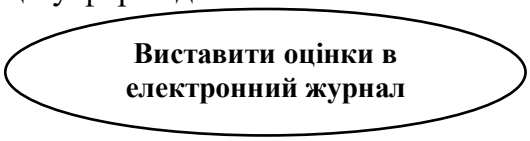
У термінології UML, цей чоловічок називається **актором** (англ. "actor"). У загальному випадку, актор позначає будь-які сутності, що використовують систему. Цими сутностями можуть бути люди, технічні пристрої або навіть інші системи.

Так само зобразимо акторів для решти груп користувачів:

Заст. директора **Класний керівник** **Викладач** **Студент**

На рисунку зображено всі групи користувачів, які можуть користуватися нашою системою

Кожна група користувачів використовує певні функції системи. На діаграмі варіантів використання функція системи зображується еліпсом, усередині якого записується ім'я функції у формі дієслова з пояснювальними словами.



Цей еліпс представляє дію «Виставити оцінки в електронний журнал»

У термінології UML, цей еліпс називається варіантом використання (англ. «use-case»). У загальному випадку, варіант використання - набір дій, який може бути використаний актором для взаємодії з системою.

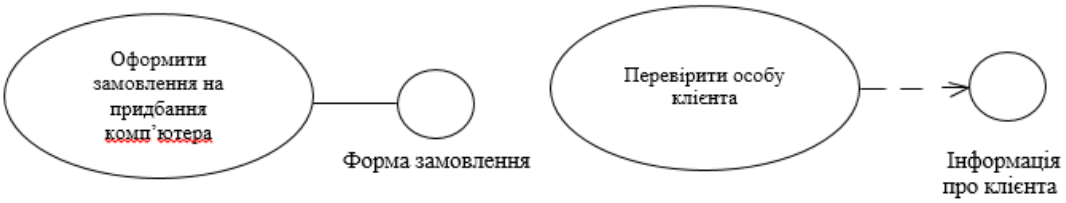
Інтерфейси

Інтерфейс (interface) служить для специфікації параметрів моделі, які видимі ззовні без вказівки їх внутрішньої структури. У мові UML інтерфейс є класифікатором і характеризує тільки обмежену частину поведінки сутності, яка моделюється. Стосовно до діаграм варіантів використання, інтерфейси визначають сукупність операцій, які забезпечують необхідний набір сервісів або функціональності для акторів. Інтерфейси не можуть містити ні атрибутів, ні станів, ні спрямованих асоціацій. Вони містять тільки операції без вказівки особливостей їх реалізації. Формально інтерфейс еквівалентний абстрактному класу без атрибутів і методів з наявністю тільки абстрактних операцій.

На діаграмі варіантів використання інтерфейс зображується у вигляді маленького кола, поруч із яким записується його ім'я. Якщо ім'я записується на англійському, то воно повинне починатися із заголовної букви I, наприклад, IsecurelnformatIon, Isensor.

Графічне зображення інтерфейсів на діаграмах варіантів використання

Графічний символ окремого інтерфейсу може з'єднуватися на діаграмі суцільною лінією з тим варіантом використання, який його підтримує. Суцільна лінія в цьому випадку вказує на той факт, що пов'язаний з інтерфейсом варіант використання повинен реалізовувати всі операції, необхідні для даного інтерфейсу, а можливо й більше. Крім цього, інтерфейси можуть з'єднуватися з варіантами використання пунктирною лінією зі стрілкою, що означає, що варіант використання призначений для специфікації тільки того сервісу, який необхідний для реалізації даного інтерфейсу.

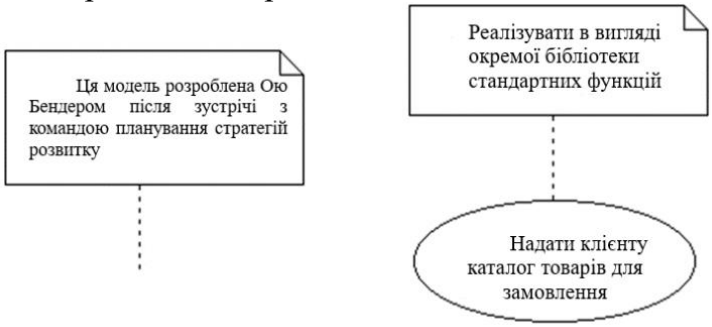


Графічне зображення взаємозв'язків інтерфейсів з варіантами використання

Примітки

Примітки (notes) у мові UML призначені для включення в модель довільної текстової інформації, що має безпосереднє відношення до контексту розроблювального проекту. У якості такої інформації можуть бути коментарі розроблювача (наприклад, дата й версія розробки діаграми або її окремих компонентів), обмеження (наприклад, на значення окремих зв'язків або екземпляри сутностей) і позначені значення. Стосовно до діаграм варіантів використання примітка може носити саму загальну інформацію, що ставиться до загального контексту системи.

Графічно примітки позначаються прямокутником з "загнутим" верхнім правим куточком. Усередині прямокутника втримується текст примітки. Примітка може ставитися до будь-якого елемента діаграми, у цьому випадку їх з'єднує пунктирна лінія. Якщо примітка ставиться до декільком елементам, то від нього проводяться, відповідно, кілька ліній. Зрозуміло, примітки можуть бути присутнім не тільки на діаграмі варіантів використання, але й на інших канонічних діаграмах.

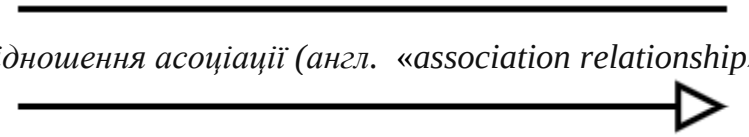


Приклади приміток у мові UML

Зв'язки між елементами

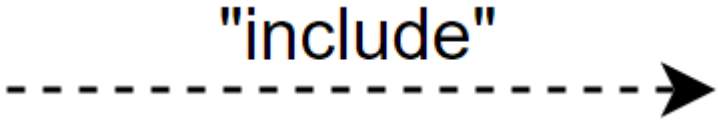
На діаграмах UML для зв'язування елементів використовуються різноманітні сполучні лінії, які називаються відношеннями. Кожне таке відношення має власну назву і використовується для досягнення певної мети.

Всі види відносин, які будуть використовуватися при роботі з діаграмою Use case.

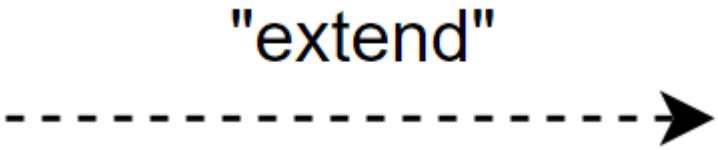


Відношення асоціації (англ. «association relationship»)

Відношення узагальнення (англ. «generalization relationship»)



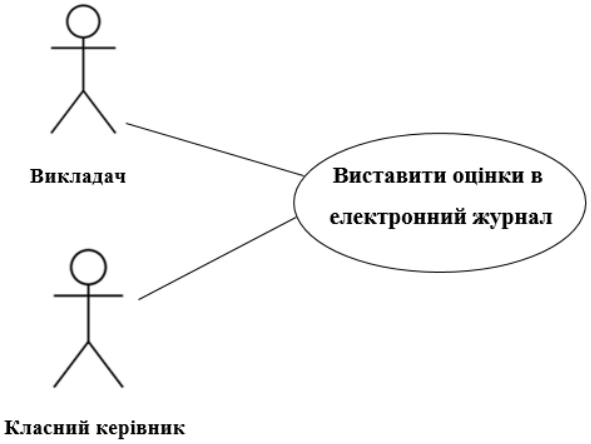
Відношення включення (англ. «include relationship»)



Відношення розширення (англ. «extend relationship»)

Відношення асоціації

Нам потрібно відображати на діаграмі інформацію про те, які варіанти використання можуть бути використані кожним актором. Зараз, наприклад, ми хочемо показати, що виставляти оцінки можуть тільки викладачі.

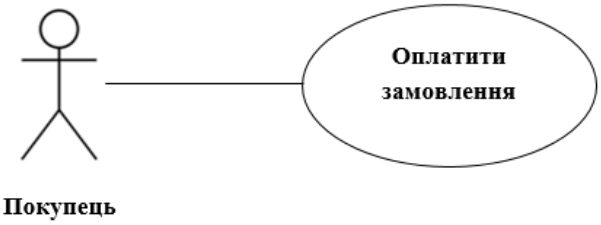


Зображуємо на діаграмі інформацію про те, що викладачі можуть виставляти оцінки

Ми з'єднали акторів із варіантом використання за допомогою суцільної лінії без стрілки. Така лінія називається ***відношенням асоціації***.

Відношення асоціації ("association relationship")

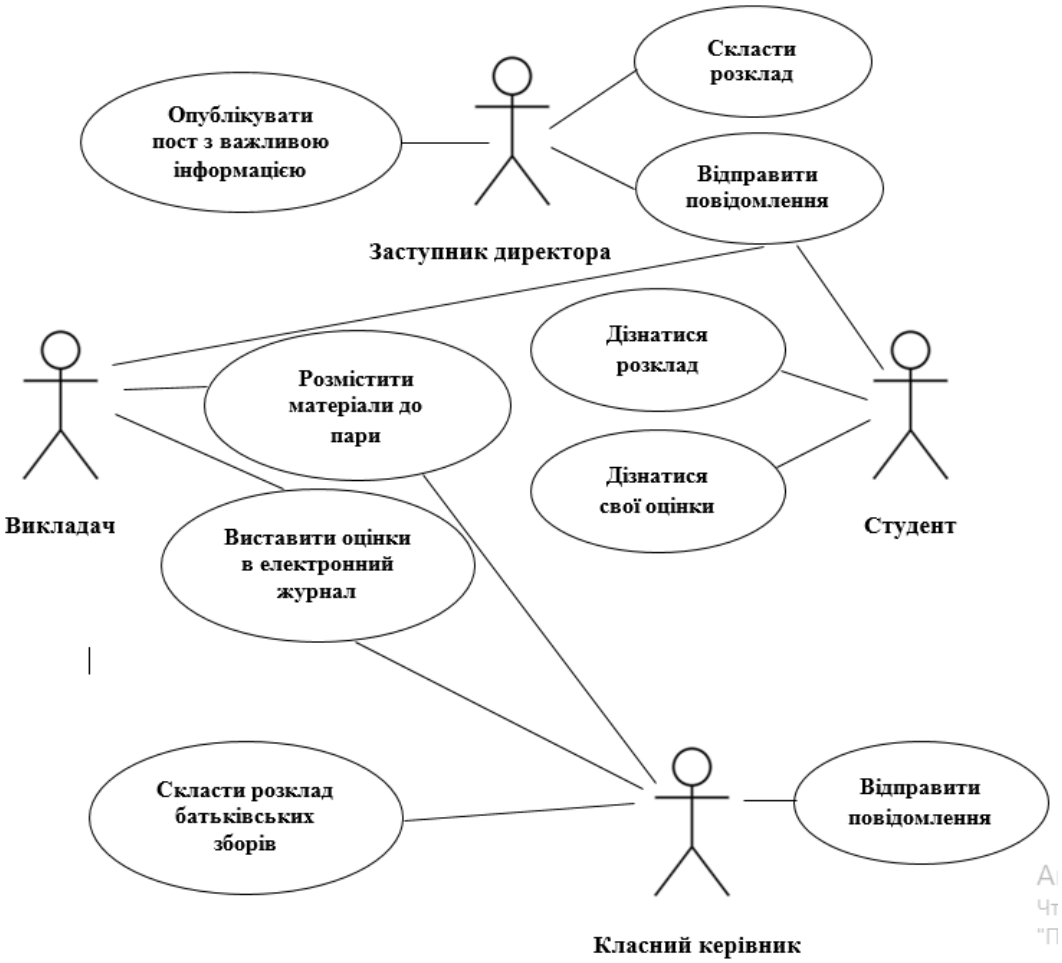
Відношення асоціації призначене **лише** для з'єднання акторів і варіантів використання. Немає жодного сенсу з'єднувати відношенням асоціації двох акторів або два варіанти використання.



Зображуємо на діаграмі можливість покупців оплачувати замовлення

Якщо на діаграмі варіантів використання актор з'єднаний із варіантом використання за допомогою відношення асоціації, це означає, що цей актор може виконувати дії, описані варіантом використання.

Додамо ще варіантів використання і з'єднаємо їх із відповідними акторами:



Перша версія діаграми

Поки що наша діаграма зовсім не вражає, тому продовжимо наповнювати її інформацією. Заодно дізнаємося всі можливості цього виду діаграм.

Відношення узагальнення

Зауважимо, що в нашій системі групи користувачів "Викладач" і "Класний керівник" мають схожі можливості. Щоб зобразити це на діаграмі, можна піти одним із трьох шляхів:

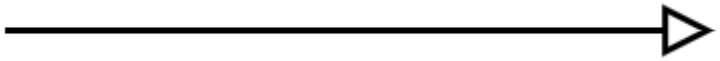
1. Дублювати варіанти використання, щоб пов'язати їх із кожним схожим актором (очевидно, невдалий варіант).

2. З'єднати кожного актора з усіма потрібними варіантами використання. Це може породити безліч перетинів ліній, що не найкращим чином позначиться на читабельності діаграми.

3. Показати за допомогою одного з видів відносин, що актори пов'язані між собою. Це означатиме, що один із них може користуватися всіма варіантами використання, з якими з'єднаний інший актор.

Останній варіант схожий на принцип повторного використання коду під час написання програм або на успадкування класів в об'єктно-орієнтованому програмуванні. Перевага цього варіанта в тому, щоб зменшити кількість зв'язків на діаграмі.

Зрозуміло, ми скористаємося третім шляхом. У цьому нам допоможе, так зване, *відношення узагальнення*. Відношення узагальнення позначається суцільною лінією з порожнистою трикутною стрілкою.

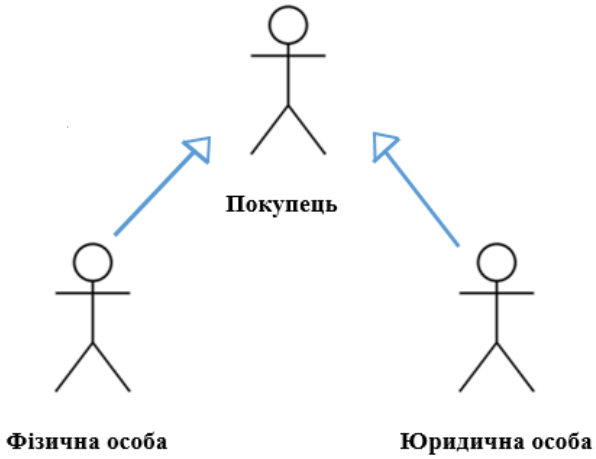


Відношення узагальнення (англ. "generalization relationship")

Відношення узагальнення означає, що деякий актор (варіант використання) може бути узагальнений до іншого актора (варіанту використання). Стрілка спрямована від окремого випадку (спеціалізації) до загального випадку.

Нижче наведено кілька прикладів використання відношення узагальнення.

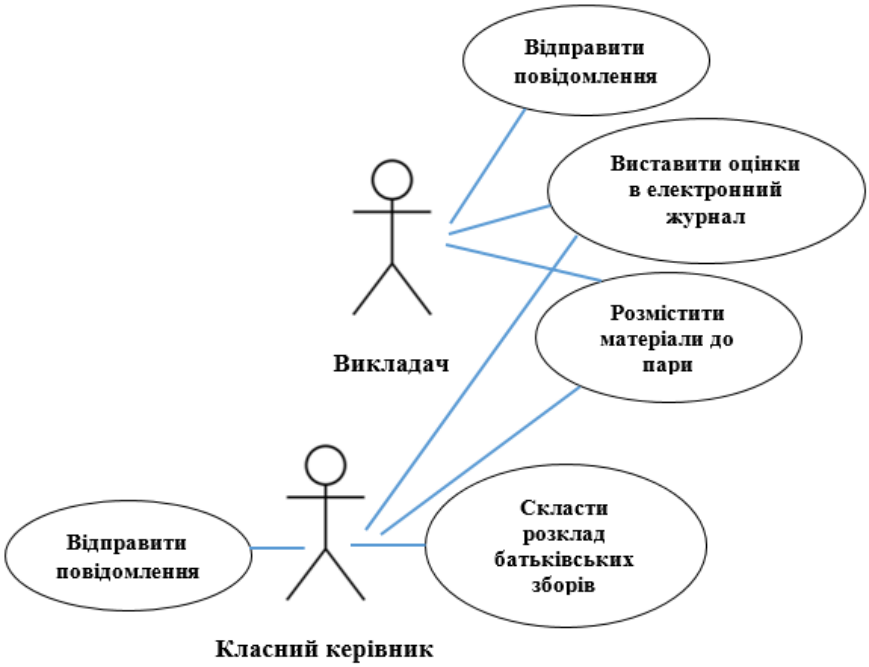
Купівля гірського і швидкісного велосипеда - ЧАСНИЙ випадок купівлі велосипеда



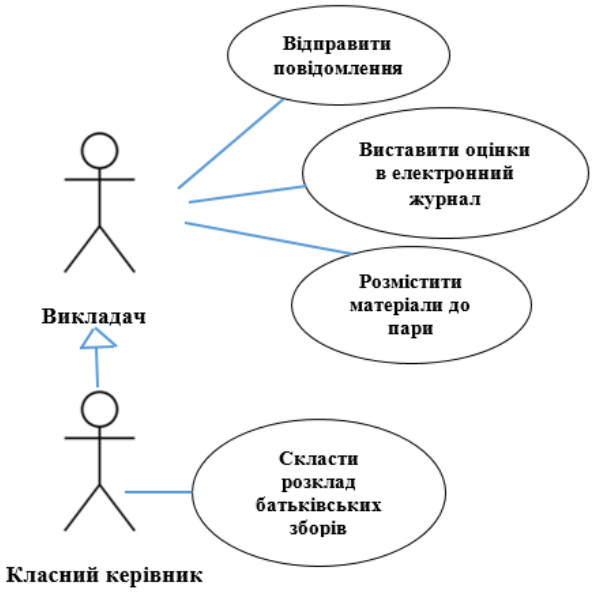
Фізичну особу та юридичну особу можна УЗАГАЛЬНИТИ до звичайного покупця

Як можна помітити, відношення узагальнення використовується, щоб показати, що одна дія є окремим випадком іншої дії або що одну групу людей можна узагальнити до іншої групи.

Повернімося до нашого основного прикладу. Зобразимо відношення узагальнення від актора "Класний керівник" до актора "Викладач".



До використання відношення узагальнення

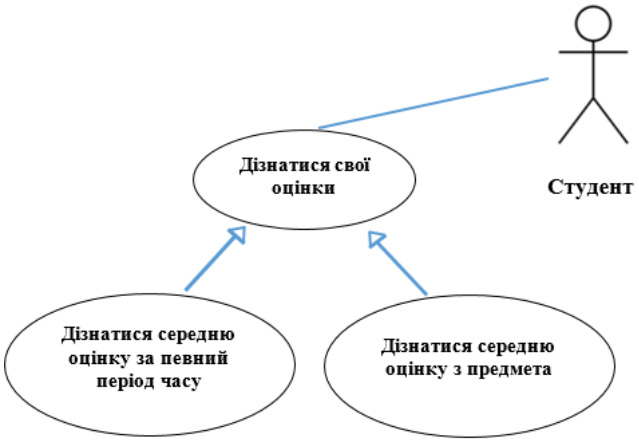


Після додавання відношення узагальнення

На рисунку зверху одразу видно, наскільки зрозумілішою стає діаграма під час використання відношення узагальнення: зникли всі повтори варіантів використання і перетини ліній. Зрозуміло, це величезний плюс для тих, хто читатиме цю діаграму надалі.

