Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»

С. М. Алхімова

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Частина 2

ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

С. М. Алхімова

Об'єктно-орієнтоване програмування

Частина 2

Об'єктно-орієнований підхід до розробки програмного забезпечення

Підручник

Затверджено Вченою радою КПІ ім. Ігоря Сікорського
як підручник для здобувачів ступеня бакалавра
за спеціальністю «Комп'ютерні науки»

Київ
КПІ ім. Ігоря Сікорського
2019

УДК 004:4(075.8)
А54

Затверджено Вченою радою КПІ ім. Ігоря Сікорського
(Протокол № 11 від 10.12.2018 р.)

Рецензенти: *Ю. В. Бойко*, канд. фіз.-мат. наук, доц.,
Київський національний університет імені Тараса Шевченка

В. А. Бородін, канд. техн. наук, доц.,
Київський національний університет імені Тараса Шевченка

Відповідальний редактор *Є. А. Настенко*, д-р біол. наук, ст. наук. співроб.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Алхімова С. М.

А54 Обєктно-орієнтоване програмування : підручник. У 2-х ч. Ч. 2. Обєктно-орієнтований підхід до розробки програмного забезпечення / С. М. Алхімова. – Київ : КПІ ім. Ігоря Сікорського, Вид-во «Політехніка», 2019. – 192 с.

ISBN 978-966-622-923-9

Подано теоретичні відомості та практичні приклади, що стосуються розробки сучасного програмного забезпечення. Викладений матеріал спрямований на поглиблення знань щодо використання об'єктно-орієнтованого підходу у цій сфері. Детально розглянуто основні етапи розробки програм, такі як аналіз вимог, проектування, реалізація коду та тестування. Крім цього, наведено вимоги щодо виконання та оформлення курсової роботи з дисципліни «Об'єктно-орієнтоване програмування», варіанти завдань для самостійної роботи та список контрольних запитань для самоперевірки.

Для студентів факультету біомедичної інженерії, які навчаються за спеціальністю «Комп'ютерні науки», а також усіх, хто цікавиться об'єктно-орієнтованим програмуванням.

УДК 004.4(075.8)

ISBN 978-966-622-923-9

ЗМІСТ

ПЕРЕДМОВА

Сьогодні сфера інформаційних технологій є однією з найстабільніших на ринку праці України, а кількість відкритих вакансій на фахівців цієї сфери набагато більше, ніж, власне, фахівців. Не виникає сумнівів, що підготовка студентів з відповідних спеціальностей має бути спрямована, перш за все, на те, щоб випускати фахівців, які здатні відповідати вимогам сучасних технологічних компаній. Вивчення студентами дисциплін, особливо в області програмування, має надавати їм цілісну форму поєднання знань, умінь, навичок, яка дозволить в майбутньому виконувати професійні завдання.

Об'єктно-орієнтований підхід сьогодні є найбільш затребуваним під час розробки програмних продуктів. Однак через те, що апаратно-програмні комплекси та технології розвиваються дуже швидкими темпами, для забезпечення студентів відповідним рівнем володіння професійними компетенціями у цій сфері потрібно постійно переглядати, доповнювати і модернізувати існуючі підходи навчання об'єктно-орієнтованому програмуванню.

Метою створення підручника було підвищення рівня професійної підготовки студентів з програмування за рахунок системності та компактності викладення сучасних уявлень про об'єктно-орієнтований підхід на основі мови програмування C++, а також про всі етапи розробки сучасного програмного забезпечення з його використанням.

Видання повністю розкриває дисципліну «Об'єктно-орієнтоване програмування», яку автор викладає протягом багатьох років на кафедрі біомедичної кібернетики Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». Воно призначене забезпечити читачів необхідним теоретичним матеріалом для вивчення об'єктно-орієнтованого програмування, навчити їх методично правильно застосовувати прийоми об'єктно-орієнтованого підходу на різних етапах розробки програмного забезпечення.

Підручник складається з двох частин, що відповідно присвячені двом кредитним модулям навчальної дисципліни.

Перша частина підручника допоможе зрозуміти сутність і зміст об'єктно-орієнтованого програмування. Її матеріал вміщує основні відомості з теоретичного курсу, демонстраційні приклади програмної реалізації типових задач об'єктно-орієнтованого програмування мовою C++, а також роз'яснення щодо виконання та варіанти завдань однакового ступеню складності до комп'ютерних практикумів, список контрольних запитань для самоперевірки.

Друга частина підручника спрямована на поглиблення знань щодо використання об'єктно-орієнтованого підходу у сфері розробки сучасного програмного забезпечення. Її матеріал містить теоретичні відомості щодо основних етапів розробки програмного забезпечення, практичні приклади проектування та написання коду складових програмного забезпечення мовами

UML та C++ відповідно, а також вимоги щодо виконання та оформлення курсової роботи, варіанти завдань для самостійної роботи та список контрольних запитань для самоперевірки.

Передбачено, що читачі підручника вже мають певний власний досвід програмування мовою C++ і не потребують відомостей щодо деталей синтаксису та семантики мовних конструкцій для розробки програм цією мовою.

Підручник призначений для студентів факультету біомедичної інженерії, які навчаються за спеціальністю 122 «Комп'ютерні науки та інформаційні технології», також буде корисний для студентів, аспірантів та викладачів інженерно-технічних спеціальностей вищих навчальних закладів як додатковий з об'єктно-орієнтованого програмування.

Автор з вдячністю прийме будь-які конструктивні зауваження стосовно викладеного у підручнику матеріалу, які можна надсилати за адресою:

Кафедра біомедичної кібернетики
КПІ ім. Ігоря Сікорського
пр-т Перемоги, 37, м. Київ, Україна, 03056
e-mail: asnarta@gmail.com

ВСТУП

Матеріал підручника спрямований на поглиблене вивчення об'єктно-орієнтованого підходу у розробці сучасного програмного забезпечення та має бути використаний під час виконання студентами курсової роботи з дисципліни «Об'єктно-орієнтоване програмування».

Метою курсової роботи є систематизація, засвоєння і поглиблення теоретичних знань із дисципліни «Об'єктно-орієнтоване програмування». Також її метою є отримання практичних навичок створення програмного забезпечення, використовуючи об'єктно-орієнтований підхід під час аналізу і проектування мовою моделювання UML та під час написання коду мовою програмування C++ в інтегрованому середовищі розробки програмного забезпечення Microsoft Visual Studio.

Уніфіковану графічну мову моделювання UML використовують для опису, візуалізації, проектування та документування об'єктно-орієнтованих систем. Вона призначена для супроводу процесу моделювання програмного забезпечення на основі об'єктно-орієнтованого підходу. Моделі системи, що розроблені мовою UML, сьогодні використовують на всіх етапах життєвого циклу програмного забезпечення від бізнес-аналізу до супроводу.

Мова C ++ є однією з найпопулярніших мов програмування. Її характерною рисою є, насамперед, потужна підтримка об'єктно-орієнтованого підходу до розробки програм. Широкий діапазон стандартних типів даних цієї мови і можливості створення типів користувача дозволяють легко реалізовувати специфіку предметної області. Високорозвинені схеми перетворення типів дозволяють забезпечити компроміс між строгою типізацією даних і ефективністю виконання програм. Засоби явного керування областю видимості змінних через простір імен надають зручний підхід до структурування великих програм. Для підвищення надійності програм, створених мовою C++, слугує простий і гнучкий механізм керування винятковими ситуаціями.