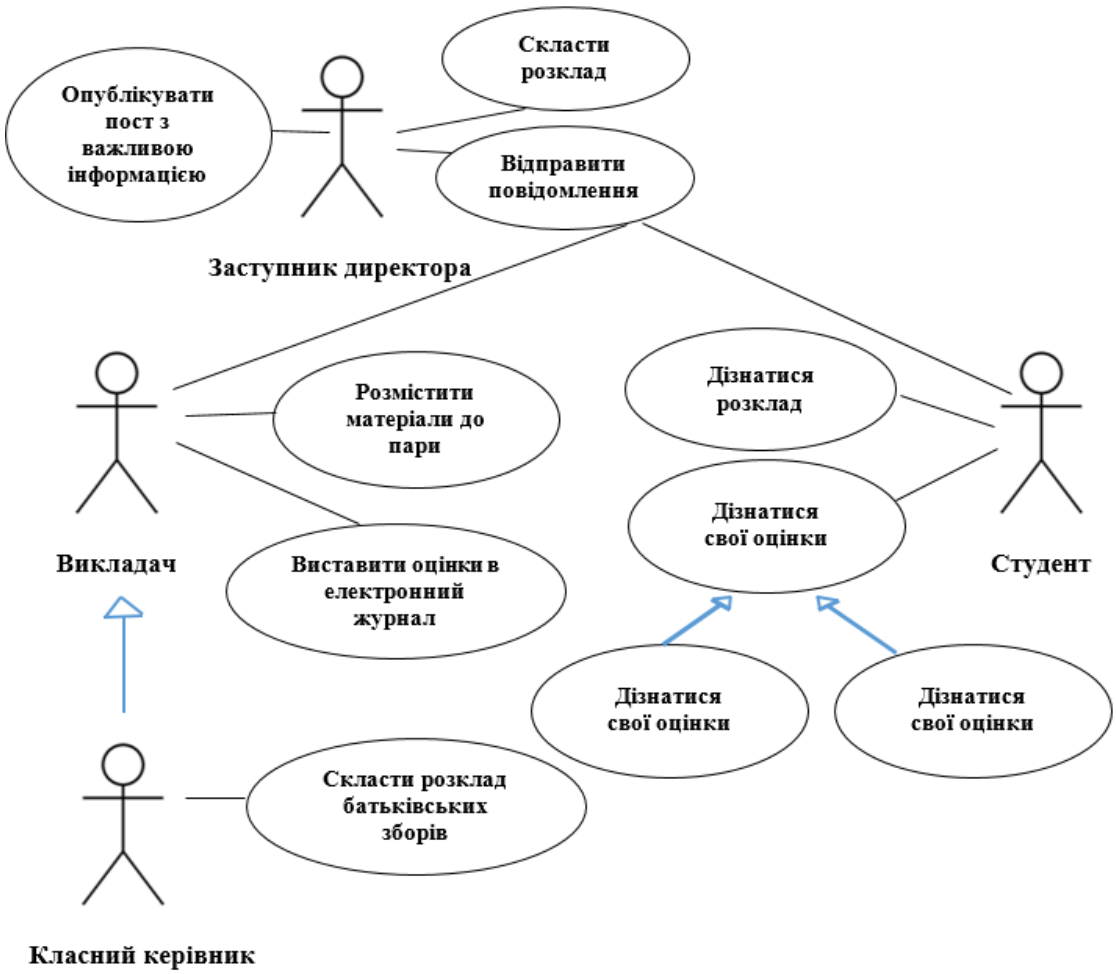
Давайте звернемо увагу на дію «Дізнатися свої оцінки». Логічно припустити, що студенти захочуть не тільки знати список своїх оцінок, а й знати свою середню оцінку за деякий період часу або середню оцінку з певного предмета.

Зобразимо це на діаграмі. Для цього створимо два варіанти використання "Дізнатися середню оцінку за певний період часу" і "Дізнатися середню оцінку з предмета" і з'єднаємо їх із варіантом використання "Дізнатися свої оцінки" відношенням узагальнення.



Уточнюємо на діаграмі, що студенти мають можливість дізнатися середню оцінку за деякий період часу та середній бал із деякого предмета

Приєднаємо це до основної діаграми:



Друга версія діаграми

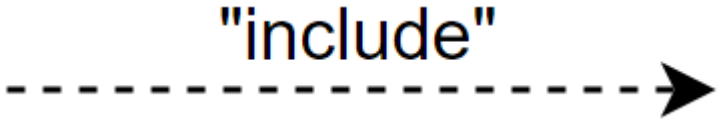
Відношення включення

Для заступника директора ми зазначали, що йому потрібно складати розклади. *Умовно* розклад можна поділити на три категорії:

1. Розклад занять
2. Розклад заходів
3. Розклад канікул

Усе це складається заступником директора, тому покажемо це на діаграмі. Для цього будемо використовувати ***відношення включення***.

Відношення включення позначається пунктирною лінією з V-подібною стрілкою на кінці, над стрілкою додається напис **"include"**.

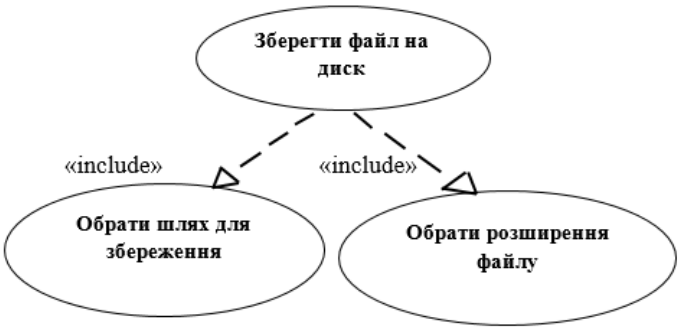


Відношення включення (англ. "include relationship")

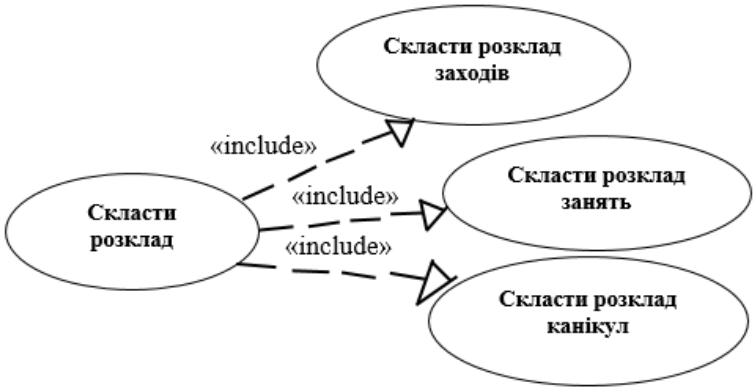
У загальному випадку, відношення включення використовується, щоб показати, що деякий варіант використання включає в себе інший варіант використання як складову частину.

Коли ми використовуємо відношення включення, ми маємо на увазі, що складові варіанти використання ОБОВ'ЯЗКОВО входять до складу загального варіанта використання.

Пояснемо сенс і цього відношення на невеликому прикладі. Коли користувач зберігає результати своєї роботи у файл, він зазначає місце збереження і розширення файлу (наприклад, якщо він редагував фотографію в photoshop, він може зберегти її в різних форматах). Цей процес можна зобразити на діаграмі варіантів використання таким чином:

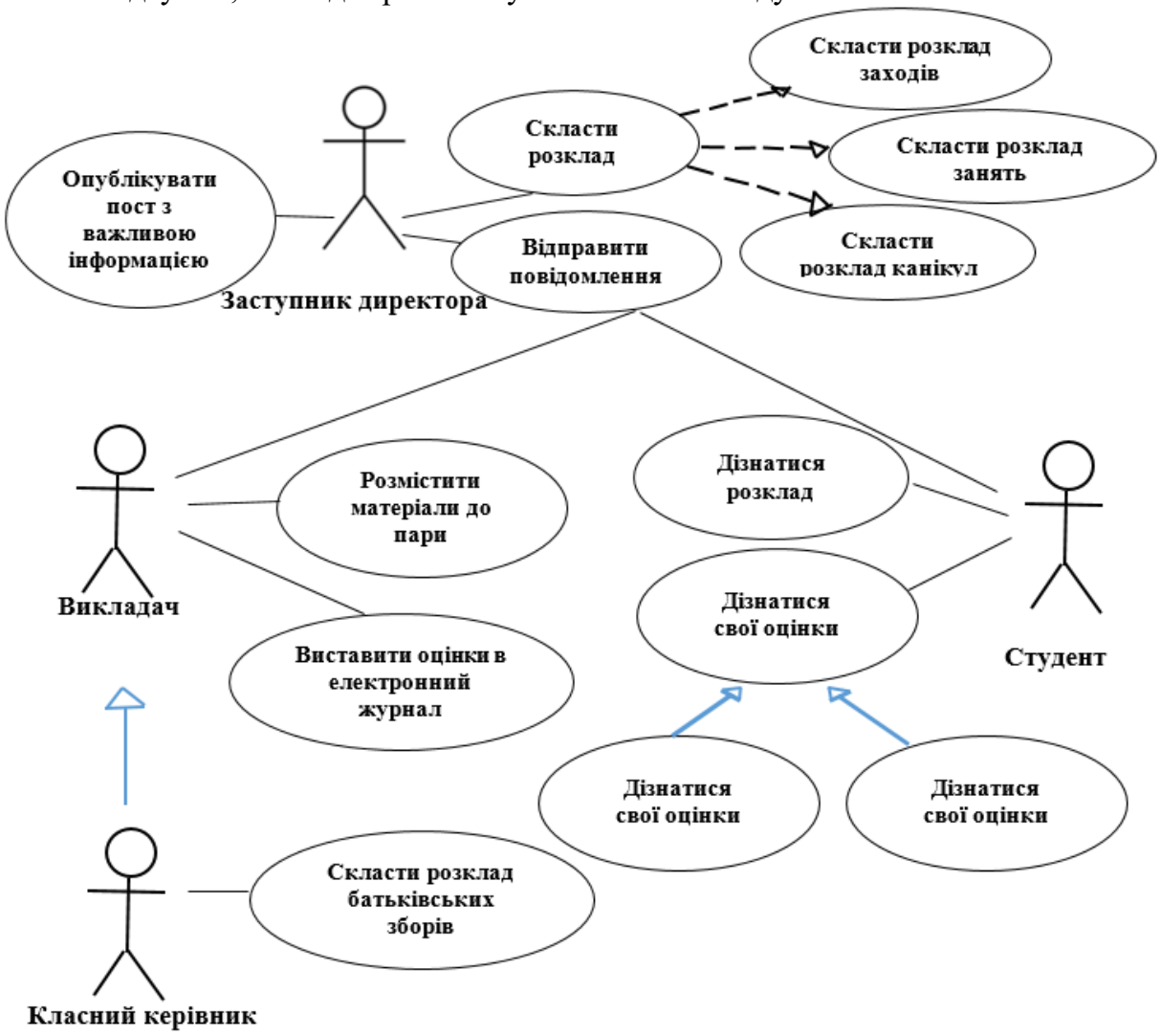


Відношення включення використовується для зображення складеної дії. Знову повернемося до нашого основного прикладу.



Складання розкладу ВКЛЮЧАЄ в себе складання розкладу занять, заходів, канікул (обов'язково).

Як підсумок, наша діаграма набуває такого вигляду:



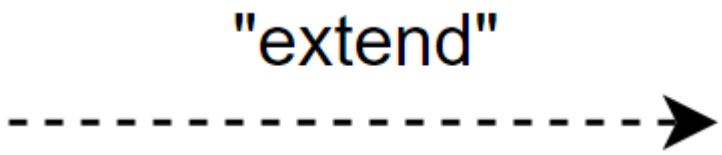
Третя версія діаграми

Загалом, на цьому можна зупинитися. Хоч наш приклад і демонстраційний, він трохи відображає функціональність реального застосунку. Проте залишився ще один елемент, який ми не розглянули.

Відношення розширення

Потрібно сказати, що в діаграмах варіантів використання застосовується ще один вид зв'язку - ***відношення розширення***. Застосування відношення розширення дещо специфічне, оскільки неправильне його використання може заплутати читача діаграми. Проте, для повноти картини ми все одно розглянемо застосування цього відношення на практиці. Востаннє модифікуємо нашу діаграму!

Під час дистанційного навчання студентам необхідно виконувати домашні завдання і надсилати їх у вигляді архіву або фотографій вчителям. Виходить, потрібно додати можливість прикріплювати файл до повідомлення в нашій системі. Щоб відобразити це на діаграмі треба використовувати відношення розширення. Відношення розширення позначається пунктирною лінією з V-подібною стрілкою на кінці (схоже на відношення включення), над стрілкою додається напис "extend ".



Відношення розширення (англ. "extend relationship")

Навіщо над пунктирними лініями додавати написи "include" і "extend"?

Щоб краще зрозуміти цей тип відносин, розглянемо приклад. Припустимо, ви робите замовлення в мережі швидкого харчування. Ви хочете замовити бургер. Вам, найімовірніше, запропонують розширити ваше замовлення картоплею фрі або соусом. Давайте зобразимо процес замовлення на діаграмі варіантів використання.

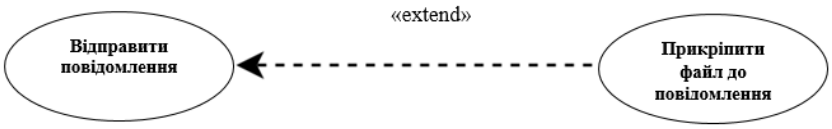
На діаграмі передбачається, що до замовлення МОЖЕ БУТИ додана картопля фрі або соус (необов'язково)

Два нижніх варіанти використання описують можливі "розширення" для базового варіанта використання. Виходячи з цього прикладу, ми можемо зробити важливе зауваження.

Можна сказати, що *відношення розширення* - це вибіркове відношення включення. Якщо відношення включення означає, що елемент обов'язково включається до складу іншого елемента, то у випадку відношення розширення це включення необов'язкове.

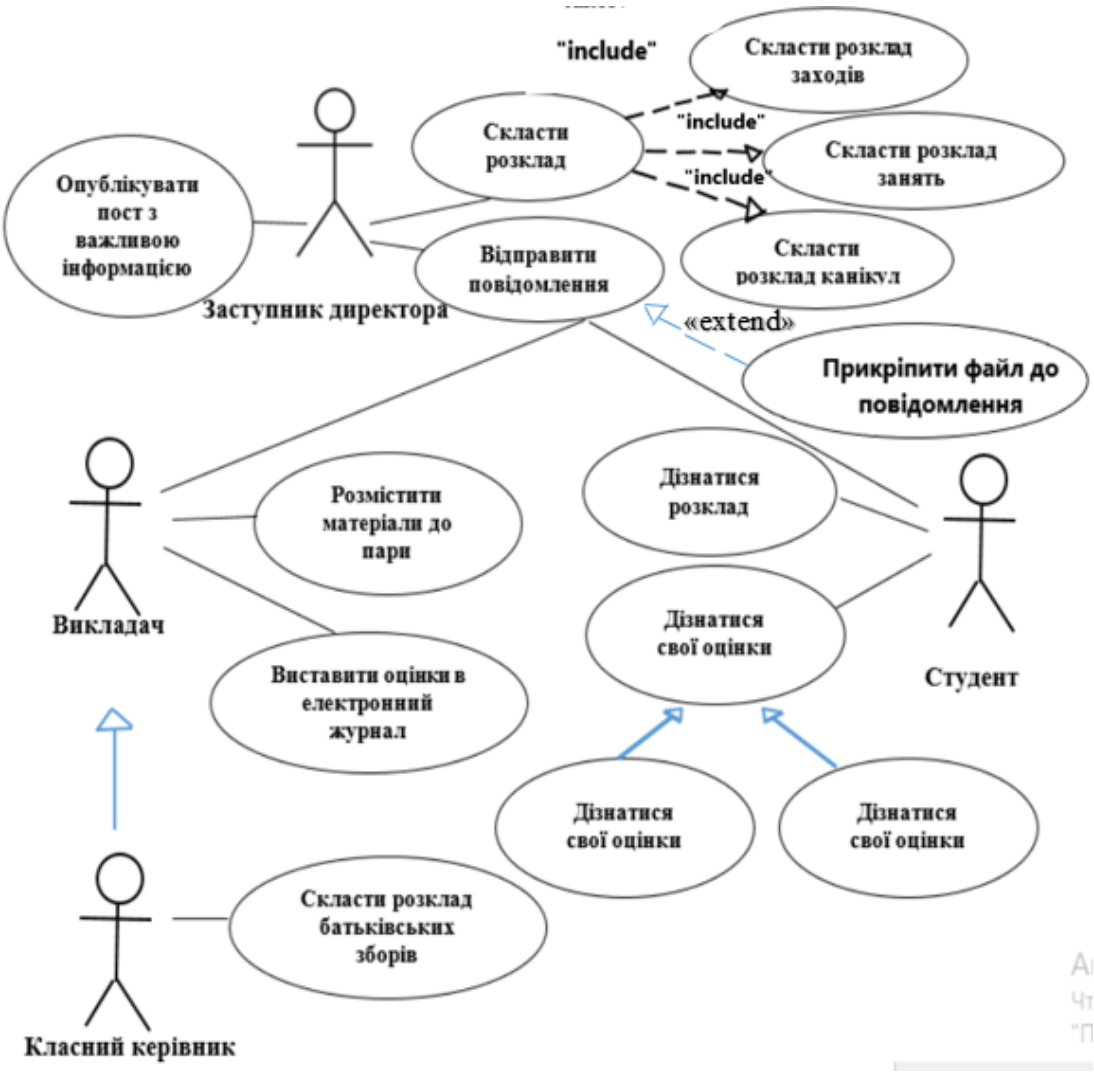
Розуміння цього критично важливе для грамотного використання цього виду відношень.

Повернемося до нашого основного прикладу. Ми хочемо, щоб дія "прикріпити файл до повідомлення" розширювала дію "надіслати повідомлення". На діаграмі це зображується таким чином:



Розширюємо функціонал надсилання повідомлень за допомогою функції прикріплення файлів до повідомлення (Необов'язково прикріплювати файл до кожного повідомлення)

Як підсумок, отримаємо таку діаграму:



Четверта версія діаграми

Тема: Діаграма класів

Мета: Розглянути основні поняття діаграми класів. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   1. Готовність групи до заняття;
   2. Психоемоційний настрій;
   3. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   1. Повідомлення теми та мети заняття;
   2. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграма класів. Переваги та недоліки класів.
2. Класи:

- Атрибути класу
- Операції(методи) класу

3. Відношення між класами:

- асоціація
- залежність
- узагальнення
- реалізація
- агрегація
- композиція

IV. Узагальнення та систематизація знань.

IV. Підведення підсумків заняття.

VI. Домашнє завдання:

1. Ознайомитись з теоретичними відомостями лекції
2. Вивчити основні поняття лекції.
3. Дати відповіді на контрольні питання.

Контрольні питання:

1. Що таке клас?
2. Що таке атрибут класу?
3. Що таке квантор видимості?
4. Що таке тип атрибуту?
5. Що таке операція класу?
6. Як записуються операції?
7. Що таке асоціація?
8. Для чого використовується кратність?
9. Що таке N-арна асоціація?
10. Що таке спадкування?
11. Що таке множинне спадкування?
12. Для чого використовують рядки-обмеження?
13. Які ключові слова можна використовувати в якості рядків-обмежень? Що вони означають?
14. Що таке агрегація? Як графічно можна представити агрегацію?
15. В чому різниця між агрегацією та композицією?
16. Що таке залежність?
17. Що таке стереотип залежності? Які стереотипи залежностей Вам відомі?

Діаграма класів (*class diagram*) - статичне представлення структури моделі. Відображає статичні елементи, такі як: класи, типи даних, їх зміст та відношення.

Діаграма класів є ключовим елементом редактора UML-діаграм, оскільки часто додатки генеруються саме з діаграми класів

На відміну від діаграми послідовностей, діаграми діяльності тощо, діаграма класів є найпопулярнішою діаграмою UML.

Переваги діаграми класів:

- ❖ Будь-яку просту або складну модель даних можна проілюструвати за допомогою діаграми класів для отримання максимальної інформації.
- ❖ Схематику програми можна зрозуміти за допомогою неї.

- ❖ Будь-яка потреба в системі може бути візуалізована та передана по всьому бізнесу для конкретних дій.
- ❖ Будь-яка вимога щодо реалізації конкретного коду може бути виділена за допомогою діаграм і запрограмована до описаної структури.
- ❖ Опис, незалежний від реалізації, може бути наданий та переданий компонентам.

Недоліки діаграми класів:

- ❖ Діаграми класів часто можуть зайняти багато часу
- ❖ Складна діаграма не допомагає розробникам програмного забезпечення в їх роботі. Можуть виникнути ситуації, коли розробники засмучуються через структуру діаграм класів.
- ❖ Розміщення конструкції може перешкоджати розробникам та компаніям.

Незважаючи на важливість діаграми класів у життєвому циклі розробки програмного забезпечення, вона, звичайно, не позбавлена недоліків і може ускладнити життя розробникам та компаніям.

Діаграма класів може бути поділена на три компоненти:

1) Верхній розділ, який складається з назви класу, і є обов'язковим компонентом.
2) У середньому розділі описані якості класу та використовуються під час опису конкретного екземпляра класу.
3) У нижньому розділі описано взаємодію класу з даними.

На діаграмах класів зазвичай представлені такі елементи:

- ✓ Класи;
- ✓ Інтерфейси;
- ✓ Залежності, узагальнення та асоціації

Також діаграми можуть включати в себе пакети або підсистеми; ті та інші групують елементи моделі у більш великі утворення. Іноді в діаграму класів потрібно додати примірники.

Класи

Клас в мові UML служить для позначення множини об’єктів, які мають однакову структуру, поведінку і відносини з об’єктами інших класів.

Графічно клас зображується у вигляді прямокутника, який додатково може бути розділений горизонтальними лініями на розділи або секції. У цих розділах можуть зазначатися ім’я класу, атрибути (змінні) та операції (методи).