Інтегроване середовище розробки програмного забезпечення Microsoft Visual Studio є набором інструментів, що багаторазово прискорює процес розробки програм. Сьогодні Microsoft Visual Studio є одним з найкращих засобів розробки програмного забезпечення з потужними та ефективними можливостями для написання коду, що включають реалізацію стандартної бібліотеки, компілятор, компонувальник і відлагоджувач. Також Microsoft Visual Studio має редактор графічного інтерфейсу користувача, засоби автоматизації проектування та тестування програм, інструменти підтримки систем контролю версій і взаємодії всередині команди розробників, засоби статичного аналізу коду, профілювання тощо.

Основна задача курсової роботи полягає у підготовці студентів до самостійної практичної діяльності з виконання всього комплексу задач розробки сучасного програмного забезпечення.

Виконання курсової роботи передбачає проведення аналізу вимог проектування, написання коду та тестування програми відповідно до варіанта завдання (додаток А). Крім цього необхідно підготувати звіт.

Робота студентів над курсовою роботою має складатися з такої послідовності кроків:

Крок 1. Отримання номера варіанта завдання курсової роботи від керівника.

Крок 2. Оформлення титульної сторінки, завдання на курсову роботу та календарного плану. Перевірка оформлення зазначених сторінок і підпис у керівника, що підтверджує отримання завдання на курсову роботу і початок роботи над нею.

Крок 3. Ознайомлення з теоретичним матеріалом, який наведений у розділі «Процес розробки програмного забезпечення», та вимогами щодо оформлення пояснювальної записки та електронного звіту з курсової роботи. Огляд технічної літератури за темою роботи. Оформлення вступу пояснювальної записки до курсової роботи. Перевірка наявності та, у разі необхідності, встановлення всього програмного забезпечення, що необхідне для виконання курсової роботи.

Крок 4. Ознайомлення з теоретичним матеріалом, який наведений у розділі «Вимоги до програмного забезпечення». Аналіз функціональних та нефункціональних вимог і розробка специфікації вимог до програмного продукту, який необхідно розробити в курсовій роботі відповідно до варіанта завдання. За результатами розробки специфікації вимог до програмного продукту виконується оформлення першого розділу пояснювальної записки до курсової роботи.

Крок 5. Ознайомлення з теоретичним матеріалом, який наведений у розділі «Проектування програмного забезпечення». Виконання попереднього проектування архітектури програмного продукту, який необхідно розробити в курсовій роботі відповідно до варіанта завдання.

Крок 6. Ознайомлення з теоретичним матеріалом, який наведений у розділі «Реалізація коду програмного забезпечення», та вимогами щодо оформлення коду. Виконання першого етапу

розробки коду програмного продукту, що полягає у визначенні класів та реалізації меню користувача. Відлагодження коду програмного продукту. Початкове оформлення архітектури програмного продукту через визначення класів та відношень між ними на діаграмі класів і часової послідовності подій на діаграмі послідовності відповідно до правил проектування діаграм в нотації UML. За результатами розробки архітектури програмного продукту виконується часткове оформлення другого розділу пояснювальної записки до курсової роботи. Перший контроль за процесом виконання курсової роботи, консультація у викладача.

Крок 7. Виконання другого та третього етапів розробки коду програмного продукту, що полягає у створенні об'єктів, використанні контейнера та організації роботи з даними через файл. Відлагодження коду програмного продукту. За необхідності внесення змін до архітектури програмного продукту. Другий контроль за процесом виконання курсової роботи, консультація у викладача.

Крок 8. Виконання четвертого етапу розробки коду програмного продукту, що полягає в реалізації коду для виконання пошуку даних у контейнері. Відлагодження коду програмного продукту. У разі необхідності, внесення змін до архітектури програмного продукту.

Крок 9. Ознайомлення з теоретичним матеріалом, який наведений у розділі «Тестування програмного забезпечення». Розробка тест-плану, який містить набір тестових сценаріїв для повного покриття вимог відповідно до визначеної специфікації. За результатами розробки тест-плану тестування програмного продукту виконується часткове оформлення третього розділу пояснювальної записки до курсової роботи.

Крок 10. Розробка модульних тестів для перевірки роботи контейнера. Виконання модульного тестування роботи контейнера у розробленому програмному продукті.

Крок 11. Аналіз виявлених помилок після проведення модульного тестування розробленого програмного продукту. Внесення відповідних змін до коду програми, що дозволять коректно обробляти виняткові ситуації, використовуючи механізми керування винятками мови C++. У разі необхідності, внесення змін до архітектури програмного продукту.

Крок 12. Розробка діаграми об'єктів відповідно до правил проектування діаграм у нотації UML, яка містить принаймні 10 об'єктів, що є елементами контейнера програмного продукту, з описом реальних даних для перевірки роботи програми. Виконання системного тестування розробленого програмного продукту (обов'язковою є перевірка захисту від некоректних дій користувача під час роботи з програмним продуктом та можливості запуску виконуваного файлу (*.exe) з будь якого місця диску).

Крок 13. Аналіз виявлених помилок після проведення системного тестування розробленого програмного продукту. Внесення відповідних змін до коду програми, що дозволять коректно обробляти виняткові ситуації, використовуючи механізми керування винятками мови C++. У разі необхідності, внесення змін до архітектури програмного продукту.

Крок 14. Остаточне оформлення другого та третього розділів пояснювальної записки до курсової роботи. Написання висновків. Формування додатків до пояснювальної записки, в яких необхідно навести повний текст коду розробленого програмного продукту і повний текст коду, розробленого для тестування роботи контейнера модульних тестів. Написання тексту анотації трьома мовами: українською, російською, англійською.

Крок 15. Перевірка оформлення пояснювальної записки до курсової роботи відповідно до вимог. Друк пояснювальної записки та підпис у керівника роботи. Формування електронного звіту з курсової роботи відповідно до вимог.

Крок 16. Надання керівнику на перевірку пояснювальної записки (роздрукованого звіту) та електронного звіту до виконаної курсової роботи. Отримання допуску до захисту курсової роботи.

Крок 17. Проходження співбесіди із захисту курсової роботи у визначений керівником час, на якій необхідно продемонструвати роботу розробленої програми на персональному комп'ютері та відповісти на запитання щодо етапів її розробки.

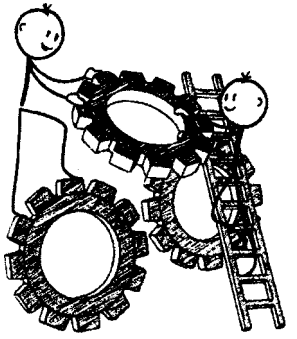
Під час виконання кожного з етапів курсової роботи слід пам'ятати, що до роботи висувається така вимога, як відсутність плагіату. Курсові роботи, в яких або в програмному коді, або в тексті пояснювальної записки буде

виявлено ознаки плагіату, знімаються з захисту, а їх авторам виставляється оцінка "не задовільно". У зазначених випадках для здачі курсової роботи студенту необхідно буде виконати нову роботу відповідно до іншого варіанта, що визначить керівник роботи.

Крім того, не допускаються до захисту роботи, текст пояснювальної записки яких або електронний звіт не відповідають вимогам оформлення. Також не допускаються до захисту роботи, надані на перевірку з порушенням термінів їх виконання.

У результаті виконання курсової роботи студенти отримують знання з використання об'єктно-орієнтованого підходу для створення повноцінного програмного забезпечення. Досвід розробки програмного забезпечення, набутий студентами під час виконання курсової роботи, є основою для підготовки дипломних робіт за спеціальністю та буде корисним у подальшій практичній роботі за фахом.

РОЗДІЛ 1
ПРОЦЕС РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



Процес розробки (життєвий цикл) програмного забезпечення починається з моменту виникнення необхідності в певному програмному забезпеченні і закінчується введенням його в дію з подальшою експлуатацією.

Складові етапи процесу розробки програмного забезпечення регламентуються міжнародним стандартом ISO / IEC 12207 : 1995 «Information Technology – Software Life Cycle Process». Цей стандарт лише визначає етапи життєвого циклу, не розкриваючи в деталях, як реалізовувати або виконувати дії і завдання кожного з етапів.

1.1. Етапи розробки програмного забезпечення

Процес розробки програмного забезпечення визначається як сукупність взаємопов'язаних дій, що перетворюють деякі початкові дані у кінцеві та характеризуються певними завданнями і методами їх вирішення. Умовно такі дії можна згрупувати, виділивши наступні основні етапи розробки програмного забезпечення:

- постановка завдання;
- специфікація вимог;
- проектування;
- реалізація;
- тестування;
- впровадження і супровід.

Процес розробки програмного забезпечення відрізняється тим, що зазвичай до робіт над наступним етапом приступають, коли рівень готовності попереднього досягає близько 80 – 90 %. Найбільшою мірою цю обставину слід враховувати під час специфікації вимог, оскільки саме у визначенні вимог якийсь рівень невизначеності іноді зберігається практично до завершення проекту. Цей момент є серйозним фактором ризику і повинен безперервно контролюватися.

1.1.1. Постановка завдання

Цей етап характерний формулюванням цілей і завдань, які повинно вирішувати програмне забезпечення. На ньому виділяють базові взаємозв'язки і сутність проекту, тобто готується основа для всього подальшого процесу.

В результаті цього етапу часто створюють документ постановки завдання, в якому може міститися наступна інформація:

- формулювання умов завдання;
- короткий опис програмного забезпечення, що буде розроблятися, його призначення;
- дати початку і закінчення розробки програмного забезпечення;
- підстави для розробки програмного забезпечення;
- коротка характеристика об'єкта розробки;
- замовник (або замовники) програмного забезпечення;
- перелік основних вимог користувача до програмного забезпечення, що буде розроблятися;
- опис вихідних даних програмного забезпечення з точки зору їх змісту і призначення (звіти, файли, записи, поля даних, таблиці тощо);
- опис інформації, яку отримують в результаті роботи програмного забезпечення, що буде розроблятися;
- зовнішні обмеження;
- окупність капіталовкладень, що визначає прибуток, який надасть створення програмного забезпечення в поняттях, які відповідають цільовим призначенням організації замовника;
- термінологія, яка може виявитися специфічною для певної розробки програмного забезпечення;
- інші угоди сторін.

1.1.2. Специфікація вимог

У рамках цього етапу фіксуються вимоги замовника (функції програмного забезпечення, що розробляється, та особливості його експлуатації), підбираються методи оптимального виконання поставленого в проекті завдання (підхід, архітектура, технологія, середа та мова програмування), визначається ступінь автоматизації проекту, відбувається виявлення найбільш актуальних для проекту бізнес-процесів. На цьому ж етапі, як правило, вирішуються питання з фінансування і терміни реалізації проекту.

Опис можливостей і обмежень, які накладаються на програмне забезпечення, що розробляється, називається вимогами, а сам процес формування, аналізу, документування та перевірки цих можливостей і обмежень – розробкою вимог. Специфікацією вимог називають точний формалізований опис функцій і обмежень програмного забезпечення, що розробляється.

Таким чином, для отримання специфікації виконують аналіз технічного завдання, формулюють змістовну постановку задачі, вибирають математичний апарат формалізації, будують модель предметної області, визначають підзадачі і вибирають або розробляють методи їх вирішення.

1.1.3. Проектування

Процес проектування складного програмного забезпечення зазвичай включає:

- проектування загальної структури, тобто визначення основних частин (компонентів) і їх взаємозв'язків з управління та обміну даними;
- декомпозицію компонентів і побудову структурних ієрархій відповідно до рекомендацій блочно-ієрархічного підходу;
- проектування компонентів.

Результатом проектування є детальна модель програмного забезпечення, що розробляється.

Тип моделі залежить від обраного або заданого підходу (структурний, об'єктно-орієнтований або компонентний) і конкретної технології проектування. У будь-якому випадку процес проектування охоплює як проектування обробляючих програм (підпрограм) і визначення взаємозв'язків між ними, так і проектування даних, з якими взаємодіють ці програми або підпрограми.

1.1.4. Реалізація

За результатами попереднього етапу починається розробка програмного забезпечення, яка включає в себе поетапне написання коду обраною мовою програмування для обраної платформи, на якій буде працювати система.

Окрім безпосереднього написання коду, процес реалізації програмного забезпечення включає в себе самотестування, яке виконується безпосередньо програмістом, і відлагодження, тобто пошук помилок в коді.

Процес реалізації може відбуватися паралельно з наступним етапом розробки – тестуванням програмного забезпечення, що допомагає вносити зміни безпосередньо в ході написання коду.

Основним завданням процесу реалізації програмного забезпечення є перетворення вихідного проекту, що визначається специфікацією вимог, в працюючий код.

1.1.5. Тестування

Тестування програмного забезпечення – це процес дослідження програмного забезпечення з метою отримання інформації про його якість. Як вже зазначалося раніше, цей етап розробки програмного забезпечення найтіснішим чином пов'язаний з попереднім.

На етапі тестування проводять роботи, що дають можливість перевірити систему на відповідність вимогам. Як результат, отримують висновок про готовність програмного забезпечення і висновок щодо якості проведеної роботи.

1.1.6. Впровадження і супровід

Процес впровадження програмного забезпечення є не менш трудомістким, ніж процес його розробки. Впровадження програмного

забезпечення – це процес налаштування програмного забезпечення під умови його використання, а також навчання користувачів роботі з новим програмним забезпеченням.

Завершення впровадження нового програмного забезпечення включає виконання наступних робіт:

- створення детальної інструкції з експлуатації нового програмного забезпечення;
- навчання групи фахівців з боку замовника роботі з новим програмним забезпеченням, що може бути виконано віддалено або на території замовника;
- внесення змін стосовно першого досвіду експлуатації замовником нового програмного забезпечення, що може включати виправлення якихось помилок або доробку якогось функціоналу.