Ім’я класу має бути унікальним в межах пакету, який описується деякою сукупністю діаграм класів або однією діаграмою.

Ім'я вказується в самій верхній секції прямокутника, тому вона часто називається секцією імені класу.

Ім'я класу записується в центрі секції імені напіжирним шрифтом і повинне починатися з великої літери.

Для позначення імені абстракного класу використовується курсив

У деяких випадках необхідно вказати, до якого пакета відноситься клас. Для цього використовується спеціальний символ розподільник – подвійна двокрапка – (::)

Синтаксис рядка буде таким <Ім’я пакету>::<Ім’я класу>

Наприклад **Банк::Рухунок**

Атрибути класу

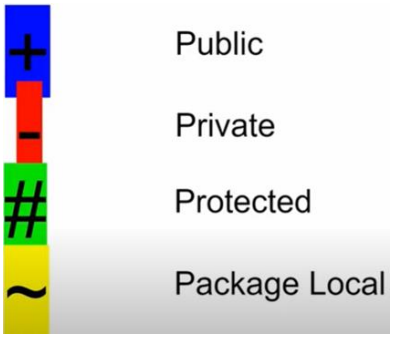
Ім’я атрибута – рядок тексту, який використовується в якості ідентифікатора відповідного атрибута і тому повинний бути унікальним в межах даного класу.

Атрибути класу або властивості записуються у другій зверху секції прямокутника класу. В UML кожному атрибуту класу відповідає окремий рядок тексту, який складається з квантора видимості атрибута і , можливо, початкового значення:

<квантор видимості><ім’я атрибута>[кратність]: <тип атрибута> = <початкове значення>

{рядок властивості}

Квантор видимості може приймати одне із можливих значень і відображається за допомогою відповідних спеціальних символів:



Квантор видимості може приймати одне із можливих значень і відображається за допомогою відповідних спеціальних символів:

«+» позначає атрибут з областю видимості типу загальнодоступний (public). Атрибут з цією областю видимості доступний з будь-якого іншого класу пакету, в якому визначена діаграма;

«#» атрибут із зоною видимості типу захищений (protected). Недоступний для всіх класів, за винятком підкласів даного класу;

«-» атрибут із зоною видимості типу закритий «private». Недоступний для всіх класів без винятків;

«~» видимість для елементів того ж простору імен (пакета) (package). Квантор видимості може бути опущений. У цьому випадку його відсутність просто означає, що видимість атрибута не вказується. Ця ситуація відрізняється від прийнятих за замовчуванням угод у традиційних мовах програмування, коли відсутність квантора видимості трактується як public або private. Однак замість умовних графічних позначень можна записувати відповідне ключове слово: public, protected, private.

Ім'я атрибута являє собою рядок тексту, який використовується в якості ідентифікатора відповідного атрибута й тому повинен бути унікальним в межах даного класу. Ім'я атрибута є єдиним обов'язковим елементом синтаксичного позначення атрибута.

Кратність атрибута характеризує загальну кількість конкретних атрибутів даного типу, що входять до складу окремого класу. У загальному випадку кратність записується у формі рядка тексту у квадратних дужках після імені відповідного атрибута:

[нижня_границя1 .. верхня_границя1, нижня_границя2.. верхня_границя2, ..., нижня_границяk .. верхня_границяk],

де нижня_границя й верхня_границя є позитивними цілими числами, кожна пара яких служить для позначення окремого замкненого інтервалу цілих чисел, у якого нижня (верхня) границя дорівнює значенню нижня_границя (верхня_границя).

У цілому дана умовна позначка кратності відповідає теоретико-множинному об'єднанню відповідних інтервалів. У якості верхньої_границі може використовуватися спеціальний символ "*", який означає довільне позитивне ціле число. Інакше кажучи, це означає необмежене зверху значення кратності відповідного атрибута.

Значення кратності з інтервалу випливають у монотонно зростаючому порядку без пропуску окремих чисел, що лежать між нижньої й верхньої границями. При цьому дотримуються наступного правила: відповідні нижні й верхні границі інтервалів включаються в значення кратності. Якщо в якості кратності вказується однина, то кратність атрибута ухвалюється рівної даному числу. Якщо ж вказується єдиний знак "*", то це означає, що кратність атрибута може бути довільним позитивним цілим числом або нулем.

Якщо кратність атрибута не зазначена, то за замовчуванням ухвалюється її значення рівне 1..1, тобто в точності 1.

Тип атрибута являє собою вираження, семантика якого визначається мовою специфікації відповідної до моделі. У нотації UML тип атрибута іноді визначається залежно від мови програмування, яка передбачається використовувати для реалізації даної моделі. У найпростішому випадку тип атрибута вказується рядком тексту, що має осмислене значення в межах пакета або моделі, до яких ставиться розглянутий клас.

Початкове значення служить для завдання деякого початкового значення для відповідного атрибута в момент створення окремого екземпляра класу. Тут необхідно дотримуватися правила приналежності значення типу конкретного атрибута. Якщо початкове значення не зазначене, то значення відповідного атрибута не визначене на момент створення нового екземпляра класу. З іншого боку, конструктор відповідного об'єкта може перевизначати початкове значення в процесі виконання програми, якщо в цьому виникає необхідність.

Операції (методи) класу

Квантор видимості, як і у випадку атрибутів класу, може ухвалювати одне із трьох можливих значень і, відповідно, відображається за допомогою спеціального символу.

Символ "+" позначає операцію з областю видимості типу загальнодоступний (public). Символ "#" позначає операцію з областю видимості типу захищений (protected).

І, нарешті, символ "-" використовується для позначення операції з областю видимості типу закритий (private).

Квантор видимості для операції може бути опущений.

Ім'я операції являє собою рядок тексту, який використовується в якості ідентифікатора відповідної до операції й тому повинна бути унікальної в межах даного класу. Ім'я атрибута є єдиним обов'язковим елементом синтаксичного позначення операції.

Список параметрів є переліком розділених комами формальних параметрів, кожний з яких може бути представлений у наступному виді:

<вид параметра><ім'я параметра>:<вираження типу>=<значення параметра за замовчуванням>.

Тут *вид параметра* - є одне із ключових слів in, out або inout зі значенням in за замовчуванням, у випадку якщо вид параметра не вказується.

Ім'я параметра є ідентифікатор відповідного формального параметра.

Вираження типу є залежною від конкретної мови програмування специфікацією типу значення, що вертається, для відповідного формального параметра.

Значення за замовчуванням у загальному випадку являє собою вираження для значення формального параметра, синтаксис якого залежить від конкретної мови програмування й підкоряється прийнятим у ньому обмеженням.

Вираз типу значення, що повертається, також є залежною від мови реалізації специфікацією типу або типів значень параметрів, які повертаються об'єктом після виконання відповідної операції.

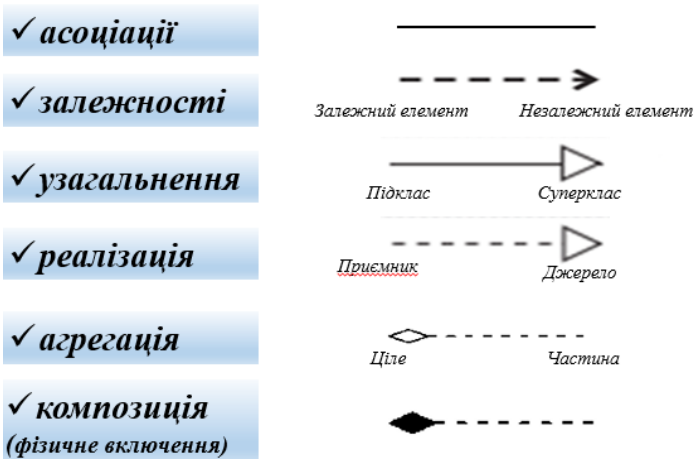
Рядок властивості служить для вказівки значень властивостей, які можуть бути застосовані до даного елемента. Рядок властивості не є обов'язковим, він може бути відсутнім, якщо ніякі властивості не специфіковані.

Відношення між класами

Крім внутрішньої будови або структури класів на відповідній діаграмі вказуються відношення між класами.

Кожне з цих відношень має власне графічне представлення на діаграмі, яке відображає взаємозв'язки між об'єктами відповідних класів.

Базовими відношеннями в мові UML є:



Відношення асоціації

Відношення асоціації відповідає наявності деякого відношення між класами. Дане відношення позначається суцільною лінією з додатковими спеціальними символами, які характеризують окремі властивості конкретної асоціації.

Ім'я асоціації є необов'язковим елементом її позначення. Якщо воно задане, то записується з великої букви поруч з лінією відповідної асоціації.

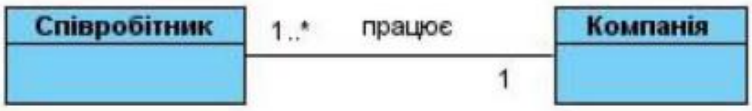
Для асоціації може позначатися кількість екземплярів об’єктів кожного класу, які беруть участь у зв’язку.

(0 - якщо жодного, 1 - якщо один, * - якщо багато)

Можуть вказуватися мінімальна й максимальна кількість, наприклад, 0 або 1...* означає, що на відповідному кінці асоціації може не бути жодного екземпляра, бути один або багато.

Бінарна асоціація – служить для зображення довільного відношення між двома класами. Вона зв’язує в точності два різних класа й може бути ненаправленим або направленим відношенням. Окремий випадок бінарної асоціації- рефлекційна асоціація, що зв'язує клас із самим собою. Ненаправлена асоціація зображується лінією без стрілки.

Як простий приклад ненаправленої бінарної асоціації можна розглянути відношення між двома класами – класом Компанія та класом Співробітник. Вони зв’язані між собою бінарною асоціацією «працює». Для даного відношення визначений наступний порядок читання - співробітник працює в компанії.



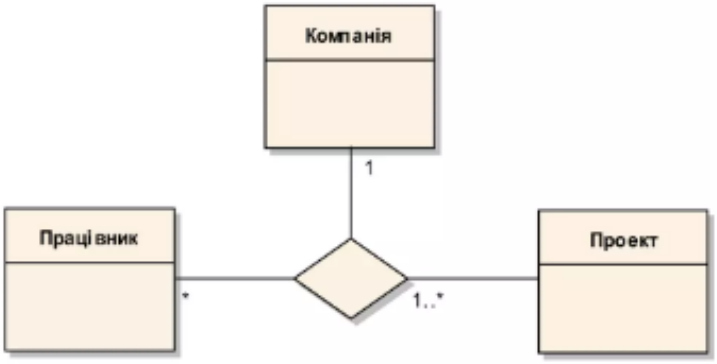
Тернарна асоціація й асоціації більш високої арності в загальному випадку називаються *N-арною асоціацією*. Така асоціація зв'язує деяким відношенням 3 і більш класів, при цьому один клас може брати участь в асоціації більш ніж один раз.

Клас асоціації має певну роль у відповідному відношенні, що може бути явно зазначене на діаграмі. Кожний екземпляр N-арної асоціації являє собою N-арний кортеж значень об'єктів з відповідних класів.

Бінарна асоціація є часним випадком N-арної асоціації, коли значення N=2, і має своє власне позначення.

N-арна асоціація графічно позначається ромбом, від якого ведуть лінії до символів класів даної асоціації. У цьому випадку ромб з'єднується із символами відповідних класів суцільними лініями. Звичайно лінії проводяться від вершин ромба або від середини його сторін. Ім'я N-арної асоціації записується поруч із ромбом відповідної асоціації.

Деякий клас може бути приєднаний до ромба пунктирною лінією. Це означає, що даний клас забезпечує підтримку властивостей відповідної N-Арної асоціації, а сама N-Арна асоціація має атрибути, операції й/або асоціації. Інакше кажучи, така асоціація, у свою чергу, є класом з відповідним позначенням у вигляді прямокутника і є самостійним елементом мови UML - асоціацією-класом (Association Class). N- Арная асоціація не може містити символ агрегації ні для якої зі своїх ролей.



Графічне зображення тернарної асоціації між трьома класами

Відношення узагальнення

Узагальнення – відношення між більш загальним поняттям (предком) і менш загальним поняттям (нащадком).

Менш загальний елемент моделі повинен бути позгодженний з більш загальним елементом і може містити додаткову інформацію. Дане відношення використовується для подання ієрархічних взаємозв'язків між різними елементами мови UML, такими як пакети, класи, варіанти використання.

Наслідування (Inheritance) - спеціальний концептуальний механізм, за допомогою якого більш спеціальні елементи включають в себе структуру і поведінку більш загальних елементів. Клас-нащадок має всі властивості і поведінку класу-предка, а також має власні властивості і поведінку, які можуть бути відсутніми у класа-предка.

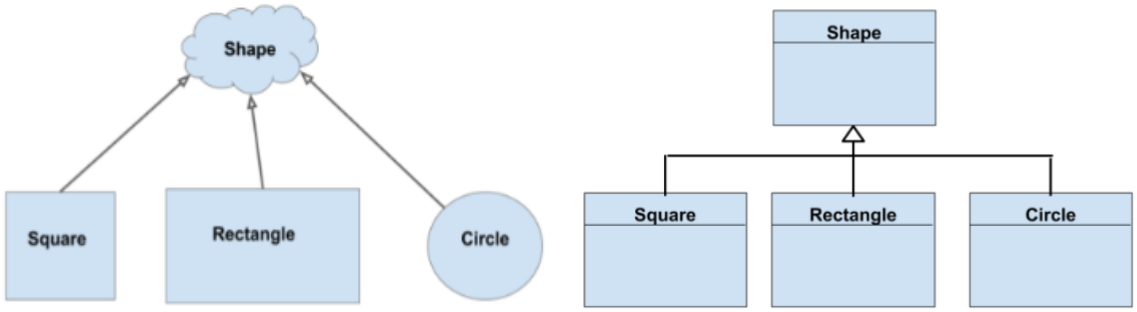
Батько, предок (Parent) - щодо узагальнення більш загальний елемент.

Нащадок (Child) - спеціалізація одного з елементів відношення узагальнення, званого в цьому випадку батьком.

На діаграмах відношення узагальнення позначається суцільною лінією з трикутною стрілкою на одному з кінців. Стрілка вказує на більш загальний клас (клас-предок), а її початок - на більш спеціальний клас (клас-нащадок).

На діаграмі може вказуватися кілька ліній для одного відношення узагальнення, що відбиває його характер. У цьому випадку більш загальний клас розбивається на підкласи одним відношенням узагальнення.

Наприклад, клас Shape(Фігура) може виступати в якості суперкласу для підкласів, що відповідають конкретним геометричним фігурам, таким як, Square(Квадрат), Rectangle(Прямокутник), Circle(Коло) та ін. Даний факт може бути представлений графічно у формі діаграми класів наступного виду:



Поруч із стрілкою узагальнення може розміщатися рядок тексту, що вказує на деякі додаткові властивості цього відношення. Даний текст буде ставитися до всіх ліній узагальнення, які йдуть до класів-нащадків. Інакше кажучи, відзначена властивість стосується всіх підкласів даного відношення. При цьому текст слід розглядати як обмеження, і тоді він записується у фігурних дужках.

У якості обмежень можуть бути використані наступні ключові слова мови UML:

1. ***{complete}*** *- означає, що в даному відношенні узагальнення специфіковані всі класи-нащадки, і інших класів-нащадків у даного класу-предка бути не може. Приклад - клас Клієнт_банку є предком для двох класів: Фізична_особа й Компанія, і інших класів-нащадків він не має. На відповідній діаграмі класів це можна вказати явно, записавши поруч із лінією узагальнення даний рядок- обмеження;*
2. ***{disjoint}*** *- означає, що класи-нащадки не можуть містити об'єктів, що одночасно є екземплярами двох або більше класів. У наведеному вище прикладі ця умова також виконується, оскільки передбачається, що ніяка конкретна фізична особа не може бути одночасно й конкретною компанією. У цьому випадку поруч із лінією узагальнення можна записати даний рядок-обмеження;*

3. {incomplete} *- означає випадок, протилежний першому. А саме, передбачається, що на діаграмі зазначені не всі нащадки. Надалі можливо заповнити їхній перелік не змінюючи вже побудовану діаграму. Приклад - діаграма класу "Автомобіль", для якої вказівка всіх без винятку моделей автомобілів порівняння зі створенням відповідного каталогу. З іншого боку, для окремого завдання, такого як розробка системи продажу автомобілів конкретних моделей, у цьому немає необхідності. Але вказати неповноту структури класів-нащадків все-таки випливає;*

4. {overlapping} *- означає, що окремі екземпляри класів- нащадків можуть належати одночасно декільком класам. Приклад - клас "Багатокутник" є класом-предком для класу "Прямокутник" і класу "Ромб". Однак існує окремий клас "Квадрат", екземпляри якого одночасно є об'єктами перших двох класів. Цілком природно таку ситуацію вказати явно за допомогою даною рядка-обмеження.*

Відношення Агрегація

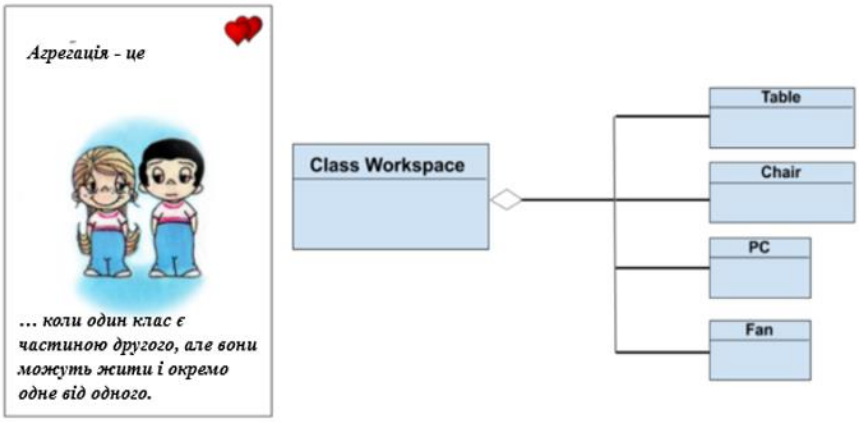
Агрегація (Aggregation) - спеціальна форма асоціації, яка служить для подання відношення типу "частина-ціле" між агрегатом (ціле) і його складовою частиною.

Ставлення агрегації має місце між декількома класами в тому випадку, якщо один з класів є сутність, яка включає в себе в якості складових частин інші сутності.

Графічно відношення агрегації зображується суцільною лінією, один з кінців якої являє собою не зафарбований усередині ромб. Цей ромб вказує на той клас, який являє собою "ціле" або клас-контейнер. Решта класи є його "частинами".

Як приклад відношення агрегації можна розглянути взаємозв'язок типу "частина-ціле", який має місце між класом Системний блок персонального комп'ютера і його складовими частинами: Процесор, Материнська плата, Оперативна пам'ять, Жорсткий диск і Гнучкий диск. Використовуючи позначення мови UML, компонентний склад системного блоку можна представити у вигляді відповідної діаграми класів, яка в даному випадку ілюструє ставлення агрегації.

Наприклад, робоче місце програміста складається зі стільця, стола, комп’ютера та вентилятора, але при знищенні класу «робоче місце», в нас просто залишуться всі ці класи, лише окремо один від одного.

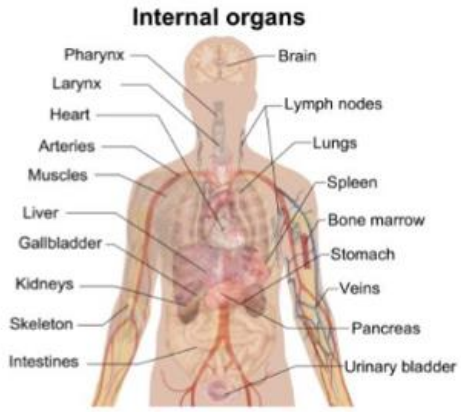


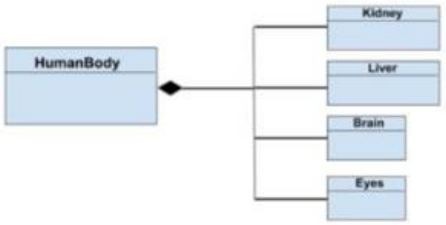
Відношення композиції

Композиція (Composition) - різновид відношення агрегації, при якій складові частини цілого мають такий же час життя, що й саме ціле. Ці частини знищуються разом зі знищенням цілого.

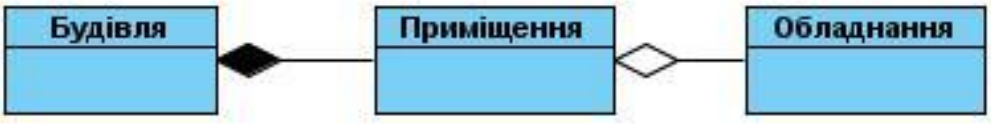
Графічно відношення композиції зображується суцільною лінією, один з кінців якої являє собою зафарбований усередині ромб. Цей ромб вказує на той клас, який представляє собою клас-композит. Решта класи є його "частинами"

Наприклад, наше тіло складається з органів, але самі по собі вони не дієздатні.





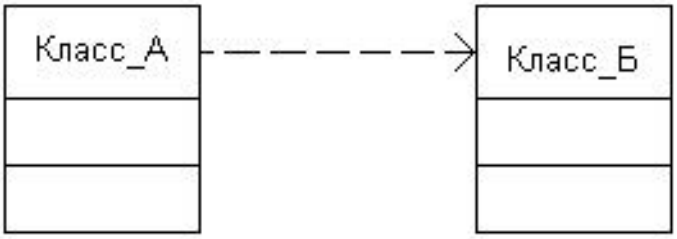
Зауваження: Для виявлення типу зв'язку – *агрегація* чи *композиція* варто проаналізувати, що буде з класом-частиною після знищення класу контейнеру. Чи матиме він окремий сенс в рамках поставленої задачі? Так, наприклад, після знищення Будівлі, Приміщення припиняє своє існування, а от Обладнання матиме сенс.



Відношення залежності

Відношення залежності графічно зображуються пунктирною лінією між відповідними елементами зі стрілкою, направленою від класу-клієнту залежності до незалежного класу або класу-джерела.

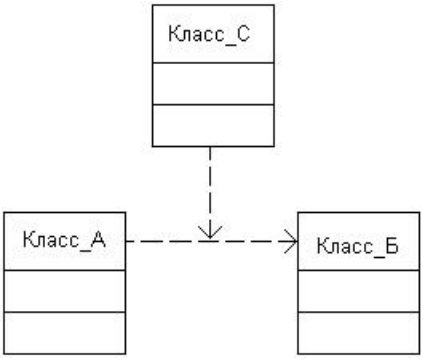
На рисунку зображено два класи: Клас_А і Клас_Б, при цьому Клас_Б є джерелом деякої залежності, а Клас_А - клієнтом цієї залежності.



Графічне зображення відношення залежності на діаграмі

Наприклад, у класа Person є метод hasRead з вхідним параметром book, який повертає true, якщо, наприклад, людина прочитала книгу.

У якості класу-клієнта й класу-джерела залежності можуть виступати цілі безлічі елементів моделі. У цьому випадку одна лінія зі стрілкою, що виходить від джерела залежності, розщеплюється в деякій крапці на кілька окремих ліній, кожна з яких має окрему стрілку для класу-клієнта. Наприклад, якщо функціонування Класу_С залежить від особливостей реалізації Класу_А и Класу_Б, то дана залежність може бути зображена в такий спосіб (рисунок).



Графічне представлення залежності між класом-клієнтом (Клас_С) і класами-джерелами (Клас_А та Клас_Б)

Стрілка може позначатися необов'язковим, але стандартним ключовим словом у лапках і необов'язковим індивідуальним іменем.

Для відношення залежності зумовлені ключові слова (стереотипи), які позначають деякі спеціальні його види:

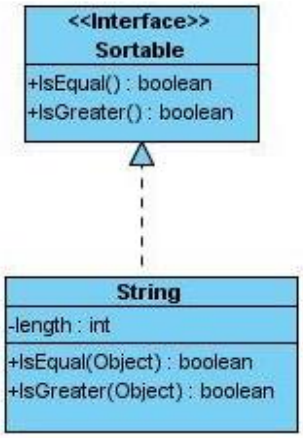
- **«access»** - для позначення доступності відкритих атрибутів і операцій класу-джерела для класів-клієнтів;
- **«bind»** - клас-клієнт може використовувати деякий шаблон для своєї подальшої параметризації;
- **«derive»** - атрибути класу-клієнту можуть бути обчислені по атрибутах класу-джерела;
- **«import»** - відкриті атрибути і операції класу-джерела стають частиною класу-клієнту, так, ніби вони були оголошені безпосередньо в ньому;
- **«refine»** - вказує, що клас-клієнт служить уточненням клас-джерела в силу причин історичного характеру, коли з'являється додаткова інформація в ході роботи над проектом;
- **«use»** - семантика початкового елементу залежить від семантики відкритого змісту цільового елемента;
- та інші.

Відношення реалізації

Реалізація (Realization) - це семантичне відношення між класифікаторами, при якому один класифікатор визначає сутність, а інший гарантує її виконання.

Відношення реалізації в діаграмах класів зустрічаються між інтерфейсами і класами, що реалізовують їх.

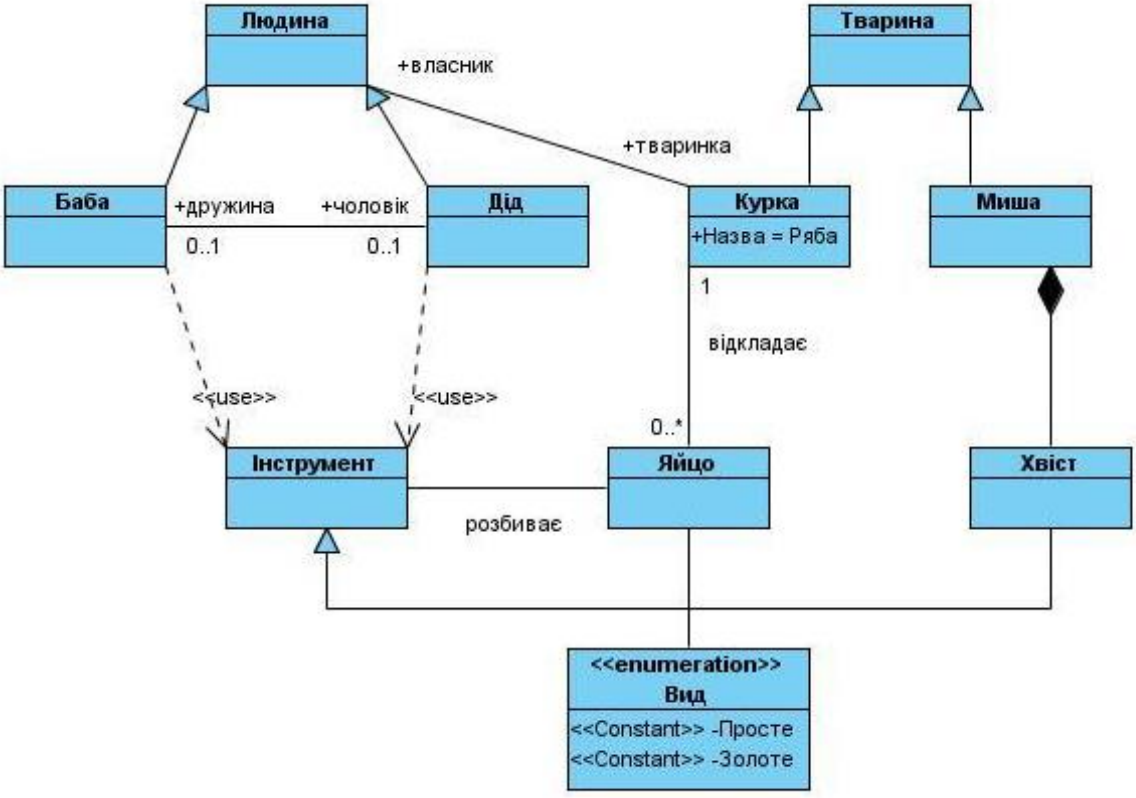
Зображається відношення реалізації у вигляді пунктирної лінії з незафарбованою стрілкою, як щось середнє між відношеннями узагальнення та залежності.



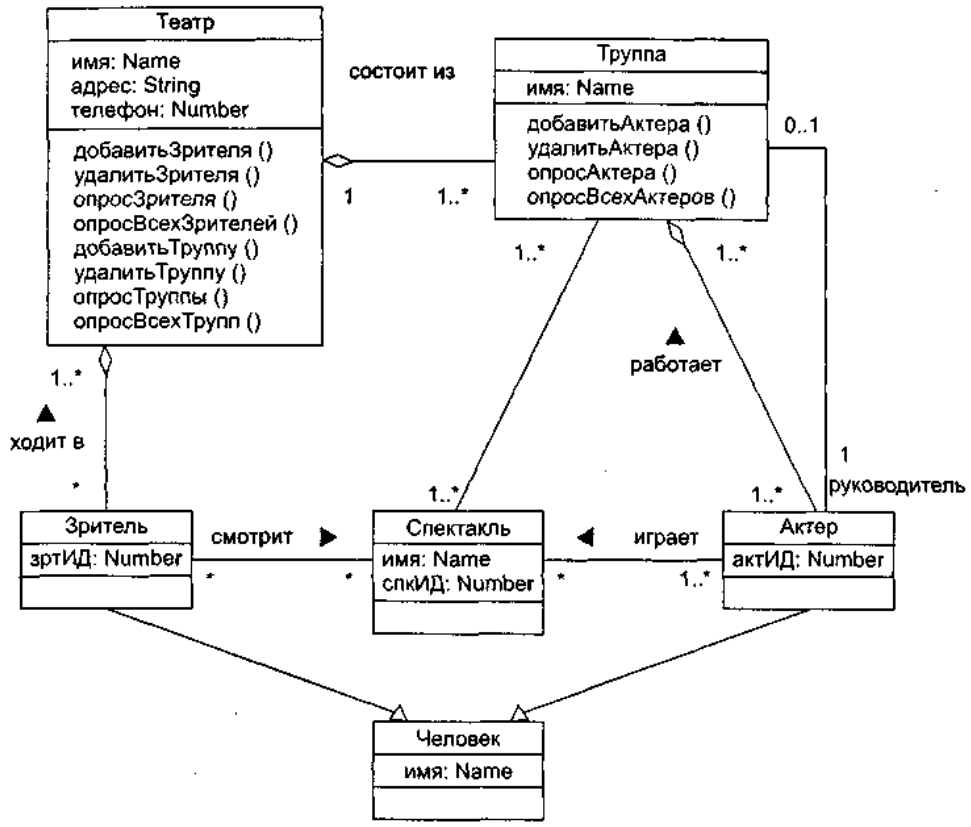
Приклад:

Для наочності вищевикладеного матеріалу, розглянемо діаграму класів, яка ілюструє відому казку «Курка Ряба».

У казці Дід (клас Дід) та Баба (клас Баба) є Людьми (класи Дід та Баба наслідують від класу Людина), та вони є власниками (відношення асоціації) Курки (клас Курка) на ім'я Ряба. В свою чергу, Курка Ряба є твариною (тобто клас Курка наслідує від класу Тварина). Курка може відкладати яйця (клас Яйце, відношення асоціації «відкладати»). Зауважимо, що яйце в казці фігурувало як золоте, так і просте, що відображується на діаграмі за допомогою класу <<enumeration>> Вид. Ще однією дійовою особою казки є Миша (клас Миша), яка також є Твариною та володіє Хвостом (клас Хвіст) як інструментом (клас Інструмент) для розбиття яйця (відношення асоціації «розбиває)».



Приклад діаграми класів інформаційної системи театру



Тема: Діаграма схем станів.

Мета: Розглянути основні поняття діаграми станів. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Моделювання поведінки програмної системи;
2. Діаграми схем станів;
   a. Автомати;
   b. Стан;
   c. Список внутрішніх дій;
   d. Початковий стан;
   e. Кінцевий стан;
   f. Перехід;
   g. Подія;
   h. Сторожова умова.

IV. Узагальнення та систематизація знань;

V. Підведення підсумків занять;

VI. Домашнє завдання:
1. Ознайомитись з теоретичним матеріалом теми.
2. Вивчити основні поняття лекції.
3. Відповісти на контрольні питання.

Контрольні питання:
1. Що використовують для моделювання поведінки системи?
2. Дайте визначення поняттю автомат?
3. За допомогою чого можна відобразити автомат?
4. Що відображає діаграма схем станів?
5. На виконанні яких обов'язкових умов заснований формалізм діаграм станів?
6. Що таке стан?
7. Дайте визначення поняттю список внутрішніх подій.
8. Які види псевдо-станів Вам відомі?
9. Що таке подія?
10. Для чого використовується сторожова умова?

Динамічні моделі забезпечують представлення поведінки систем. «Динамізм» цих моделей полягає в тому, що в них відбувається зміна станів у процесі роботи системи (залежно від часу). Засоби мови UML для створення динамічних моделей численні й різноманітні. Ці засоби орієнтовані не тільки на властиво програмні системи, але й на відображення вимог замовника до поведінки таких систем.

Моделювання поведінки програмної системи

Для моделювання поведінки системи використовують:
1. автомати;
2. взаємодії.

Автомат (State machine) описує поведінку в термінах послідовності станів, через які проходить об'єкт протягом свого життя. *Взаємодія* (Interaction) описує поведінку в термінах обміну повідомленнями між об'єктами.

Таким чином, автомат задає поведінку системи як цільної, єдиної сутності; моделює життєвий цикл єдиного об'єкта. У силу цього автоматний підхід зручно застосовувати для формалізації динаміки окремого важкого для розуміння блоку системи.

Взаємодії визначають поведінку системи у вигляді комунікацій між його частинами (об'єктами), представляючи систему як співтовариство спільно працюючих об'єктів. Саме тому взаємодії вважають основним апаратом для фіксації повної динаміки системи.

Автомати відображають за допомогою:
- діаграм схем станів;
- діаграм діяльності.

Взаємодії відображають за допомогою:
- діаграм співробітництва (кооперації);
- діаграм послідовності.

Діаграми схем станів

Діаграма схем станів - одна з п'яти діаграм UML, що моделюють динаміку систем. Діаграма схем станів відображає кінцевий автомат, виділяючи потік керування, що випливає від стану до стану. *Кінцевий автомат* - поведінка, яка визначає послідовність станів у ході існування об'єкта. Ця послідовність розглядається як відповідь на події й включає реакції на ці події.

Діаграма схем станів показує:

1) набір станів системи;

2) події, які викликають перехід з одного стану в інше;

3) дії, які відбуваються в результаті зміни стану.

Головне призначення цієї діаграми - описати можливі послідовності станів і переходів, які в сукупності характеризують поведінка елемента моделі протягом його життєвого циклу. Діаграма станів представляє динамічну поведінку сутностей, на основі специфікації їх реакції на сприйняття деяких конкретних подій. Системи, які реагують на зовнішні дії від інших систем або від користувачів, іноді називають реактивними. Якщо такі дії ініціюються в довільні випадкові моменти часу, то говорять про асинхронну поведінку моделі.

Хоча діаграми станів найчастіше використовуються для опису поведінки окремих екземплярів класів (об'єктів), але вони також можуть бути застосовані для специфікації функціональності інших компонентів моделей, таких як варіанти використання, актори, підсистеми, операції й методи.

Діаграма станів по суті є графом спеціального виду, який представляє деякий автомат. Поняття автомата в контексті UML має досить специфічну семантику, засновану на теорії автоматів. Вершинами цього графа є стани й деякі інші типи елементів автомата (псевдо-стани), які зображуються відповідними графічними символами. Дуги графа служать для позначення переходів зі стану в стан. Діаграми станів можуть бути вкладені друг у друга, створюючи вкладені діаграми більш детального представлення окремих елементів моделі.

Автомати

Автомат (state machine) у мові UML являє собою деякий формалізм для моделювання поведінки елементів моделі й системи в цілому. У метамоделі UML автомат є пакетом, у якому визначена безліч понять, необхідних для представлення поведінки сутності у вигляді дискретного простору з кінцевим числом станів і переходів. З іншого боку, автомат описує поведінку окремого об'єкта у формі послідовності станів, які охоплюють усі етапи його життєвого циклу, починаючи від створення об'єкта й закінчуючи його знищенням. Кожна діаграма станів представляє деякий автомат.

Рисунок 1 - Найпростіший приклад діаграми станів для технічного обладнання типу комп'ютер

Основними поняттями, що входять у формалізм автомата, є стан і перехід. Головна відмінність між ними полягає в тому, що тривалість знаходження системи в окремому стані суттєво перевищує час, який затрачається на перехід з одного стану в інше. Передбачається, що в межі час переходу з одного стану в інше

дорівнює нулю (якщо додатково нічого не сказане). Інакше кажучи, перехід об'єкта зі стану в стан відбувається миттєво.

Поведінка моделюється як послідовне переміщення по графові станів від вершини до вершини по єднальних їхніх дугах з обліком їх орієнтації. Для графа станів системи можна ввести в розгляд спеціальні властивості.

Одним з таких властивостей є виділення із усієї сукупності станів двох спеціальні: *початкового й кінцевого*. Хоча ні в графові станів, ні на діаграмі станів час знаходження системи в тому або іншому стані явно не враховується, передбачається, що послідовність зміни станів упорядкована в часі. Інакше кажучи, кожний наступний стан завжди наступає пізніше попереднього йому стану.

Формалізм автоматів допускає вкладення одних автоматів в інші для уточнення внутрішньої структури окремих більш загальних станів (макростанів). У цьому випадку вкладені автомати одержали назву *під-автоматів*. Під-автомати можуть використовуватися для внутрішньої специфікації процедур і функцій, що утворюють поведінку вихідного об'єкта. Наприклад, стан несправності технічного обладнання (рис. 1) може бути деталізований на окремі під-стани, кожне з яких може характеризувати несправність окремих підсистем, що входять до складу даного обладнання.

У мові UML поняття автомата доповнене спеціальною семантикою вхідних у відповідний пакет елементів.

Формалізм звичайного автомата заснований на виконанні наступних обов'язкових умов:

1. Автомат не запам'ятовує історію переміщення зі стану в стан. З погляду моделюємої поведінки визначальним є сам факт знаходження об'єкта в тому або іншому стані, але ніяк не послідовність станів, у результаті якої об'єкт перейшов у поточний стан. Інакше кажучи, автомат "забуває" усі стани, які передували поточному в цей момент часу.
2. У кожний момент часу автомат може перебувати в одному й тільки в одному зі своїх станів. Це означає, що формалізм автомата призначений для моделювання послідовної поведінки, коли об'єкт протягом свого життєвого циклу послідовно проходить через усі свої стани. При цьому автомат може перебувати в окремому стані як завгодно довго, якщо не відбувається ніяких подій.
3. Хоча процес зміни станів автомата відбувається в часі, явно концепція часу не входить у формалізм автомата. Це означає, що тривалість знаходження автомата в тому або іншому стані, а також час досягнення того або іншого стану ніяк не специфікуються. Інакше кажучи, час на діаграмі станів присутній в неявному виді, хоча для окремих подій може бути зазначений інтервал часу й у явному виді.
4. Кількість станів автомата повинне бути обов'язково кінцевою (у мові UML розглядаються тільки кінцеві автомати), і всі вони повинні бути специфіковані явно. При цьому окремі псевдо-стани можуть не мати специфікацій (початковий і кінцевий стану). У цьому випадку їх призначення й семантика повністю визначаються з контексту моделі й розглянутої діаграми станів.
5. Граф автомата не повинен містити ізольованих станів і переходів. Ця умова означає, що для кожного зі станів, крім початкового, повинне бути визначений попередній стан. Кожний перехід повинен обов'язково з'єднувати два стани

автомата. Допускається перехід зі стану в себе, такий перехід ще називають "петлею".

6. Автомат не повинен містити конфліктуючих переходів, тобто таких переходів з того самого стану, коли об'єкт одночасно може перейти у два й більш наступні стани (крім випадку паралельних під-автоматів). У мові UML виключення конфліктів можливо на основі введення так званих *сторожових умов*.

Таким чином, правила поведінки об'єкта, що моделюється деяким автоматом, визначаються, з одного боку, загальним формалізмом автомата, а з іншого - його графічним зображенням у мові UML у формі конкретної діаграми станів.

Стан

У мові UML під *станом* розуміється абстрактний мета клас, використовуваний для моделювання окремої ситуації, протягом якої має місце виконання деякої умови. Стан може бути заданий у вигляді набору конкретних значень атрибутів класу або об'єкта, при цьому зміна їх окремих значень буде відбивати зміну стану моделюємого класу або об'єкта.

Не кожний атрибут класу може характеризувати його стан. Як правило, мають значення тільки такі властивості елементів системи, які відбивають динамічний або функціональний аспект її поведінки. У цьому випадку стан буде характеризуватися деякою інваріантною умовою, що включають у себе тільки значимі для поведінки класу атрибути і їх значення.

Елемент переходить у стан в момент початку відповідної діяльності й залишає даний стан у момент її завершення.

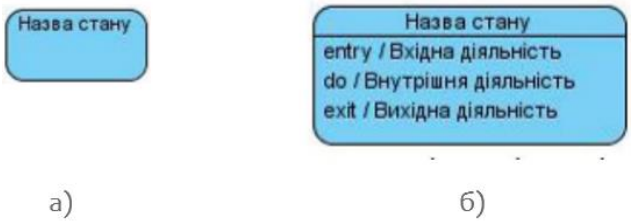


Рисунок 2 - Графічне зображення станів на діаграмі станів

Стан на діаграмі зображується прямокутником з округленими вершинами (рис. 2). Цей прямокутник, у свою чергу, може бути розділений на дві секції горизонтальною лінією. Якщо зазначена лише одна секція, то в ній записується тільки ім'я стану (рис. 2, а). А якщо ні, то в першій з них записується ім'я стану, а в другий - список деяких *внутрішніх дій* або *переходів* у даному стані (рис. 2, б). При цьому під *дією* в мові UML розуміють деяку атомарну операцію, виконання якої приводить до зміни стану або поверненню деякого значення (наприклад, "істина" або "неправда").