Після закінчення внесення оговорених змін і усунення зауважень підписується акт здачі робіт і прийняття проекту відповідно до специфікації вимог, після чого система передається замовнику і операція з впровадження вважається завершеною.

Впровадження програмного забезпечення вважається таким, що відбулося, тільки в тому випадку, якщо програмне забезпечення виконує покладені на нього функції і співробітники компанії перейшли на використання нового програмного продукту. Однак розробка програмного забезпечення не закінчується передачею програми клієнту. У ході роботи виявляються аномалії, змінюється робоче середовище, виникають нові вимоги.

Зміни в програмному забезпеченні після його передачі в експлуатацію називаються супроводом. Супровід спрямований на створення і впровадження нових версій програмного забезпечення.

Причинами випуску нових версій програмного забезпечення в процесі супроводу можуть служити:

- необхідність виправлення помилок, виявлених в процесі експлуатації попередніх версій;
- необхідність удосконалення попередніх версій, що може бути обумовлена потребами поліпшення інтерфейсу, розширення виконуваних функцій або підвищення продуктивності програмного забезпечення;
- зміна середовища функціонування, що обумовлена появою нових технічних засобів та / або програмних продуктів, з якими взаємодіє програмне забезпечення, що супроводжується.

У програмний продукт вносять необхідні зміни, які можуть вимагати перегляду вже прийнятих проектних рішень. Зі зміною моделі життєвого циклу програмного забезпечення роль цього етапу істотно зросла, оскільки більшість програм тепер створюється ітераційно: спочатку випускається порівняно проста версія, потім наступна з більшими можливостями, потім наступна і так далі.

1.2. Об'єктно-орієнтований підхід до розробки програмного забезпечення

Початок використання для розробки програмного забезпечення об'єктно-орієнтованого підходу пов'язаний передусім з такими подіями як:

- серйозний прогрес у сфері архітектури комп'ютерів;
- розвиток мов програмування;
- розвиток технологій програмування, що стали ґрунтуватися на принципах модульності та інкапсуляції даних;
- розвиток теорії баз даних;
- прогрес у сфері штучного інтелекту.

Об'єктно-орієнований підхід вже зарекомендував себе не тільки як принцип, який застосовується в програмуванні, але й як широко використовуваний підхід у проектуванні інтерфейсів користувача, баз даних і навіть архітектури комп'ютерів. Сьогодні об'єктно-орієнтований підхід є найбільш затребуваним під час розробки програмного забезпечення.

1.2.1. Основні складові об'єктно-орієнтованого підходу

Основними складовими об'єктно-орієнованого підходу є:

- об'єктно-орієнтований аналіз;
- об'єктно-орієнтоване проектування;
- об'єктно-орієнтоване програмування.

Об'єктно-орієнтований аналіз – це методологія, в якій вимоги до системи розглядаються з точки зору класів та об'єктів, що можна визначити в предметній області.

Об'єктно-орієнтоване проектування – це методологія проектування, що об'єднує в собі процес об'єктної декомпозиції і прийоми опису різних моделей системи, що проектується.

Об'єктно-орієнтоване програмування – це методологія програмування, в якій програмне забезпечення реалізують, ґрунтуючись на певній множині об'єктів, кожен з яких є екземпляром певного класу, а класи утворюють ієрархію.

Виходячи з наведених означень, програмний продукт буде об'єктно-орієнованим тільки в тому випадку, коли будуть виконуватися наступні основні вимоги:

- в якості базових елементів використовують об'єкти, а не алгоритми;
- кожен окремий об'єкт програми є екземпляром визначеного класу;
- зв'язки між класами формують ієрархії.

За результатами об'єктно-орієнтованого аналізу створюються моделі складних систем, на основі яких ґрунтується об'єктно-орієнтоване проектування. Об'єктно-орієнтоване проектування, в свою чергу, створює фундамент для реалізації складної системи з використанням методології об'єктно-орієнтованого програмування.

1.2.2. Об'єктно-орієнтований підхід у розробці складних систем

Програмне забезпечення, яке розробляють за допомогою об'єктно-орієнтованого підходу, як правило, пов'язане з роботою зі складними системами.

У переважній більшості випадків складність систем перевищує можливості людського інтелекту. Саме тому один окремий розробник або навіть й група розробників не в змозі охопити всі аспекти складної системи. Загалом, з цією складністю можна боротися, але уникнути її повністю – неможливо.

Можна виділити наступні чинники, що обумовлюють складність розробки програмного забезпечення:

- складність реального світу або тієї його частини, яку моделює програмне забезпечення, що необхідно розробити;
- складність, яка пов'язана з описом поведінки системи, що має бути дискретно визначена під час розробки програмного забезпечення;
- складність в управлінні самим процесом розробки програмного забезпечення.

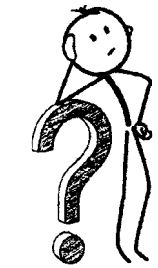
Ось чому під час створення програмного забезпечення використовують надійні способи побудови складних систем, які з високою ймовірністю будуть гарантувати успіх в їх розробці.

Обґрунтувати використання об'єктно-орієнтованого підходу під час розробки програмного забезпечення можна наступними властивостями складних систем:

- складні системи складаються з взаємозв'язаних підсистем, що також можуть бути поділені на частини;
- поділ системи на складові є відносним, оскільки в більшості випадків залежить від аспектів дослідження цієї системи;
- зв'язок між частинами всередині компонентів системи набагато сильніший, ніж між самими компонентами, що спричиняє сильну взаємодію частин всередині компонентів та слабкішу між самими компонентами;
- для складних систем характерна ієрархічність;
- ієрархічні системи зазвичай складаються з невеликої кількості різних типів їх підсистем, які можуть бути по-різному поєднані та організовані;
- поточна складна система, що функціонує, є результатом розвитку більш простої системи, що працювала раніше.

Досвід розробки складного програмного забезпечення показує, що найбільш успішними є ті програми, в яких закладені добре продумані структури класів та об'єктів і які володіють згаданими вище властивостями для систем, з якими вони працюють.

Контрольні запитання до розділу



1. Яким стандартом регламентуються складові етапи процесу розробки програмного забезпечення?
2. Що таке процес розробки програмного забезпечення?
3. Які існують основні етапи розробки програмного забезпечення?
4. Що є серйозним фактором ризику і повинно безперервно контролюватися під час розробки програмного забезпечення?
5. Чим відрізняється процес розробки програмного забезпечення стосовно переходів від одного його етапу до наступного?
6. Чим характеризується етап постановки завдання під час розробки програмного забезпечення?
7. Які дії виконують в рамках етапу створення специфікації вимог під час розробки програмного забезпечення?
8. Що таке вимоги до програмного забезпечення?
9. Що таке розробка вимог до програмного забезпечення?
10. Що таке специфікація вимог програмного забезпечення?
11. Які дії необхідно виконати для розробки специфікації вимог програмного забезпечення?
12. Що в себе включає процес проектування складного програмного забезпечення?
13. Що є результатом проектування програмного забезпечення?
14. Від чого залежить тип моделі програмного забезпечення, що розробляється?
15. Які дії входять до етапу реалізації програмного забезпечення під час його розробки?
16. Чи може процес реалізації програмного забезпечення відбуватися паралельно з його тестуванням?
17. Яке основне завдання процесу реалізації програмного забезпечення?
18. Що таке тестування програмного забезпечення?
19. Що отримують в результаті тестування програмного забезпечення?
20. Що називається впровадженням програмного забезпечення?
21. Які дії мають бути виконані для завершення впровадження нового програмного забезпечення?
22. За яких умов підписується акт здачі робіт і прийняття проекту відповідно до специфікації вимог?
23. Що називається супроводом програмного забезпечення?
24. Які існують причини випуску нових версій програмного забезпечення в процесі його супроводу?
25. Що є характерною рисою ітераційного створення програмного забезпечення?
26. З чим пов'язаний початок використання об'єктно-орієнтованого підходу для розробки програмного забезпечення?