Список внутрішніх дій

Ця секція містить перелік внутрішніх дій або *діяльностей*, які виконуються в процесі знаходження елемента в даному стані. Кожне з дій записується у вигляді окремого рядка й має наступний формат:

< мітка-дії '/' вираження - дії>

Мітка дії вказує на обставини або умови, при яких буде виконуватися діяльність, певна вираженням дії. При цьому *вираження дії* може використовувати будь-які атрибути й зв'язки, які належать області імен або контексту об'єкта.

Перелік міток дії має фіксовані значення в мові UML, які не можуть бути використані в якості імен подій. Ці значення наступні:

- entry - ця мітка вказує на дію, специфікована наступним за нею вираженням дії, яка виконується в момент входу в даний стан (вхідна дія);
- exit - ця мітка вказує на дію, специфікована наступним за нею вираженням дії, яка виконується в момент виходу з даного стану (вихідна дія);
- do - ця мітка специфицирует діяльність, що виконується ("do activity"), яка виконується протягом усього часу, поки об'єкт перебуває в даному стані, або доти , поки не закінчиться обчислення, специфіковане наступним за нею вираженням дії.

В останньому випадку при завершенні події генерується відповідний результат;

- include - ця мітка використовується для звертання до під-автомату, при цьому наступне за нею вираження дії містить ім'я цього під-автомату.

Як приклад стану розглянемо аутентифікацію клієнта для доступу до банкомату. Список внутрішніх дій у даному стані може включати наступні дії: перша **дія вхідна** – вона виконується при вході в цей стан і пов'язана з одержанням рядка символів, що відповідають PIN-коду клієнта. Далі виконується діяльність по перевірці введеного клієнтом PIN-коду. При успішному завершенні цієї перевірки виконується **дія на виході**, що відображає меню доступних для клієнта опцій.

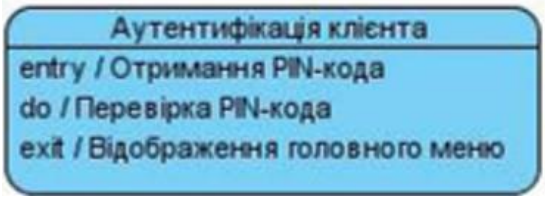


Рисунок 3 - Приклад стану з непустою секцією внутрішніх дій

Початковий стан

Початковий стан являє собою окремий випадок стану, який не містить ніяких внутрішніх дій (псевдостан). У цьому стані перебуває об'єкт за замовчуванням у початковий момент часу. Воно служить для вказівки на діаграмі станів графічної області, від якої починається процес зміни станів.

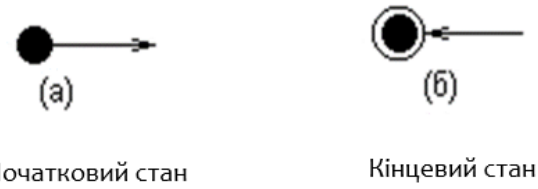


Рисунок 4 – Початковий та кінцевий стани

Кінцевий стан

Кінцевий (фінальне) стан являє собою окремий випадок стану, який також не містить ніяких внутрішніх дій (псевдостан). У цьому стані буде перебувати об'єкт за замовчуванням після завершення роботи автомата в кінцевий момент часу. Воно служить для вказівки на діаграмі станів графічної області, у якій завершується процес зміни станів або життєвий цикл даного об'єкта.

Перехід

Простий перехід (simple transition) являє собою відношення між двома послідовними станами, яке вказує на факт зміни одного стану іншим. Перебування об'єкта в першому стані може супроводжуватися виконанням деяких дій, а перехід у другий стан буде можливий після завершення цих дій, а також після задоволення

деяких додаткових умов. У цьому випадку говорять, що перехід спрацьовує, Або відбувається спрацьовування переходу. До спрацьовування переходу об'єкт перебуває в попередньому від нього стані, називаним вихідним станом, або в джерелі, а після його спрацьовування об'єкт перебуває наступному від нього стані (цільовому стані).

Перехід здійснюється при настанні деякої події: закінчення виконання діяльності (do activity), одержанні об'єктом повідомлення або прийманням сигналу. Спрацьовування переходу може залежати не тільки від настання деякої події, але й від виконання певного умови, називаної *сторожовою умовою*. Об'єкт перейде з одного стану в інше в тому випадку, якщо відбулася зазначена подія й сторожова умова прийняла значення "істина".

На діаграмі станів перехід зображується суцільною лінією зі стрілкою, яка спрямована в цільовий стан (наприклад, "вихід з ладу" на мал. 1). Кожний перехід може бути позначений рядком тексту, який має наступний загальний формат:

<сигнатура події>'['<сторожова умова>']' <вираження дії>.

При цьому сигнатура події описує деяка подія з необхідними аргументами:

<ім'я події>'('<список параметрів, розділених комами>')'.

Подія

Подія являє собою специфікацію деякого факту, що має місце в просторі й у часі. Про події говорять, що вони "відбуваються", при цьому окремі події повинні бути впорядковані в часі. Після настання деякої події не можна вже повернутися до попередніх подій, якщо така можливість не передбачена явно в моделі.

Семантика поняття події фіксує увагу на зовнішніх проявах якісних змін, що відбуваються при переході моделюємого об'єкта зі стану в стан. Наприклад, при включенні електричного перемикача відбувається деяка подію, у результаті якого кімната стає освітленою. Після успішного ремонту комп'ютера також відбувається немаловажна подія - відновлення його працездатності. Якщо підняти трубку звичайного телефону, те, у випадку його справності, ми очікуємо почути тоновий сигнал. І цей факт теж є подією.

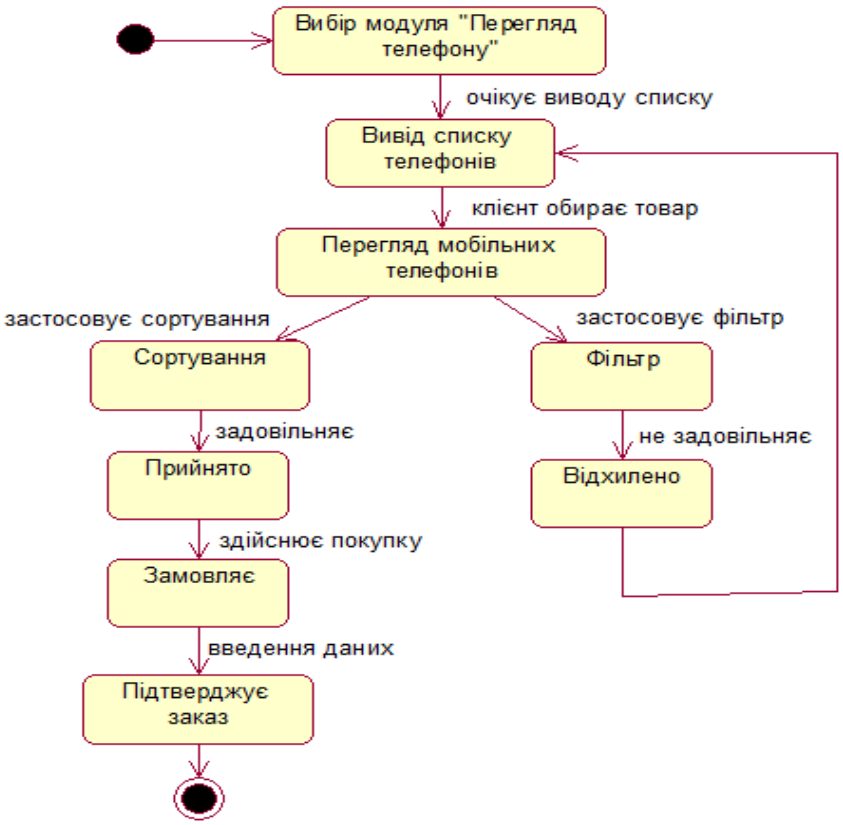
У мові UML події відіграють роль стимулів, які ініціюють переходи з одних станів в інші. У якості подій можна розглядати сигнали, виклики, закінчення фіксованих проміжків часу або моменти закінчення виконання певних дій. Перехід *тригерний*, тобто специфікований подією-тригером. Наприклад, переходи на рис. 1 є тригерними, оскільки з кожним з них зв'язана деяка подія-тригер, що відбувається асинхронно в момент виходу з ладу технічного обладнання або в момент закінчення його ремонту.

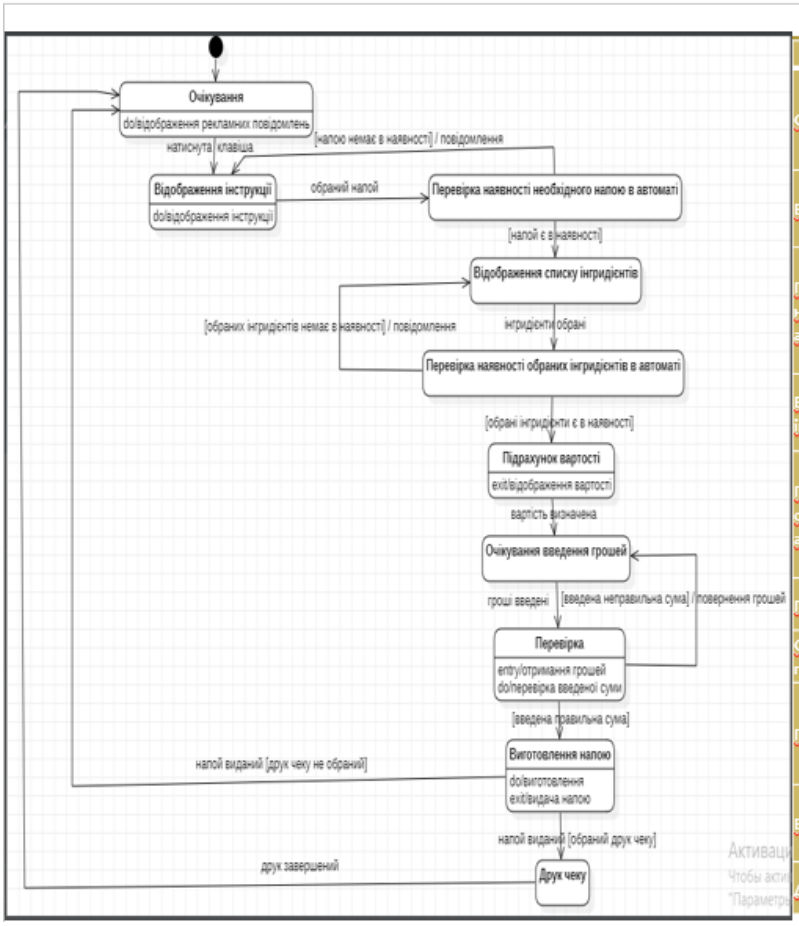
Якщо поруч зі стрілкою переходу не зазначений ніякий рядок тексту, то відповідний перехід є не тригерним, і в цьому випадку з контексту діаграми станів повинне бути ясно, після закінчення якої діяльності він спрацьовує.

Сторожова умова

Сторожова умова (guard condition), якщо вона є, завжди записується в прямих дужках після події-тригера і являє собою деякий булевський вираз. Нагадаємо, що булевское вираження повинне ухвалювати одне їх двох значень, що взаємно виключають: "істина" або "неправда".

Діаграма станів «Моделювання роботи ІС для магазину мобільних телефонів»





Стан	Опис стану
Очікування	Автомат по виготовленню кави знаходиться в режимі очікування вхідних впливів. В цей час автомат відображає рекламні повідомлення.
Відображення інструкції	Після натискання будь-якої із клавіш автомат знаходиться в режимі відображення інструкції.
Перевірка наявності необхідного напою в автоматі	Автомат знаходиться в режимі перевірки наявності обраного напою. У випадку його відсутності автомат видає відповідне повідомлення та переходить у стан відображення інструкції.
Відображення списку інгридієнтів	Автомат відображає список інгридієнтів, які можна додати до обраного напою.
Перевірка наявності обраних інгридієнтів в автоматі	Автомат перевіряє наявність обраних інгридієнтів. В випадку їх відсутності автомат видає відповідне повідомлення та переходить в стан відображення списку інгридієнтів.
Підрахунок вартості	Автомат визначає вартість напою та відображає її на екрані
Очікування введення грошей	Автомат очікує вводу грошей в відповідний приймочний пристрій
Перевірка	Автомат порівнює отриману суму з потрібною. Якщо сума неправильна, автомат повертає гроші та переходить в стан очікування введення грошей
Виготовлення напою	Якщо введена правильна сума, автомат починає готувати обраний напій та потім видає його людині
Друк чеку	Автомат друкує чек, якщо людина обрала друк чеку

Тема: Діаграми діяльності.

Мета: Розглянути основні поняття діаграми діяльності. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.
II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.
III. Виклад нового матеріалу:

План:

1. Стан дії;
2. Переходи;
3. Доріжки;
4. Об’єкти;

IV. Узагальнення та систематизація знань;
V. Підведення підсумків занять;

VI. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. В чому різниця діаграми діяльності від діаграми станів?
2. Які елементи використовуються на діаграмі діяльності?
3. Що таке діяльність?
4. Що таке стан дії?
5. Які переходи використовуються на цих діаграмах?
6. Для чого необхідні доріжки?
7. Що відображають об’єкти?
8. Як відобразити паралельні дії?

При моделюванні поведінки проектованої або аналізованої системи виникає необхідність не тільки представити процес зміни її станів, але й деталізувати особливості алгоритмічної й логічної реалізації виконуваних системою операцій. Традиційно для цієї мети використовувалися блок-схеми або структурні схеми алгоритмів. Кожна така схема акцентує увагу на послідовності виконання певних дій або елементарних операцій, які в сукупності приводять до одержання бажаного результату.

Важливо підкреслити та обставина, що зі збільшенням складності системи строге дотримання послідовності виконуваних операцій здобуває усе більш важливе значення. Якщо спробувати заварити кава холодною водою, то ми можемо

тільки зіпсувати одну порцію напою. Порушення послідовності операцій при ремонті двигуна може привести до його поломки або виходу з ладу.

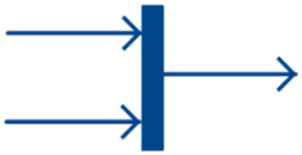
Для моделювання процесу виконання операцій у мові UML використовуються так звані *діаграми діяльності*. Застосовувана в них графічна нотація багато в чому схожа на нотацію діаграми станів, оскільки на діаграмах діяльності також присутні позначення станів і переходів. Відмінність полягає в семантиці станів, які використовуються для вистави не діяльностей, а дій, і у відсутності на переходах сигнатури подій. Кожний стан на діаграмі діяльності відповідає виконанню деякої елементарної операції, а перехід у наступний стан спрацьовує тільки при завершенні цієї, операції в попередньому стані. Графічно діаграма діяльності представляється у формі графа діяльності, вершинами якого є стани дії, а дугами - переходи від одного стану дії до іншого.

Таким чином, діаграми діяльності можна вважати часткам случаємо діаграм станів. Саме вони дозволяють реалізувати в мові UML особливості процедурного й синхронного керування, обумовленого завершенням внутрішніх діяльностей і дій. Метамодель UML надає для цього необхідні терміни й семантику. Основним напрямком використання діаграм діяльності є візуалізація особливостей реалізації операцій класів, коли необхідно представити алгоритми їх виконання. При цьому кожний стан може бути виконанням операції деякого класу або її частини, дозволяючи використовувати діаграми діяльності для опису реакцій на внутрішні події системи.

У контексті мови UML *діяльність (activity)* являє собою деяку сукупність окремих обчислень, виконуваних автоматом. При цьому окремі елементарні обчислення можуть приводити до деякого результату або дії (action). На діаграмі діяльності відображається логіка або послідовність переходу від однієї діяльності до іншої, при цьому увага фіксується на результаті діяльності. Сам же результат може привести до зміни стану системи або поверненню деякого значення.

Позначення діаграми діяльності

Символ	Ім'я	Використання
●	Початковий вузол	Відправна точка, або початковий стан
	Дія	Представлення діяльності, завдання для виконання
	Потік керування	Спрямований потік, контрольний потік діяльності
	Кінцевий вузол активності	Кінцевий стан, завершення усіх потоків процесу
	Кінцевий вузол потоку	Кінець одного потоку
	Вузол прийняття рішення	Розгалуження з умовою та кількома варіантами дій. Має один вхід декілька виходів

	Вузол злиття	Об'єднання потоків створених вузлом прийняття рішень. Має кілька входів і один вихід
	Вилка	Розподілення потоку на кілька паралельних без прийняття рішення
	Злиття	Об'єднання декількох паралельних потоків
	Надсилання сигналу	Вказує на те що сигнал надсилається на приймальну діяльність
	Отримання сигналу	Вказує на отримання сигналу
	Коментар	Дозволяє робити коментарі до діаграми. З'єднується пунктирною лінією

Стан дії

Стан дії (action state) є спеціальним випадком стану з деякою вхідною дією й принаймні одним вихідним зі стану переходом. Цей перехід неявно припускає, що вхідна дія вже завершилася. Стан дії не може мати внутрішніх переходів, оскільки воно є елементарним. Звичайне використання стану дії полягає в моделюванні одного кроку виконання алгоритму (процедури) або потоку керування.

Графічно стан дії зображується фігурою, що нагадує прямокутник, бічні сторони якого замінені опуклими дугами (рис. 1). Усередині цієї фігури записується вираження дії (action-expression), яка повинна є унікальним у межах однієї діаграми діяльності.

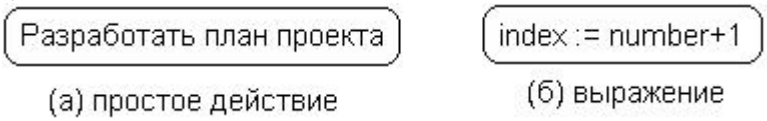


Рисунок 1 – Графічне зображення стану дії

Дія може бути записане природньою мовою, деякому псевдокоді або мові програмування. Ніяких додаткових або неявних обмежень при записі дій не накладається.

Іноді виникає необхідність представити на діаграмі діяльності деяка складна дія, яка, у свою чергу, складається з декількох більш простих дій. У цьому випадку можна використовувати спеціальне позначення так званого стану під-діяльності (subactivity state). Такий стан є графом діяльності й позначається спеціальною піктограмою в правому нижньому куті символу стану дії (рис. 2). Ця конструкція

може застосовуватися до будь-якого елемента мови UML, який підтримує «вкладеність» своєї структури. При цьому піктограма може бути додатково позначена типом вкладеної структури.

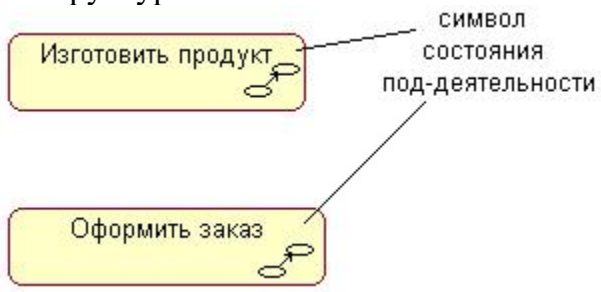


Рисунок 2 - Графічне зображення стану під-діяльності

Кожна діаграма діяльності повинна мати єдиний початковий і єдиний кінцевий стану. Вони мають такі ж позначення, як і на діаграмі станів. При цьому кожна діяльність починається в початковому стані й закінчується в кінцевому стані. Саму діаграму діяльності прийнято мати у своєму розпорядженні такий образ, щоб дії випливали зверху вниз. У цьому випадку початковий стан буде зображуватися у верхній частині діаграми, а кінцеве - у її нижній частині.

Переходи

При побудові діаграми діяльності використовуються тільки *не тригерні переходи,* тобто такі, які спрацьовують відразу після завершення діяльності або виконання відповідного дії. Цей перехід переводить діяльність у наступний стан відразу, як тільки закінчиться дія в попередньому стані. На діаграмі такий перехід зображується суцільною лінією зі стрілкою.

Якщо зі стану дії виходить єдиний перехід, то він може бути ніяк не позначений. Якщо ж таких переходів кілька, то спрацювати може тільки один з них. Саме в цьому випадку для кожного з таких переходів повинне бути явно записана *сторожова умова* в прямих дужках. При цьому для всіх вихідних з деякого стану переходів повинне виконуватися вимога істинності тільки одного з них. Подібний випадок зустрічається тоді, коли послідовно виконувана діяльність повинна розділитися на альтернативні галузі залежно від значення деякого проміжного результату. Така ситуація одержала назву *розгалуження*, а для її позначення застосовується спеціальний символ.