27. Що є основними складовими об'єктно-орієнтованого підходу?
28. Що таке об'єктно-орієнтований аналіз?
29. Що таке об'єктно-орієнтоване проектування?
30. Що таке об'єктно-орієнтоване програмування?
31. За яких умов розроблений програмний продукт вважається об'єктно-орієнтованим?
32. Які чинники обумовлюють складність розробки програмного забезпечення?
33. Які властивості складних систем обумовлюють використання об'єктно-орієнтованого підходу під час розробки програмного забезпечення?
34. Яке програмне забезпечення є найбільш успішним з огляду на досвід його розробки?

РОЗДІЛ 2
ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Після аналізу функціональних та нефункціональних вимог до програмного продукту, який необхідно розробити в курсовій роботі відповідно до варіанта, необхідно провести їх документування, склавши специфікацію вимог.

Документування специфікації вимог програмного продукту, що розробляється в курсовій роботі, має складатися з наступних кроків:

- опис варіантів використання, використовуючи коротку форму їх опису в таблицях;
- розробка діаграми варіантів використання в нотації UML;
- опис технічних обмежень, зовнішніх інтерфейсів та атрибутів якості природною мовою.

Опис функціональних вимог необхідно виконувати, дотримуючись правил опису варіантів використання та правил проектування діаграм в нотації UML в середовищі Microsoft Visio, які викладені в цьому розділі.

Опис нефункціональних вимог необхідно виконувати, дотримуючись правил опису технічних обмежень, зовнішніх інтерфейсів та атрибутів якості, які викладені в цьому розділі.

Кожне програмне забезпечення являє собою певний перетворювач вихідних даних і виведення результатів цього перетворення. Для його побудови до програмного забезпечення формують вимоги, які визначають функції і види обробки даних при виконанні цих функцій. Ці вимоги є предметом угоди між замовником і розробником програмного забезпечення.

У загальному випадку під вимогами до програмного забезпечення розуміють сукупність тверджень щодо атрибутів, властивостей або якостей системи, що підлягає розробці.

Згідно зі IEEE Standard-ом, вимога – це:

- можливість, яка необхідна користувачеві для вирішення проблем або досягнення цілей;
- можливість, якою повинна володіти система, щоб виконати контракт або задовольняти стандартам, специфікаціям або іншим формальним документам.

Таким чином, вимоги до програмного забезпечення (англ. software requirements) – це властивості програмного забезпечення, які повинні бути належним чином реалізовані для адекватного вирішення конкретних практичних завдань.

2.1. Класифікація вимог

Під час розробки вимог до програмного забезпечення важливо розуміти різницю між вимогами, що описують функціональність, і вимогами, що визначають додаткові властивості системи. Крім цього, потрібно враховувати рівень вимог до програмного забезпечення.

2.1.1. Вимоги на рівнях бізнесу, користувача та системи

Зазвичай розрізняють три рівня до вимог програмного забезпечення.

На верхньому рівні представлені вимоги, які визначають, що система повинна робити з точки зору бізнесу. Слово «бізнес» в даному контексті ближче до слова «замовник», оскільки визначені на цьому рівні вимоги відповідають цілям і політиці організації, а їх висловлюють ті, хто фінансує проект (зазвичай топ-менеджери або акціонери компанії). Приклад вимоги бізнесу: промо-сайт, що привертає увагу певної аудиторії до певної продукції компанії.

Наступний рівень вимог – це рівень вимог користувачів, на якому визначають цілі і завдання, які дозволить вирішувати система, тобто те, що користувачі зможуть робити за допомогою системи. Призначені для користувача вимоги повинні відповідати вимогам бізнесу, в іншому випадку їх не слід включати в проект. Приклад вимоги користувача: система повинна представляти діалогові засоби для введення вичерпної інформації про певну продукцію компанії на промо-сайті.

Третій рівень вимог – це рівень вимог до функціональності системи (системні вимоги), що визначають характеристики системи, яку повинні побудувати розробники для того, щоб користувачі змогли виконати свої завдання в рамках вимог бізнесу. Приклад функціональних вимог (або просто функцій): при роботі з інформацією промо-сайту повинна бути можливість створювати, редагувати і видаляти текстовий опис певної продукції компанії.

2.1.2. Функціональні та нефункціональні вимоги

На кожному рівні розрізняють функціональні та нефункціональні вимоги.

Функціональні вимоги (англ. functional requirements) включають в себе перелік сервісів, які повинна виконувати система, причому має бути вказано, як система реагує на ті чи інші вхідні дані, як вона поводиться в певних ситуаціях тощо. У деяких випадках вказується, що система не повинна робити, тобто так звані зворотні вимоги. Отже, функціональні вимоги визначають дії, які повинна виконувати система, без врахування обмежень, пов'язаних з її реалізацією.

Тим самим функціональні вимоги визначають поведінку системи в процесі обробки інформації.

Нефункціональні вимоги (англ. non-functional requirements) описують характеристики системи та її оточення, а не поведінку системи. Також наводиться перелік обмежень, що накладається на дії і функції, що виконує система. Отже, нефункціональні вимоги не визначають поведінку системи, але описують атрибути системи або атрибути системного оточення.

Основна проблема нефункціональних вимог полягає в тому, що їх виконання важко перевірити. Часто вони пишуться для того, щоб відобразити загальні цілі замовника програмного забезпечення, такі як: простота експлуатації, можливість відновлення після збоїв або швидка відповідь на запити користувача. Реалізація подібних вимог може виявитися складним завданням для розробників, оскільки вони нечітко сформульовані і відкривають простір для різних тлумачень. В ідеалі нефункціональні вимоги повинні виражатися через кількісні показники, які можна об'єктивно виміряти.

2.1.3. Властивості вимог до програмного забезпечення

Вимоги повинні мати наступні властивості:

- *атомарність* означає, що вимогу не можна розбити на більш менші вимоги або уточнити іншою вимогою;
- *одиничність* означає, що вимога стосується тільки однієї властивості системи, тобто система повинна виконувати таку-то дію;
- *завершеність* означає, що вимогу повністю визначено в одному місці документації, тобто не може бути так, щоб в різних частинах документа йшлося про один і той самий функціонал системи;.
- *послідовність* означає, що вимога не повинна суперечити іншим вимогам і обмеженням системи;
- *актуальність* означає, що вимога не стала застарілою з плином часу (тобто вимога відповідає, як приклад, сучасному законодавству або технічним реаліям);
- *відслідковуваність* означає, що вимогу зафіксовано в документації і можна зрозуміти звідки вона взялася;
- *здійсненність* означає, що вимогу можна виконати в рамках існуючих технологій;
- *зрозумілість* означає, що вимога визначена без звернення до технічного сленгу, акронімів або інших прихованих формулювань (не містить нечітких фраз, відсутнє використання негативних і складених тверджень), вимога визначає об'єкти і факти, а не суб'єктивні думки, а також можлива одна і тільки одна її інтерпретація;
- *тестуємість* означає, що виконання реалізованої вимоги можна перевірити;
- *обов'язковість* означає, що без виконання цієї вимоги користувач не зможе в повній мірі використовувати систему (необов'язковість вимоги – це протиріччя самому поняттю вимоги);

- *повнота* означає, що сукупність артефактів, що описують вимоги, вичерпним чином описує все те, що потрібно від системи, яка розробляється (інакше кажучи, треба зафіксувати все, що система повинна робити);

2.2. Специфікація вимог програмного забезпечення

Специфікація вимог програмного забезпечення (англ. Software Requirements Specification, SRS) – це закінчений опис поведінки програми, яку потрібно розробити. Ось чому специфікація вимог програмного забезпечення є основним документом, що визначає план розробки.

Основні стандарти, методології та довідники, в яких згадується специфікація вимог:

- *ГОСТ 34.602-89 «Техническое задание на создание автоматизированной системы»* регламентує структуру технічного завдання на створення системи, в яку входять програмне забезпечення, апаратне забезпечення, люди, які працюють з програмним забезпеченням, і автоматизовані процеси;
- *IEEE STD 830-1998 «IEEE Recommended Practice for Software Requirements Specifications»* визначає зміст та якісні характеристики правильно складеної специфікації вимог до програмного забезпечення і наводить кілька шаблонів специфікації вимог;
- *29148-2011 «ISO / IEC / IEEE International Standard - Systems and software engineering - Life cycle processes - Requirements engineering»* забезпечує єдине трактування процесів і продуктів, які використовуються при розробці вимог протягом усього життєвого циклу систем і програмного забезпечення.

2.2.1. Процес розробки специфікації вимог

Специфікація вимог служить основою для подальшого проектування та написання коду програмного забезпечення, а також базою для проведення його тестування.

Розробку специфікації починають з аналізу вимог до функціональності програмного забезпечення, що створюється. На етапі такого аналізу ставляться два завдання: уточнити необхідну поведінку програмного забезпечення, що створюється, та розробити концептуальну модель його предметної області з точки зору поставлених завдань. Таким чином, у процесі цього аналізу виявляють зовнішніх користувачів програмного забезпечення, що створюється, та перелік окремих аспектів його поведінки в процесі взаємодії з конкретними користувачами. Аспекти поведінки програмного забезпечення були названі «варіантами використання» або «прецедентами». Варіанти використання є засобом визначення функціональних вимог.

На додаток до визначення функціональних вимог, специфікація також має містити визначення нефункціональних вимог, які накладають обмеження на

дизайн та реалізацію програмного забезпечення, що створюється, і виявляють його експлуатаційні якості, тобто визначають те, наскільки добре воно буде працювати. Таким чином визначають технічні обмеження та зовнішні інтерфейси програмного забезпечення та формулюють атрибути (фактори) його якості.

Отже, всі вимоги, зазначені в специфікації, діляться на функціональні і нефункціональні. Центральне місце займають функціональні вимоги, які специфікують особливості реалізації окремих процесів програмного забезпечення, що розробляється. Однак багато нефункціональних вимог відносяться до системи в цілому, а не до окремих частин. Це означає, що вони більш значимі і критичні, ніж окремі функціональні вимоги. Помилка, допущена у функціональній вимозі, може знизити якість системи; а помилка в нефункціональних вимогах може зробити систему непрацездатною взагалі.

2.2.2. Особливості документування вимог

Способів документування вимог є кілька:

- документація, в якій використовується чітко структурована природна мова;
- графічні моделі, що ілюструють стани системи, процеси переходів між ними, зміну даних, логічні потоки тощо;
- формальні специфікації, де вимоги визначені за допомогою математично точних, формальних логічних мов.

Останній метод забезпечує найвищу ступінь точності, однак мало хто з розробників, і ще менше з замовників, знайомі з цим методом. У більшості проектів не потрібна такого рівня формалізація. Структурована природна мова, посилена візуальними моделями і іншими прийомами відображення інформації (такими, як діаграми в нотації UML), залишається найбільш практичним способом документування вимог в більшості проектів з розробки програмного забезпечення.

Не існує певної методики написання ідеальних вимог. Якість вимог визначається досвідом і здоровим глуздом. Однак можна сформулювати декілька загальних тверджень, які можуть допомогти покращити якість документування вимог до програмного забезпечення:

- специфікація вимог повинна бути повною, але не надмірною: повнота специфікації вимог визначається не кількістю сторінок, а тим, що там наведені всі вимоги, виконання яких дозволяє задовольнити очікування замовника (обсяг специфікації вимог залежить від проекту);
- вимоги повинні добре читатися, що означає, що вимоги повинні бути узгоджені і перебувати в самому документі так, щоб їх можна було легко і логічно знайти;
- вимоги не повинні в собі містити елементи дизайну, але дизайн повинен посилатися на вимоги.

Під час формулювання вимог природною мовою слід дотримуватися ряду правил:

- слід використовувати повні речення з правильною граматикою, правописом і пунктуацією, речення повинні бути короткими і зрозумілими;
- слід використовувати активний стан для дієслів;
- слід остерігатися синонімів і слів, близьких за значенням, тобто не слід в специфікації вимог до програмного забезпечення намагатися урізноманітнити лексику, щоб зацікавити читача, адже специфікація являє собою формальний документ, а не художній твір.

У загальному випадку вимоги змінюються з часом. Після того, як вимоги визначені, зміни повинні потрапляти під контроль внесення змін. Для багатьох проектів вимоги змінюються впритул до моменту завершення створення системи. Це відбувається частково через складність програмного забезпечення і того факту, що користувачі не знають, що їм потрібно насправді. Ця особливість вимог призвела до появи процесу управління вимогами.

2.3. Варіанти використання

Варіант використання (англ. use cases) являє собою послідовність дій (транзакцій), виконуваних системою у відповідь на подію, що ініціюється деяким зовнішнім об'єктом (дійовою особою), в якості якого можуть виступати не тільки люди, але й інші системи і пристрої. Варіант використання описує типову взаємодію між користувачем і системою і відображає уявлення про поведінку системи з точки зору користувача.

Не слід плутати варіант використання з конкретними операціями майбутньої системи. Кожен варіант використання пов'язаний з деякою метою, що має самостійне значення, наприклад, для текстового редактора «формування змісту» – це варіант використання, а «зв'язування заголовків зі спеціальними стилями» – операція, яку необхідно виконати, щоб стала можливою автоматична побудова змісту.