Графічно розгалуження на діаграмі діяльності позначається невеликим ромбом, усередині якого немає ніякого тексту (рис. 3). У цей ромб може входити тільки одна стрілка від того стану дії, після виконання якого потік керування повинен бути продовжений по одній з галузей, що взаємно виключають. Прийняте вхідну стрілку приєднувати до верхньої або лівій вершині символу розгалуження. Вихідних стрілок може бути дві або більше, але для кожної з них явно вказується відповідна сторожова умова у формі булевского вираження.

Як приклад розглянемо фрагмент відомого алгоритму знаходження корнів квадратного рівняння. У загальному випадку після приведення рівняння другому ступеня до канонічного виду: a*x*x+в*x+c=0 необхідно обчислити його дискримінант. Причому, у випадку негативного дискримінанта рівняння не має розв'язку на безлічі дійсних чисел, і подальші обчислення повинні бути припинені. При ненегативному дискримінанті рівняння має розв'язок, коріння якого можуть бути отримані на основі конкретної розрахункової формули.

Графічно фрагмент процедури обчислення корінь квадратного рівняння може бути представлений у вигляді діаграми діяльності із трьома станами дії й розгалуженням (рис. 3). Кожний з переходів, що виходять зі стану "Обчислити дискримінант", має сторожову умову, що визначає єдину галузі, по якій може бути продовжений процес обчислення корінь залежно від знаку дискримінанта. Очевидно, що у випадку його заперечності, ми відразу попадаємо в кінцевий стан, тим самим завершуючи виконання алгоритму в цілому.

Рисунок 3 - Фрагмент діаграми діяльності для алгоритму знаходження кореня квадратного рівняння

У розглянутому прикладі, як видне з рис. 3, виконувані дії з'єднуються в кінцевому стані. Однак це зовсім не є обов'язковим. Можна зобразити ще один символ розгалуження, який буде мати кілька вхідних переходів і один вихідний.

У наступному прикладі (рис. 4) розраховується загальна вартість товарів, що купуються по кредитній картці в супермаркеті. Якщо ця вартість перевищує $50, то виконується аутентифікація особистості власника картки. У випадку позитивної перевірки (картка дійсна) або якщо вартість товарів не перевищує $50, відбувається зняття суми з рахунку й оплата вартості товарів. При негативному результаті (картка недійсна) оплати не відбувається, і товар залишається в продавця.

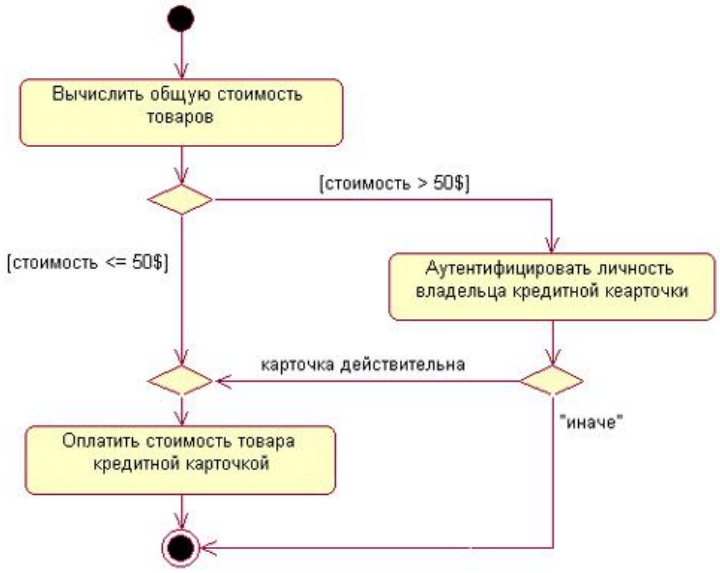


Рисунок 4 - Різні варіанти розгалужень на діаграмі діяльності

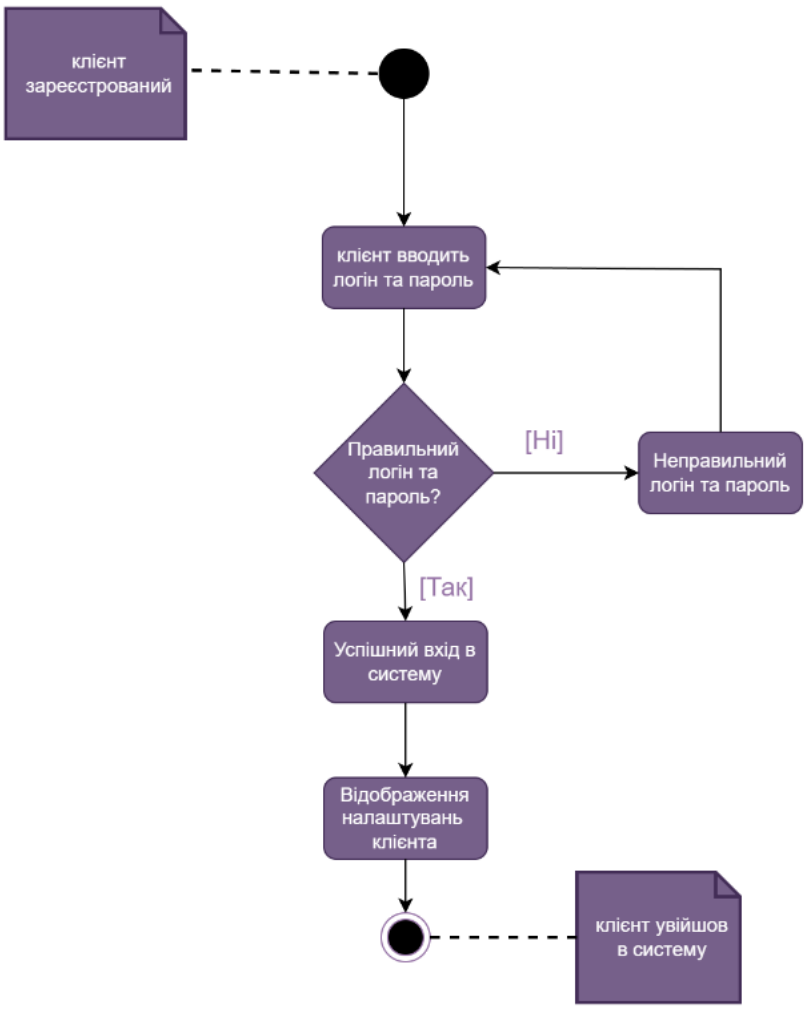


Рисунок 5 - Діаграма входу в систему

Один з найбільш значимих недоліків звичайних блок-схем або структурних схем алгоритмів пов'язаний із проблемою зображення паралельних галузей окремих обчислень. Оскільки розпаралелювання обчислень суттєво підвищує

загальна швидкодія програмних систем, необхідні графічні примітиви для вистави паралельних процесів. У мові UML для цієї мети використовується спеціальний символ для поділу й злиття паралельних обчислень або потоків керування. Таким символом є пряма риска, аналогічно позначенню переходу у формалізмі мереж Петри.

Як правило, така риска зображується відрізком горизонтальної лінії, товщина якої трохи ширше основних суцільних ліній діаграми діяльності. При цьому поділ (concurrent fork) має один вхідний перехід і кілька вихідних (рис. 6, а). Злиття (concurrent join), навпаки, має кілька вхідних переходів і один вихідний (рис. 6, б).

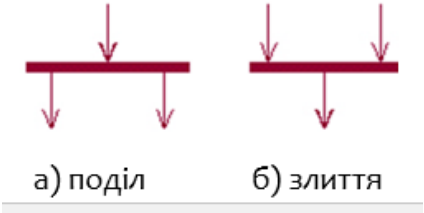


Рисунок 6 - Графічне зображення поділу й злиття паралельних потоків керування

Доріжки

Діаграми діяльності можуть бути використані не тільки для специфікації алгоритмів обчислень або потоків керування в програмних системах. Не менш важлива область їх застосування пов'язана з моделюванням бізнес-процесів. Дійсно, діяльність будь-якої компанії (фірми) також являє собою не що інше, як сукупність окремих дій, спрямованих на досягнення необхідного результату. Однак стосовно до бізнес-процесам бажане виконання кожної дії асоціювати з конкретним підрозділом компанії. У цьому випадку підрозділ відповідає за реалізацію окремих дій, а сам бізнес-процес представляється у вигляді переходів дій з одного підрозділу до іншого.

Для моделювання цих особливостей у мові UML використовується спеціальна конструкція, що одержала назву *доріжки* (swimlanes). Мається на увазі візуальна аналогія із плавальними доріжками в басейні, якщо дивитися на відповідну діаграму. При цьому всі стани дії на діаграмі діяльності діляться на окремі групи, які відділяються друг від друга вертикальними лініями. Дві сусідні лінії й утворюють доріжку, а група станів між цими лініями виконується окремим підрозділом (відділом, групою, відділенням, філією) компанії (рис. 7).

Назви підрозділів явно вказуються у верхній частині доріжки. Перетинати лінію доріжки можуть тільки переходи, які в цьому випадку позначають вихід або вхід потоку керування у відповідний підрозділ компанії. Порядок проходження доріжок не несе якої-небудь семантичної інформації й визначається міркуваннями зручності.

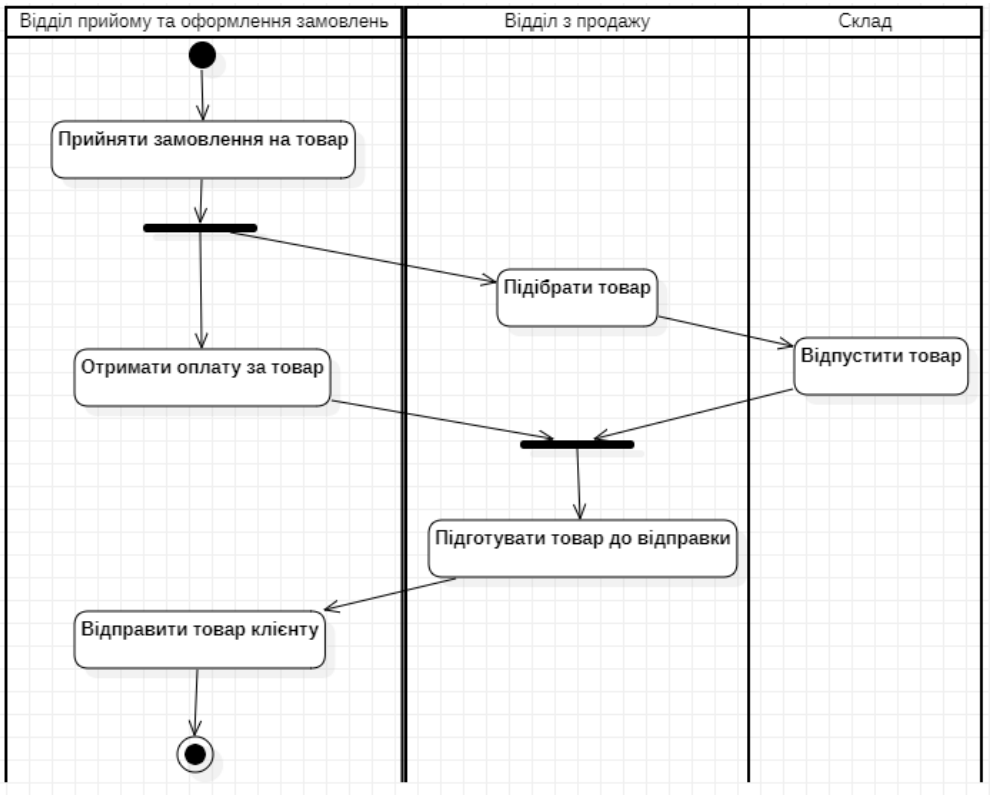


Рисунок 7 - Варіант діаграми діяльності з доріжками

Об'єкти

У загальному випадку дії на діаграмі діяльності виконуються над тими або іншими об'єктами. Ці об'єкти або ініціюють виконання дій, або визначають деякий результат цих дій. При цьому дії специфікують виклики, які передаються від одного об'єкта графа діяльності до іншого. Оскільки в такому ракурсі об'єкти відіграють певну роль у розумінні процесу діяльності, іноді виникає необхідність явно вказати їх на діаграмі діяльності.

Для графічної вистави об'єктів використовуються прямокутник класу, з тим відмінністю, що ім'я об'єкта підкреслюється. Далі після імені може вказуватися характеристика стану об'єкта в прямих дужках. Такі прямокутники об'єктів приєднуються до станів дії відношенням залежності пунктирною лінією зі стрілкою. Відповідна залежність визначає стан конкретного об'єкта після виконання попереднього дії.

На діаграмі діяльності з доріжками розташування об'єкта може мати деякий додатковий зміст. А саме, якщо об'єкт розташовано на границі двох доріжок, те це може означати, що перехід до наступного стану дії в сусідній доріжці асоційований з готовністю деякого документа (об'єкт у деякому стані). Якщо ж об'єкт цілком розташований усередині доріжки, то й стан цього об'єкта цілком визначається діями даної доріжки.

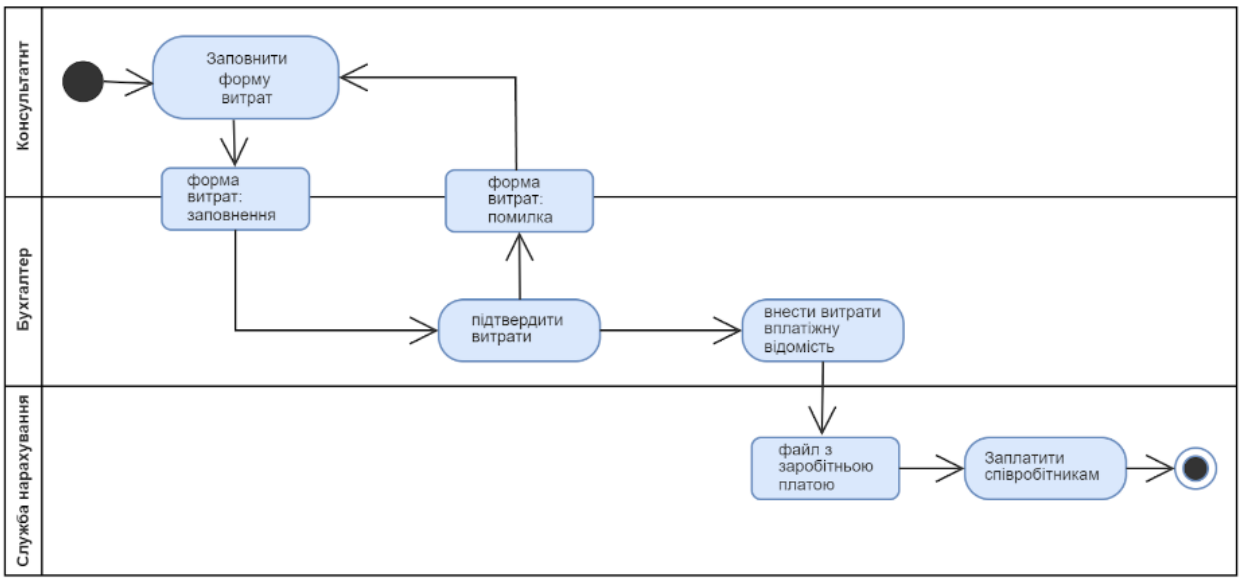


Рисунок 8 - Діаграма виплати заробітної плати

Тема: Діаграма послідовностей

Мета: Розглянути основні поняття діаграми послідовностей. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;
   c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграма послідовності
   a. Об’єкти;
   b. Лінія життя об'єкта;
   c. Фокус керування;
   d. Повідомлення;
   e. Розгалуження потоку керування.

IV. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

Контрольні питання:

1. Чим відрізняється процедурний потік від асинхронного потоку повідомлень?
2. Як вказується повторення повідомлень?
3. Як показати розгалуження повідомлень?
4. Чим відрізняється процедурний потік від асинхронного потоку повідомлень?
5. Як вказується повторення повідомлень?
6. Як відображається порядок передачі повідомлень у діаграмі послідовності?
7. Коли зручніше застосовувати діаграми послідовності?

Діаграма послідовності - другий різновид діаграм взаємодії. Відбиваючи сценарій поведінки в системі, ця діаграма забезпечує більш наочне представлення порядку передачі повідомлень. Правда, вона не дозволяє показати такі деталі, які видні на діаграмі співробітництва (структурні характеристики об'єктів і зв'язків).

Об'єкти

На діаграмі послідовності зображуються винятково ті об'єкти, які безпосередньо беруть участь у взаємодії й не показуються можливі статичні асоціації з іншими об'єктами. Для діаграми послідовності ключовим моментом є саме динаміка взаємодії об'єктів у часі. При цьому діаграма послідовності має як би два виміри.

Перший вимір - ліворуч праворуч у вигляді вертикальних ліній, кожна з яких зображує *лінію життя* окремого об'єкта, що бере участь у взаємодії. Графічно кожний об'єкт зображується прямокутником і розташовується у верхній частині своєї лінії життя (рис. 1). Усередині прямокутника записуються ім'я об'єкта й ім'я класу, розділені двокрапкою. При цьому весь запис підкреслюється, що є ознакою об'єкта, яка, як відомо, являє собою екземпляр класу.

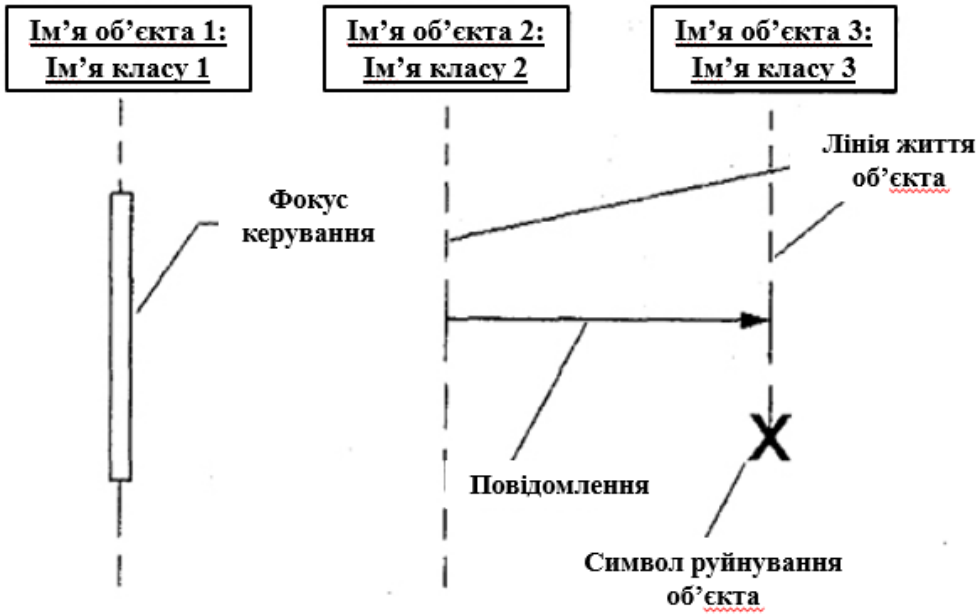


Рисунок 1 - Різні графічні примітиви діаграми послідовності

Крайнім ліворуч на діаграмі зображується об'єкт, який є ініціатором взаємодії (об'єкт 1 на рис. 1). Праворуч зображується інший об'єкт, який безпосередньо

взаємодіє з першим. Таким чином, усі об'єкти на діаграмі послідовності утворюють деякий порядок, обумовлений ступенем активності цих об'єктів при взаємодії один з одним.

Другий вимір діаграми послідовності - вертикальна часова вісь, спрямована зверху вниз. Початковому моменту часу відповідає сама верхня частина діаграми. При цьому взаємодії об'єктів реалізуються за допомогою повідомлень, які посилають одними об'єктами іншим. Повідомлення зображуються у вигляді горизонтальних стрілок з іменем повідомлення й також утворюють порядок за часом свого виникнення. Інакше кажучи, повідомлення, розташовані на діаграмі послідовності вище, ініціюються раніше тих, які розташовані нижче. При цьому масштаб на осі часу не вказується, оскільки діаграма послідовності моделює лише тимчасову впорядкованість взаємодій типу " раніше-пізніше".

Лінія життя об'єкта

Лінія життя об'єкта (object lifeline) зображується пунктирною вертикальною лінією, асоційованої з єдиним об'єктом на діаграмі послідовності. Лінія життя служить для позначення періоду часу, протягом якого об'єкт існує в системі й, отже, може потенційно брати участь у всіх її взаємодіях. Якщо об'єкт існує в системі постійно, то і його лінія життя повинна тривати по всій площині діаграми послідовності від самої верхньої її частини до самої нижньої (об'єкти 1 і 2 на рис. 1).