Залежно від мети виконання розрізняють наступні варіанти використання:

- основні – забезпечують необхідну функціональність програмного забезпечення, що створюється;
- допоміжні – забезпечують виконання необхідних налаштувань системи та її обслуговування (наприклад, архівування інформації тощо);
- додаткові – забезпечують додаткові зручності для користувача (як правило, реалізуються в тому випадку, якщо не вимагають серйозних витрат будь-яких ресурсів ні при розробці, ні при експлуатації).

У найпростішому випадку варіант використання визначається в процесі обговорення з користувачем тих функцій, які він хотів би реалізувати, або цілей, які він переслідує по відношенню до системи, що розробляється.

Переваги варіантів використання полягають в тому, що вони визначають користувачів, обмеження системи та системний інтерфейс, є зручними для спілкування користувачів з розробниками, а також використовуються для написання тестів і документації користувача.

2.3.1. Коротка та детальна форми опису варіанта використання

Залежно від рівня абстракції варіант використання може описуватися стисло або більш докладно.

Коротка форма опису містить: унікальний ідентифікатор і назву варіанта використання, його мету, дійових осіб, тип варіанта використання (основний, допоміжний або додатковий) і його короткий опис. Основні варіанти використання зазвичай описують докладно, намагаючись відобразити особливості предметної області програмного забезпечення.

Детальна форма, крім зазначеної вище інформації, включає опис типового перебігу подій і можливих альтернатив. Типовий перебіг подій подають у вигляді діалогу між користувачем і системою, послідовно нумеруючи події. Якщо користувач може вибирати варіанти, то їх описують в окремих таблицях. Також окремо подають альтернативи, пов'язані з порушенням типового перебігу подій.

2.3.2. Діаграми варіантів використання

Наочно зображувати очікувану поведінку системи дозволяють діаграми варіантів використання в нотації UML.

Основними поняттями діаграм варіантів використання є: дійова особа, варіант використання, зв'язок.

Дійова особа (актор) – зовнішня по відношенню до створюваного програмного забезпечення сутність, яка взаємодіє з ним з метою отримання або надання будь-якої інформації. Як уже згадувалося вище, дійовими особами можуть бути користувачі, інше програмне забезпечення або які-небудь технічні засоби, які взаємодіють з програмним забезпеченням, що створюється.

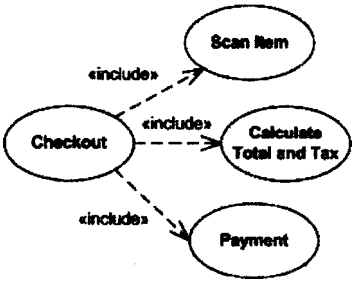
Варіант використання – деяка очевидна для діючої особи процедура, що вирішує якусь конкретну задачу. Всі варіанти використання, так чи інакше, пов'язані з вимогами до функціональності системи, що розробляється, і можуть сильно різнитися за обсягом виконуваної роботи. Варіант використання зображується у вигляді овалу з ім'ям, що його характеризує:

Варіанти використання також можуть бути пов'язані між собою.

Зв'язок відображує взаємодію дійових осіб і відповідних варіантів використання. Окремо виділяють зв'язки використання та розширення.

Використання передбачає, що існує деякий фрагмент поведінки створюваного програмного забезпечення, який повторюється в декількох

варіантах використання. Цей фрагмент оформляють як окремий варіант використання і вказують зв'язок з ним типу «include»:



Розширення застосовують, якщо є два подібних варіанти використання, що різняться наявністю в одному з них деяких додаткових дій. У цьому випадку додаткові дії визначають як окремий варіант використання, який пов'язаний з основним варіантом зв'язком типу «extend»:

Діаграми варіантів використання показують взаємодії між варіантами використання і діючими особами, відображаючи функціональні вимоги до системи з точки зору користувача. Таким чином, діаграма варіантів використання є самим загальним уявленням функціональних вимог до системи. Оскільки метою побудови діаграм варіантів використання є документування функціональних вимог у найзагальнішому вигляді, вони повинні бути гранично простими.

2.4. Технічні обмеження, зовнішні інтерфейси та атрибути якості

Як зазначалося раніше, нефункціональні вимоги накладають обмеження на дизайн та реалізацію програмного забезпечення, що розробляється, та виявляють його експлуатаційні якості. Під час визначення нефункціональних вимог у специфікації вимог до програмного забезпечення мають бути описані:

- технічні обмеження;
- зовнішні інтерфейси;
- атрибути (фактори) якості.

2.4.1. Технічні обмеження програмного забезпечення

Технічні обмеження (англ. constraints) стосуються вибору можливості розробки зовнішнього вигляду і структури (в т.ч. архітектури) продукту, мов програмування, технологій та платформ, що будуть використовуватися під час

розробки. Тобто, технічні обмеження – це умови, що обмежують вибір можливих рішень під час реалізації окремих вимог або їх наборів.

2.4.2. Зовнішні інтерфейси програмного забезпечення

Зовнішні інтерфейси (англ. external interfaces) визначають інформацію, яка гарантує, що система буде правильно взаємодіяти з користувачами і компонентами зовнішнього обладнання і зовнішнього програмного забезпечення. Серед зовнішніх інтерфейсів окремо розглядають:

- інтерфейс користувача;
- інтерфейси до зовнішнього апаратного забезпечення;
- інтерфейси до зовнішнього програмного забезпечення;
- комунікаційні інтерфейси.

Вимоги до *інтерфейсу користувача* (англ. user interfaces) визначають логічні характеристики кожного елемента, необхідного для взаємодії користувача з системою:

- посилання на стандарти графічного інтерфейсу користувача або стильові рекомендації для сімейства продуктів, яких необхідно дотримуватися;
- стандарти шрифтів, іконок, назв кнопок, зображень, колірних схем, послідовностей полів вкладок, часто використовуваних елементів управління, повідомлення тощо;
- розміри і конфігурації екранів або обмеження роздільної здатності;
- стандартні кнопки, функції або посилання на перехід, які є однаковими для всього програмного забезпечення (наприклад, кнопка довідки);
- комбінації клавіш для виконання команд;
- стандарти відображення і тексти повідомлень;
- стандарти перевірки даних (такі, як обмеження на введення значень і умови необхідності перевірки вмісту полів);
- стандарти конфігурації інтерфейсу для спрощення локалізації програмного забезпечення;
- спеціальні можливості для користувачів з вадами зору, розрізненням кольору та іншими обмеженнями.

Інтерфейси до зовнішнього апаратного забезпечення (англ. hardware interfaces) визначають характеристики кожного інтерфейсу між компонентами програмного забезпечення та обладнанням системи. Вимогами можуть бути визначені типи підтримуваних пристроїв, взаємодія з даними (вихідні та отримувані дані, їх формат, дозволені значення або їх діапазони) і елементами управління між програмним і апаратним забезпеченнями, а також протоколи взаємодії, які будуть використовуватися.

Інтерфейси до зовнішнього програмного забезпечення (англ. software interfaces) визначають зв'язок програмного забезпечення і інших зовнішніх програмних компонентів (ідентифіковані за іменем та версією), в тому числі інших програм, баз даних, операційних систем, бібліотек, веб-сайтів і інтегрованих компонентів.