Окремі об'єкти, виконавши свою роль у системі, можуть бути знищені (зруйновані), щоб звільнити займані ними ресурси. Для таких об'єктів лінія життя обривається в момент його знищення. Для позначення моменту знищення об'єкта в мові UML використовується спеціальний символ у формі латинської букви "X" (об'єкт 3 на рис. 1). Нижче цього символу пунктирна лінія не зображується, оскільки відповідного об'єкта в системі вже ні, і цей об'єкт повинен бути виключений із усіх наступних взаємодій.

Зовсім не обов'язково створювати всі об'єкти в початковий момент часу. Окремі об'єкти в системі можуть створюватися в міру необхідності, суттєво заощаджуючи ресурси системи й підвищуючи її продуктивність. У цьому випадку прямокутник такого об'єкта зображується не у верхній частині діаграми послідовності, а в тій її частині, яка відповідає моменту створення об'єкта (об'єкт 6 на рис. 2). При цьому прямокутник об'єкта вертикально розташовується в тому місці діаграми, яке по осі часу збігається з моментом його виникнення в системі. Очевидно, об'єкт обов'язково створюється зі своєю лінією життя й, можливо, з фокусом керування.

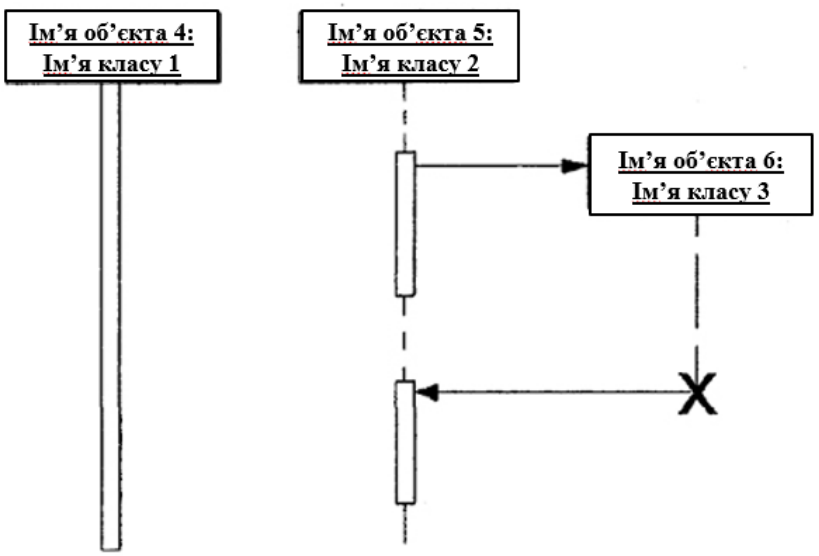


Рисунок 2 - Графічне зображення різних варіантів ліній життя й фокусів керування об'єктів

Фокус керування

У процесі функціонування об'єктно-орієнтованих систем одні об'єкти можуть перебувати в активному стані, безпосередньо виконуючи певні дії або в стані пасивного очікування повідомлень від інших об'єктів. Щоб явно виділити подібну активність об'єктів, у мові UML застосовується спеціальне поняття, що одержала назва фокуса керування (focus of control). Фокус керування зображується у формі витягнутого вузького прямокутника (див. об'єкт 1 на рис. 1), верхня сторона якого позначає початок одержання фокуса управління об'єкта (початок активності), а її нижня сторона - закінчення фокуса керування (закінчення активності). Цей прямокутник розташовується нижче позначення відповідного об'єкта й може заміняти його лінію життя (об'єкт 4 на рис. 2), якщо на всьому її протязі він є активним.

З іншого боку, періоди активності об'єкта можуть чергуватися з періодами його пасивності або очікування. У цьому випадку в такого об'єкта є кілька фокусів керування (об'єкт 5 на рис. 2). Важливо розуміти, що одержати фокус керування може тільки існуючий об'єкт, у якого в цей момент є лінія життя. Якщо ж деякий об'єкт був знищений, то знову виникнути в системі він уже не може. Замість нього лише може бути створений інший екземпляр цього ж класу, який, строго говорячи, буде іншим об'єктом.

В окремих випадках ініціатором взаємодії в системі може бути актор або зовнішній користувач. У цьому випадку актор зображується на діаграмі послідовності найпершим об'єктом ліворуч зі своїм фокусом керування (рис. 3). Найчастіше актор і його фокус керування будуть існувати в системі постійно, відзначаючи характерну для користувача активність в ініціюванні взаємодій із системою. При цьому сам актор може мати власне ім'я або залишатися анонімним.

Іноді деякий об'єкт може ініціювати рекурсивна взаємодія із самим собою. Мова йде про те, що наявність у багатьох мовах програмування спеціальних засобів побудови рекурсивних процедур вимагає візуалізації відповідних понять у формі графічних примітивів. На діаграмі послідовності рекурсія позначається невеликим прямокутником, приєднаним до правої сторони фокуса керування того об'єкта, для якого зображується ця рекурсивна взаємодія (об'єкт 7 на рис. 3).

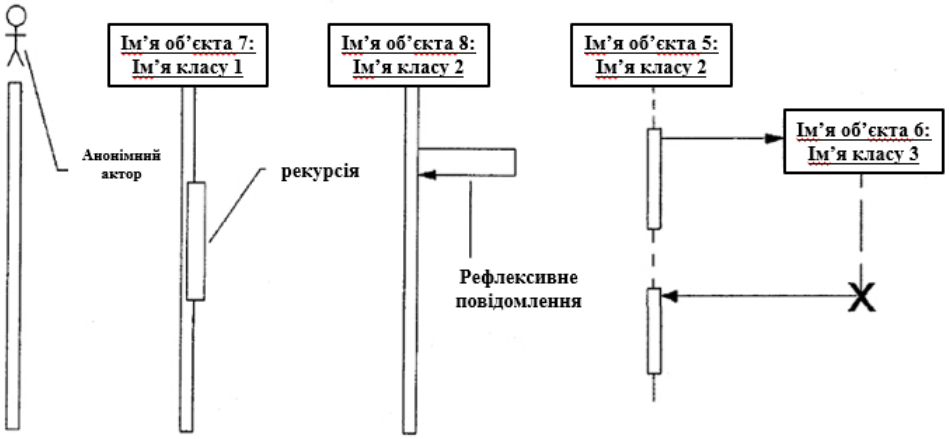


Рисунок 3 - Графічне зображення актора, рекурсії й рефлексивного повідомлення на діаграмі послідовності

Повідомлення

Як було відзначено вище, мета взаємодії в контексті мови UML полягає в тому, щоб специфікувати комунікацію між безліччю взаємодіючих об'єктів. Кожна взаємодія описується сукупністю повідомлень, якими об'єкти обмінюються між собою. У цьому змісті повідомлення (message) являє собою закінчений фрагмент інформації, який відправляється одним об'єктом іншому. При цьому приймання повідомлення ініціює виконання певних дій, спрямованих на розв'язок окремого завдання тим об'єктом, якому це повідомлення відправлене.

Таким чином, повідомлення не тільки передають деяку інформацію, але й вимагають або припускають від об'єкта, що ухвалює, виконання очікуваних дій.

Повідомлення можуть ініціювати виконання операцій об'єктом відповідного класу, а параметри цих операцій передаються разом з повідомленням. На діаграмі послідовності всі повідомлення впорядковані за часом свого виникнення в системі, яка моделюється.

У такому контексті кожне повідомлення має напрямок від об'єкта, який ініціює й відправляє повідомлення, до об'єкта, який його одержує. Іноді відправника повідомлення називають клієнтом, а одержувача-сервером. При цьому повідомлення від клієнта має форму запиту деякого сервісу, а реакція сервера на запит після одержання повідомлення може бути пов'язана з виконанням певних дій або передачі клієнтові необхідної інформації теж у формі повідомлення.

У мові UML можуть зустрічатися кілька різновидів повідомлень, кожне з яких має своє графічне зображення.

Повідомлення бувають:

1. Синхронні, що очікують відповіді (у вигляді суцільної лінії зі стрілкою).
2. Асинхронні – не очікують відповіді (у вигляді пунктирної лінії зі стрілкою).
3. Рекурсивні – направлені до себе (починаються та завершуються на одній лінії життя).

Звичайно повідомлення зображуються горизонтальними стрілками, що з'єднують лінії життя або фокуси керування двох об'єктів на діаграмі послідовності. При цьому неявно передбачається, що час передачі повідомлення досить малий в порівнянні із процесами виконання дій об'єктами. Вважається також, що за час передачі повідомлення з відповідними об'єктами не може

відбутися ніяких подій. Інакше кажучи, стани об'єктів залишаються без зміни. Якщо ж це припущення не може бути визнане слушним, то стрілка повідомлення зображується під деяким нахилом, так щоб кінець стрілки розташовувався нижче її початку.

В окремих випадках об'єкт може посилати повідомлення самому собі, ініціюючи так звані рефлексивні повідомлення (об'єкт 8 на рис.3). Такі повідомлення зображуються прямокутником зі стрілкою, початок і кінець якої збігаються. Подібні ситуації виникають, наприклад, при обробці натискань на клавіші клавіатури при введенні тексту в документ, що редагується, при наборі цифр номера телефону абонента.

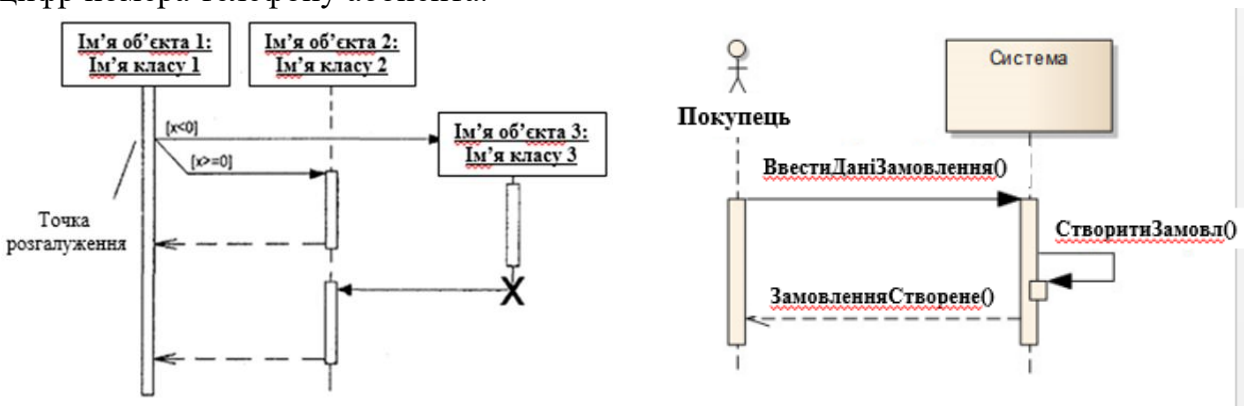


Рисунок 4 - Графічне зображення різних видів повідомлень між об'єктами на діаграмі послідовності

Таким чином, у мові UML кожне повідомлення асоціюється з деякою дією, яка повинна бути виконана його об'єктом, що прийняв. При цьому дія може мати деякі аргументи або параметри, залежно від конкретних значень яких може бути отриманий різний результат. Відповідні параметри буде мати й зухвала ця дія повідомлення. Більше того, значення параметрів окремих повідомлень можуть містити умовні вираження, утворюючи розгалуження або альтернативні шляхи основного потоку керування.

Розгалуження потоку керування

Для зображення розгалуження рисуються дві або більше стрілки, що виходять із однієї крапки фокуса керування об'єкта (фокус керування об'єкта 1 на рис. 5). При цьому відповідні умови повинні бути явно зазначені поруч із кожної зі стрілок у формі сторожової умови. Як неважко представити, якщо умова записана у формі булевського виразу, то розгалуження буде містити тільки дві галузі. У кожному разі умови повинні взаємно виключати одночасну передачу альтернативних повідомлень.

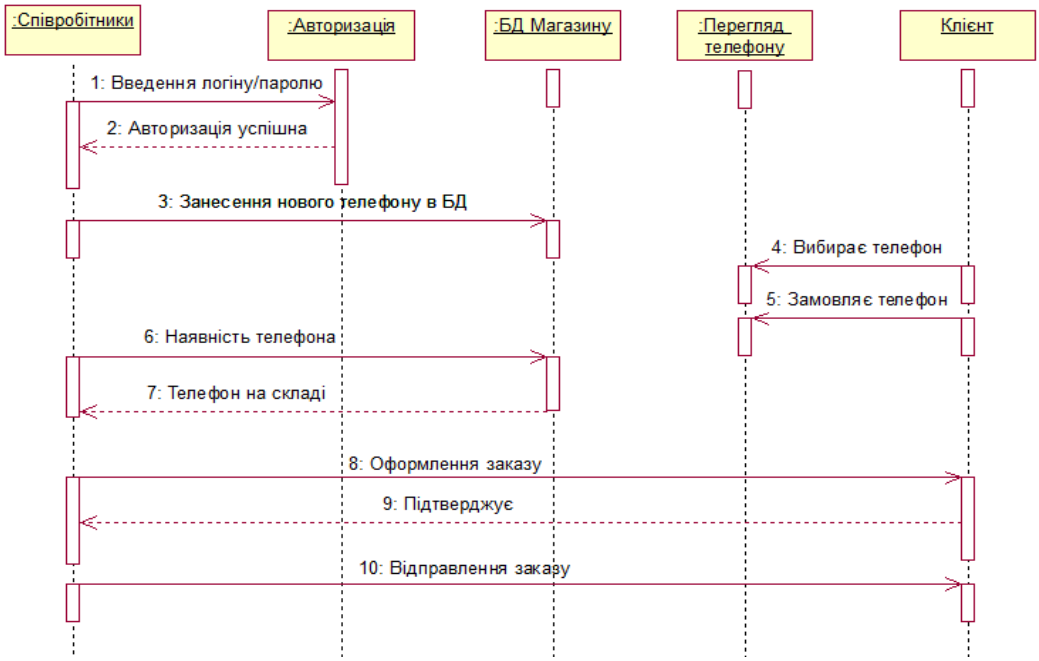


Рисунок 6 - Діаграма послідовності «Моделювання роботи інформаційної системи програмного забезпечення для магазину мобільних телефонів»

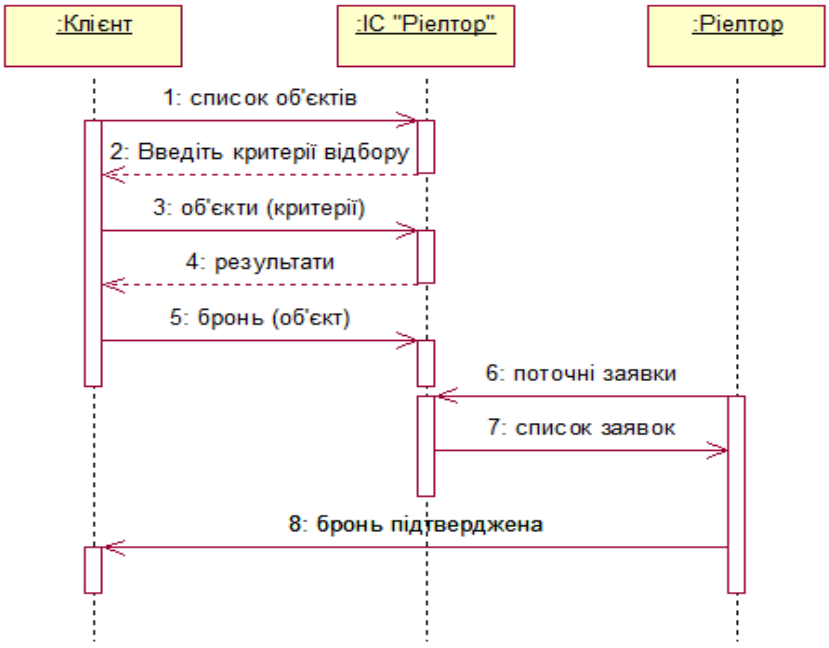


Рисунок 7 - Діаграма послідовності «Моделювання роботи ІС для керування роботою ріелтора»

Тема: Діаграма співпраці (кооперацій)

Мета: Розглянути основні поняття діаграми кооперацій. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   a. Готовність групи до заняття;
   b. Психоемоційний настрій;

c. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   a. Повідомлення теми та мети заняття;
   b. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграми кооперацій;
   a. Синтаксис імені об’єкта;
   b. Синтаксис властивості об’єкта;
   c. Повідомлення;

IV. Домашнє завдання:
   1. Ознайомитись з теоретичним матеріалом теми.
   2. Вивчити основні поняття лекції.
   3. Відповісти на контрольні питання.

   Контрольні питання:
   1. Як представляється ім'я об'єкта в діаграмі кооперацій?
   2. Що таке мультиоб’єкт на діаграмі кооперацій?
   3. Що таке зв’язок?
   4. Що загального в діаграмі послідовності й діаграмі співробітництва? Чим вони відрізняються друг від друга?

Діаграми взаємодії призначені для моделювання динамічних аспектів системи. Діаграма взаємодії показує взаємодію, що включає набір об'єктів і їх відносин, а повідомлення, що також пересилаються між об'єктами.

Існують два різновиди діаграми взаємодії - діаграма послідовності й діаграма кооперацій.

Діаграма послідовності - це діаграма взаємодії, яка виділяє впорядкування повідомлень за часом.

Діаграма кооперацій - це діаграма взаємодії, яка виділяє структурну організацію об'єктів, що посилають повідомлення, що її ухвалюють. Елементами діаграм взаємодії є учасники взаємодії - об'єкти, зв'язки, повідомлення.

Діаграми співробітництва (кооперації або співпраці) (collaboration diagram)

Подібно до діаграм послідовності, діаграми кооперації відображають потік подій у конкретному сценарії варіанта використання. Головна особливість діаграми кооперації полягає в можливості графічно уявити не тільки послідовність взаємодії, а й усі структурні відносини між об'єктами, що беруть участь у цій взаємодії.

Насамперед на діаграмі кооперації у вигляді прямокутників зображуються об'єкти, які беруть участь у взаємодії, що містять ім'я об'єкта, його клас і, можливо, значення атрибутів. Далі, як і на діаграмі класів, вказуються асоціації між об'єктами у вигляді різних сполучних ліній. При цьому можна явно вказати імена асоціації та ролей, які відіграють об'єкти в даній асоціації. Додатково можуть бути зображені динамічні зв'язки - потоки повідомлень. Вони подаються також у вигляді сполучних ліній між об'єктами, над якими розташовується стрілка із зазначенням

напрямку, імені повідомлення та порядкового номера в загальній послідовності ініціалізації повідомлень.

На відміну від діаграми послідовності, на діаграмі кооперації зображуються тільки відносини між об'єктами, які відіграють певні ролі у взаємодії, а послідовність взаємодій і паралельних потоків визначається за допомогою порядкових номерів.

Об'єкти діаграми кооперації

Окремі аспекти специфікації об'єктів як елементів діаграм уже розглядалися раніше під час опису діаграм послідовності. Ці ж об'єкти є основними елементами з яких будується діаграма кооперації. Для графічного зображення об'єктів використовується такий самий символ прямокутника, що й для класів.

Об'єкт є окремим екземпляром класу, який створюється на етапі виконання програми. Він може мати своє власне ім'я та конкретні значення атрибутів. Для позначення ролі класифікатора необхідно вказати або ім'я класу (разом із двокрапкою), або ім'я ролі (разом із похилою рискою). В іншому разі прямокутник відповідатиме звичайному класу. Якщо роль, яку має відігравати об'єкт, успадковується від кількох класів, то всі вони мають бути зазначені явно і розмежовуватися комою і двокрапкою.

Окремі приклади зображення об'єктів і класів на діаграмі кооперації наводяться на рис. 1. У першому випадку (рис. 1, а) позначено об'єкт з ім'ям «клієнт», що грає роль «ініціатор запиту». Далі (рис. 1, б) слідує позначення анонімного об'єкта, який відіграє роль ініціатора запиту. В обох випадках не вказано клас, на основі якого буде створено ці об'єкти. Позначення класу присутнє в наступному варіанті запису (рис. 1, в), причому об'єкт також анонімний.

Рис. 1 - Варіанти запису імен об'єктів, ролей і класів на діаграмах кооперації

Стосовно рівня специфікації на діаграмах кооперації можуть бути присутніми іменовані класи із зазначенням ролі класу в кооперації (рис. 1, г) або анонімні класи, коли зазначено тільки його роль (рис. 1, д). Останній випадок характерний для ситуації, коли в моделі можуть бути присутніми кілька класів з ім'ям «Клієнт», тому потрібно явно вказати ім'я відповідного пакету База даних (рис. 1, є).

Мультиоб'єкт (multiobject) являє собою цілу множину об'єктів на одному з кінців асоціації (рис. 2, а). На діаграмі кооперації мультиоб'єкт використовується для того, щоб показати операції та сигнали, адресовані всій безлічі об'єктів, а не тільки одному. При цьому стрілка повідомлення відноситься до всієї множини об'єктів, які позначають даний мультиоб'єкт. На діаграмі кооперації може бути

явно вказано відношення композиції між мультиоб'єктом і окремим об'єктом із його множини (рис. 2, б).

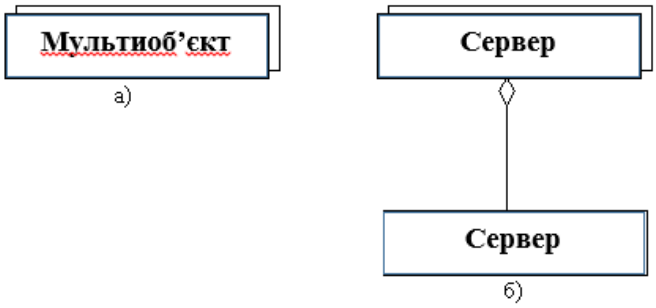


Рис. 2 - Графічне зображення мультиоб'єктів на діаграмі кооперації

У контексті мови UML усі об'єкти поділяються на дві категорії: пасивні та активні. Пасивний об'єкт оперує тільки даними і не може ініціювати діяльність з управління іншими об'єктами. Водночас пасивні об'єкти можуть посилати сигнали в процесі виконання запитів, які вони отримують.

Активний об'єкт має свою власну нитку управління і може ініціювати діяльність з управління іншими об'єктами. При цьому під ниткою розуміється потік керування, який може виконуватися паралельно з іншими обчислювальними нитками або нитками керування в межах одного обчислювального процесу.

У наведеному фрагменті діаграми кооперації (рис. 3) активний об'єкт "a: Абонент, що викликає" є ініціатором процесу встановлення з'єднання для обміну інформацією з іншим абонентом.

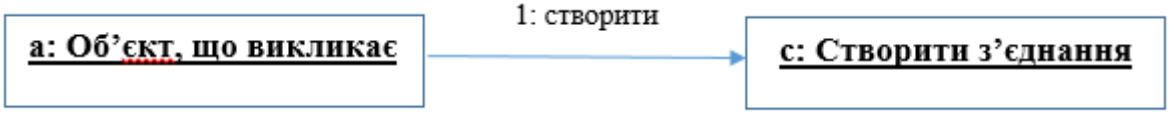


Рис. 3 - Активний об'єкт (ліворуч) на діаграмі кооперації

Складений об'єкт (composite object) або об'єкт-контейнер призначений для представлення об'єкта, що має власну структуру та внутрішні потоки (нитки) управління. Складений об'єкт є екземпляром складеного класу (класу-контейнера), який пов'язаний відношенням агрегації або композиції зі своїми частинами. Аналогічні відносини пов'язують між собою і відповідні об'єкти.

На діаграмах кооперації складовий об'єкт складається з двох секцій: верхньої та нижньої. У верхній секції записується ім'я складового об'єкта, а в нижній - його елементи (рис. 4), які можуть бути складовими об'єктами.

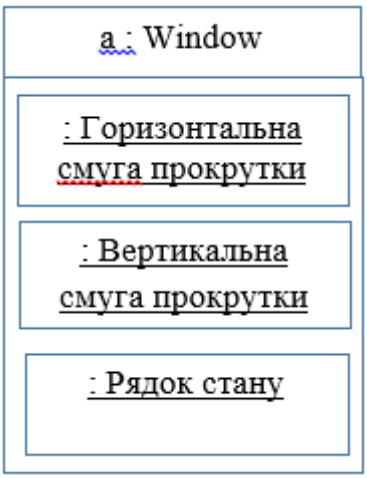


Рис. 4 - Складовий об'єкт на діаграмі кооперації

Зв'язок (link) є екземпляром або прикладом довільної асоціації. Зв'язок як елемент мови UML може мати місце між двома і більше об'єктами. Зв'язок на діаграмі кооперації зображується відрізком прямої лінії, що з'єднує два прямокутники об'єктів. На кожному з кінців цієї лінії можуть бути явно вказані імена ролей даної асоціації. Поруч із лінією в її середній частині може записуватися ім'я відповідної асоціації. Зв'язки не мають власних імен, оскільки повністю ідентичні як екземпляри асоціації. Для зв'язків не вказується також і кратність.

Стосовно діаграм кооперації повідомлення мають деякі додаткові семантичні особливості. Вони визначають комунікацію між двома об'єктами, один з яких передає іншому деяку інформацію. При цьому перший об'єкт очікує, що після отримання повідомлення другим об'єктом послідує виконання деякої дії. Таким чином, саме повідомлення є причиною або стимулом для початку виконання операцій, надсилання сигналів, створення і знищення окремих об'єктів. Зв'язок забезпечує канал для спрямованої передачі повідомлень між об'єктами від об'єкта-джерела до об'єкта-одержувача.

Приклад діаграми кооперації

На рис. 5 наведено діаграму кооперацій, що описує, як клієнт знімає з рахунку 20$.

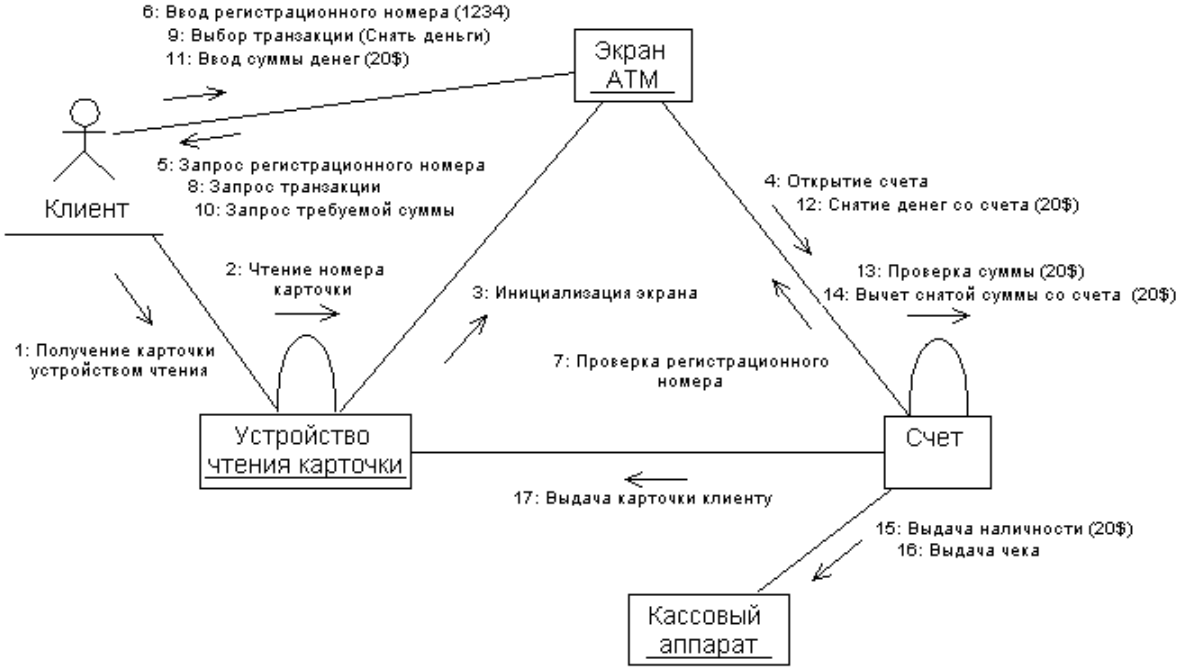


Рис. 5 - Діаграма кооперацій для зняття клієнтом 20$

З діаграми кооперацій легше зрозуміти потік подій і відносини між об'єктами, проте важче усвідомити послідовність подій, тому для сценарію створюють діаграми обох типів.

Тема: Діаграма компонентів

Мета: Розглянути основні поняття діаграми компонентів. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   1. Готовність групи до заняття;
   2. Психоемоційний настрій;
   3. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   1. Повідомлення теми та мети заняття;
   2. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграма компонентів
2. Елементи діаграми компонентів:
   - компоненти
   - інтерфейси
3. Різновиди компонентів

IV. Узагальнення та систематизація знань.

IV. Підведення підсумків заняття.

VI. Домашнє завдання:
1. Ознайомитись з теоретичними відомостями лекції
2. Вивчити основні поняття лекції.
3. Дати відповіді на контрольні питання.

Контрольні питання

1. Що відноситься до елементів компонентної діаграми?
2. Що таке компонент?
3. Як задається ім'я компонента?
4. Що таке інтерфейс?
5. Які стандартні стереотипи передбачені в UML для компонентів? Описати кожен стереотип.

Компонентна діаграма - перша із двох різновидів діаграм реалізації, що моделюють фізичні аспекти об'єктно-орієнтованих систем. Компонентна діаграма показує організацію набору компонентів і залежності між компонентами.

Елементами компонентних діаграм є компоненти й інтерфейси, а також відношення залежності й реалізації. Як і інші діаграми, компонентні діаграми можуть включати примітки й обмеження. Крім того, компонентні діаграми можуть містити пакети або підсистеми, використовувані для угруповання елементів моделі у великі фрагменти.

Компоненти

По своїй суті компонент є фізичним фрагментом реалізації системи, який містить у собі програмний код (вихідний, двійковий, що виконується), сценарні

описи або набори команд операційної системи (маються на увазі командні файли). Мова UML дає наступне визначення.

Компонент - фізична й замінна частина системи, яка відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів.

Інтерфейс - дуже важлива частина поняття "компонент". Графічно компонент зображується як прямокутник із вкладками, що звичайно включає ім'я (рис. 1.1).

Рисунок 1.1 - Позначення компонента

Як показано на рис. 1.2, за допомогою відношення залежності можна явно відобразити відношення між компонентом і класами, які він реалізує.

Рисунок 1.2 - Класи в компоненті

Інтерфейси

Інтерфейс - список операцій, які визначають послуги класу або компонента.

Компонування системи

За останні піввіку розроблювачі апаратури пройшли шлях від комп'ютерів розміром з кімнату до малюсіньких "ноутбуків", що забезпечили зрослі функціональні можливості. За ті ж піввіку розроблювачі програмного забезпечення пройшли шлях від великих систем на Асемблері й Фортрані до ще більших систем на C++ і Java. На жаль, але програмний інструментарій розбудовується повільніше, чим апаратний інструментарій.

Цей секрет - компонента. Розроблювач апаратури створює систему з готових апаратних компонентів (мікросхем), що виконують певні функції, що й надають набір послуг через ясні інтерфейси. Завдання конструкторів спрощується за рахунок повторного використання результатів, отриманих іншими.

Повторне використання - магістральний шлях розвитку програмного інструментарію. Створення нового ПЗ з існуючих, працездатних програмних компонентів приводить до більш надійного й дешевого коду. При цьому строки розробки суттєво скорочуються.

Основна мета програмних компонентів - допускати складання системи із двійкових замінних частин. Вони повинні забезпечити початкове створення системи з компонентів, а потім і її розвиток - додавання нових компонентів і заміну деяких старих компонентів без перебудови системи в цілому. Ключ до втілення

такої можливості - інтерфейси. Після того як інтерфейс визначений, до виконуваної системи можна підключити будь-який компонент, який задовольняє йому або забезпечує цей інтерфейс. Для розширення системи проводяться компоненти, які забезпечують додаткові послуги через нові інтерфейси. Такий підхід ґрунтується на особливостях компонента:

- Компонент фізичний. Він живе у світі бітів, а не логічних понять і не залежить від мови програмування
- Компонент - замінний елемент. Властивість заменяемості дозволяє замінити один компонент іншим компонентом, який задовольняє тим же інтерфейсам. Механізм заміни застережений сучасними компонентними моделями (COM, COM+, CORBA, Java Beans), що вимагають незначних перетворень або, що надають утиліти, які автоматизують механізм
- Компонент є частиною системи, він рідко автономний. Частіше компонент співпрацює з іншими компонентами й існує в архітектурній або технологічному середовищі, призначеної для його використання. Компонент зв'язаний і фізично, і логічно, він позначає фрагмент великої системи
- Компонент відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів

Різновиди компонентів

Мир сучасних компонентів досить широкий і різноманітний. У мові UML для позначення нових різновидів компонентів використовують механізм стереотипів. Стандартні стереотипи, передбачені в UML для компонентів:

"executable" - Компонент, який може виконуватися у фізичному вузлі (має розширення .exe)

"library" - Статична або динамічна об'єктна бібліотека (має розширення .dll)

"file" - Компонент, який представляє файл, що містить вихідний код або дані (має розширення .ini)

"table" - Компонент, який представляє таблицю бази даних (має розширення .tbl)

"document" - Компонент, який представляє документ (має розширення .hlp)

У мові UML не визначені піктограми для перерахованих стереотипів, застосовувані на практиці піктограми компонентів показані на рис. 1.3-1.7.

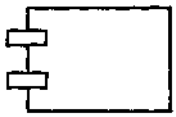


Рисунок 1.3 – Піктограма елемента що виконується.

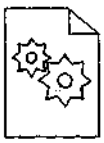


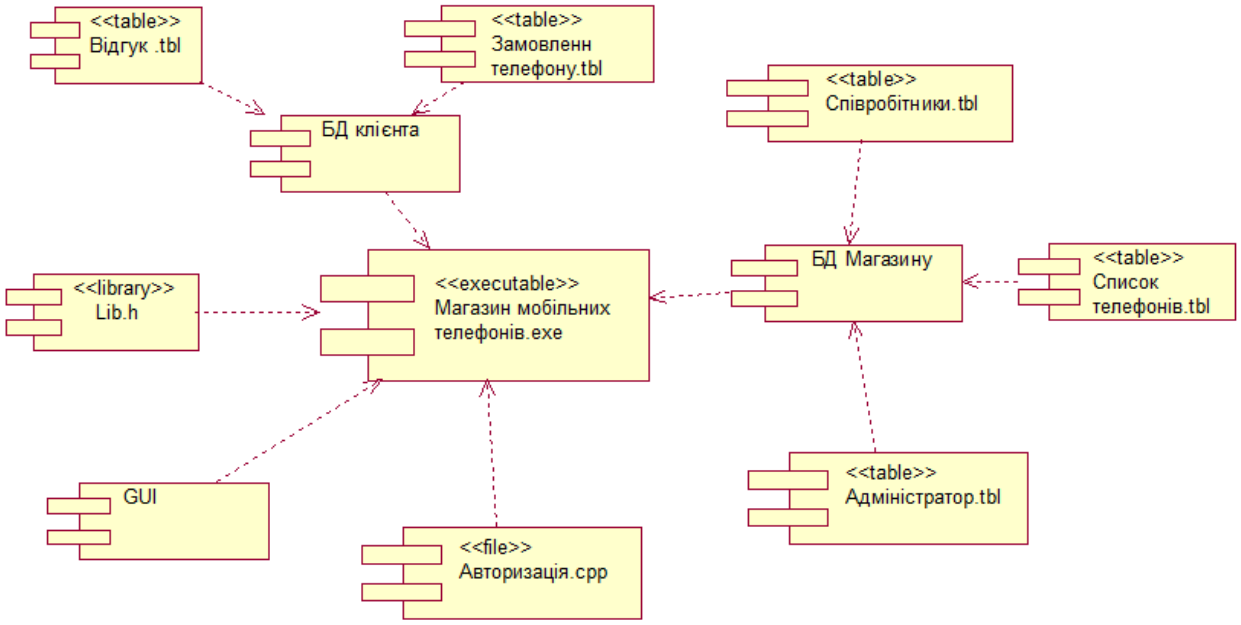
Рисунок 1.4 – Піктограма об’єктної бібліотеки

Рисунок 1.5 – Піктограма документу з ісходним кодом

Рисунок 1.6 – Піктограма таблиці даних БД

Рисунок 1.7 – Піктограма документу

Приклад:



Тема: Діаграма розгортання

Мета: Розглянути основні поняття діаграми розгортання. Ознайомитись з графічною нотацією діаграми.

Структура заняття:

I. Організаційний момент:
   1. Готовність групи до заняття;
   2. Психоемоційний настрій;
   3. Перевірка присутніх.

II. Актуалізація опорних знань студентів:
   1. Повідомлення теми та мети заняття;
   2. Відповіді та запитання.

III. Виклад нового матеріалу:

План:

1. Діаграма розміщення
2. Елементи діаграми розгортання:
   - вузли
   - відношення залежності
   - відношення асоціації

IV. Узагальнення та систематизація знань.

IV. Підведення підсумків заняття.

VI. Домашнє завдання:

1. Ознайомитись з теоретичними відомостями лекції
2. Вивчити основні поняття лекції.
3. Дати відповіді на контрольні питання.

Контрольні питання

1. Які вершини й ребра утворюють діаграму розміщення?
2. Чим відрізняється вузол від компонента?
3. Де можна використовувати й де не можна використовувати екземпляри компонентів?

Діаграма розміщення (розгортання) - друга із двох різновидів діаграм реалізації UML, що моделюють фізичні аспекти об'єктно-орієнтованих систем. Діаграма розміщення показує конфігурацію обробних вузлів у період роботи системи, а також компоненти, "живучі" у них.

Елементами діаграм розміщення є вузли, а також відносини залежності й асоціації. Як і інші діаграми, діаграми розміщення можуть включати примітки й обмеження. Крім того, діаграми розміщення можуть включати компоненти, кожний з яких повинен жити в деякому вузлі, а також містити пакети або підсистеми, використовувані для угруповання елементів моделі у великі фрагменти. При необхідності візуалізації конкретного варіанта апаратної топології в діаграми розміщення можуть міститися об'єкти.

Вузли

Вузол - фізичний елемент, який існує в період роботи системи й представляє комп'ютерний ресурс, що має пам'ять, а можливо, і здатність обробки. Графічно вузол зображується як куб з іменем (рис. 1).

Рисунок 1 - Позначення вузла

Як і клас, вузол може мати додаткову секцію, що відображає розташовувані в ньому елементи (рис. 2).

Рисунок 2 - Розміщення компонентів у вузлі

Зрівняємо вузли з компонентами. Звичайно, у них є подібні характеристики:

- наявність імені;
- можливість бути вкладеним;
- наявність екземплярів.

Остання характеристика говорить про те, що на попередньому рисунку зображений тип Контролера. Зображення конкретного екземпляра, що належить цьому типу, презентовано на рис. 3.

Тепер обговоримо відмінності вузлів від компонентів. По-перше, вони належать до різних рівнів ієрархії у фізичній реалізації системи. Фізично система складається з вузлів, а вузли - з компонентів. По-друге, у кожного з них своє призначення. Компонент призначений для фізичного впакування й матеріалізації набору логічних елементів (класів і кооперацій). Вузол же є тем місцем, де фізично розміщаються компоненти, тобто відіграє роль "квартири" для компонентів.

Рисунок 3 - Екземпляр вузла

Відношення між вузлом і компонентами, які він розміщує, можна відобразити явно. Відношення залежності між вузлом Контролер і компонентами Введення.exe, Висновки.exe ілюструє рис. 4. Правда, найчастіше такі відносини не відображаються. Їх зручно представляти в окремій специфікації вузла.

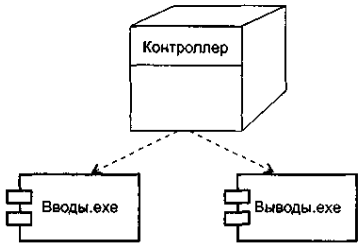


Рисунок 4 - Залежність вузла від компонентів

Угруповання набору об'єктів або компонентів, розташованих у вузлі, звичайно називають *розповсюджуваним модулем*.

Графічно діаграма розміщення - це граф з вузлів (або екземплярів вузлів), з'єднаних асоціаціями, які показують існуючі комунікації. Екземпляри вузлів можуть містити екземпляри компонентів, що живуть або запускаються у вузлах. Екземпляри компонентів можуть містити об'єкти. Як показано на рис. 5, компоненти з'єднуються один з одним пунктирними стрілками залежностей (прямо або через інтерфейси).

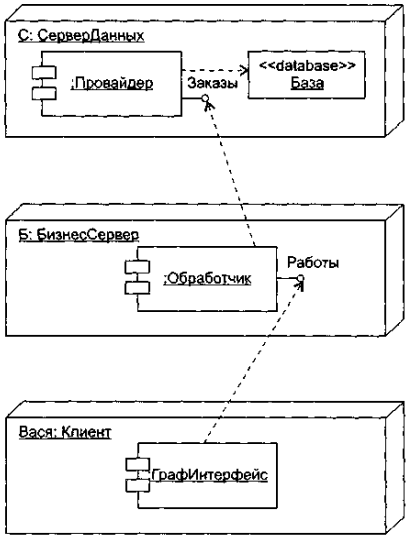


Рисунок 5 - Моделювання розміщення компонентів

На цій діаграмі зображена типова трирівнева система:

1. рівень бази даних реалізований екземпляром С вузла СерверДанных;
2. рівень бізнес-логіки представлений екземпляром Б вузла БизнесСервер;
3. рівень графічного інтерфейсу користувача утворений екземпляром Вася вузла Клієнт.

https://www.guru99.com/uk/deployment-diagram-uml-example.html