Вимогами можуть бути визначені призначення, формати і вміст повідомлень, даних і контрольних значень, обмін якими відбувається між компонентами програмного забезпечення; відповідність між вихідними та отримуваними даними між системами і всі перетворення, які повинні відбуватися з даними при переміщенні між системами; дані, до яких матимуть спільний доступ компоненти програмного забезпечення; служби, необхідні зовнішнім компонентам програмного забезпечення; природа взаємодії між компонентами.

Комунікаційні інтерфейси (англ. communications interfaces) визначають вимоги для будь-яких функцій взаємодії, які будуть використовуватися продуктом, включаючи електронну пошту, веб-браузер, мережеві протоколи і електронні форми. Вимогами можуть бути визначені відповідні формати повідомлень, особливості безпеки взаємодії або шифрування, швидкість передачі даних і механізмів узгодження і синхронізації.

2.4.3. Атрибути якості програмного забезпечення

Атрибути якості (англ. quality attributes) – це вимоги, які визначають якісні характеристики програмного забезпечення, що розробляється.

Для класифікації атрибутів якості можна використовувати багаторівневу модель, що визначена в стандарті ISO 9126 «Software Quality Characteristics». На верхньому рівні виділено шість основних характеристик якості програмного забезпечення:

- функціональність;
- надійність;
- використовуємість;
- ефективність;
- супроводжуємість;
- переносимість.

Кожна з шести основних характеристик описується за допомогою декількох атрибутів, що до неї входять. Для кожного з атрибутів визначається набір метрик, що дозволяють його оцінити.

Функціональність (англ. functionality) – це здатність програмного забезпечення в певних умовах вирішувати завдання, потрібні користувачам. Визначає, що саме робить програмне забезпечення, які завдання воно вирішує. Містить наступні атрибути якості:

- *придатність* (англ. suitability) – це здатність програмного забезпечення забезпечувати відповідний набір функцій для зазначених завдань і цілей користувача;
- *точність* (англ. accuracy) – це здатність програмного забезпечення видавати потрібні результати;
- *здатність до взаємодії* (англ. interoperability) – це здатність програмного забезпечення взаємодіяти з однією або більшою кількістю зазначених систем;
- *захищеність* (англ. sequrity) – це здатність програмного забезпечення захищати інформацію і дані так, щоб не уповноважені суб'єкти або системи не могли читати або змінювати їх, а уповноважені суб'єкти або системи не отримували відмови на доступ до них;
- *узгодженість функціональності* (англ. functionality compliance) – це відповідність програмного забезпечення наявним індустріальним стандартам, нормативним і законодавчим актам, іншим регулюючим нормам.

Надійність (англ. reliability) – це здатність програмного забезпечення підтримувати певну працездатність в заданих умовах. Містить наступні атрибути якості:

- *безвідмовність* (англ. maturity) – це здатність програмного забезпечення запобігати відмовам внаслідок виникнення помилок в програмному забезпеченні;
- *відмовостійкість* (англ. fault tolerance) – це здатність програмного забезпечення підтримувати заданий рівень якості функціонування у випадках виникнення помилок в програмному забезпеченні або під час порушення встановленого інтерфейсу;
- *відновлюваність* (англ. recoverability) – це здатність програмного забезпечення в разі відмови відновлювати рівень якості функціонування і пошкоджені дані;
- *погодженість надійності* (англ. reliability compliance) – це відповідність програмного забезпечення наявним стандартам надійності.

Використовуємість (англ. usability) – це здатність програмного забезпечення бути зручним під час навчання та використання, а також бути привабливим для користувачів. Містить наступні атрибути якості:

- *зрозумілість* (англ. understandability) – це показник, зворотний до зусиль, які витрачаються користувачами на сприйняття основних понять програмного забезпечення і усвідомлення їх застосовності для вирішення своїх завдань;
- *легкість вивчення* (англ. learnability) – це показник, зворотний зусиллям, що витрачаються користувачами на навчання роботі з програмним забезпеченням;

- *оперуємість* (англ. operability) – це показник, зворотний зусиллям, що вживаються користувачами для вирішення своїх завдань за допомогою програмного забезпечення;
- *привабливість* (англ. attractiveness) – це здатність програмного забезпечення бути привабливим для користувачів;
- *погодженість практичності* (англ. usability compliance) – це відповідність програмного забезпечення наявним стандартам зручності використання.

Ефективність (англ. efficiency) – це здатність програмного забезпечення при заданих умовах забезпечувати необхідну працездатність стосовно виділеним для цього ресурсам. Можна визначити її і як відношення одержуваних за допомогою програмного забезпечення результатів до витрачених на це ресурсів усіх типів. Містить наступні атрибути якості:

- *часові характеристики* (англ. time behavior) – це здатність програмного забезпечення видавати очікувані результати, а також забезпечувати передачу необхідного обсягу даних за відведений час;
- *використання ресурсів* (англ. resource utilization) – це здатність вирішувати потрібні завдання з використанням певних обсягів ресурсів (маються на увазі такі ресурси, як оперативна і довгострокова пам'ять, мережеві з'єднання, пристрої введення і виведення тощо);
- *погодженість ефективності* (англ. efficiency compliance) – це відповідність програмного забезпечення наявним стандартам ефективності.

Супроводжуємість (англ. maintainability) – це зручність проведення всіх видів діяльності, пов'язаних із супроводом програмного забезпечення. Містить наступні атрибути якості:

- *аналізуємість* (англ. analyzabily) – це можливість зручного проведення аналізу помилок, дефектів і недоліків, а також зручного аналізу необхідності змін і їх можливих наслідків;
- *змінюваність* (англ. changeability) – це показник, зворотний до трудовитрат, які необхідні на виконання змін програмного забезпечення;
- *стійкість* (англ. stability) – це показник, зворотний ризику виникнення несподіваних ефектів під час внесення до програмного забезпечення необхідних змін;
- *тестуємість* (англ.testability) – це показник, зворотний до трудовитрат, які необхідні на проведення тестування та інших видів перевірки того, що внесені до програмного забезпечення зміни привели до потрібних результатів;
- *узгодженість супроводжуємості* (англ. maintainability compliance) – це відповідність програмного забезпечення наявним стандартам зручності супроводу.

Переносимість (англ. portability) – це здатність програмного забезпечення зберігати працездатність при перенесенні з одного оточення в інше, включаючи організаційні, апаратні і програмні аспекти оточення. Містить наступні атрибути якості:

- *адаптуємість* (англ. adaptability) – це здатність програмного забезпечення пристосовуватися до різних оточень без проведення для цього додаткових дій (крім заздалегідь передбачених);
- *встановлюємість* (англ.installability) – це здатність програмного забезпечення бути встановленим або розгорнутим в певному оточенні;
- *співіснування* (англ. co-existence) – це здатність програмного забезпечення співіснувати з іншими програмами в спільному з ними оточенні, розділяючи з ними спільні ресурси;
- *заміннність* (англ. replaceability) – це можливість застосування програмного забезпечення замість інших програмних систем для вирішення тих же завдань в певному оточенні;
- *узгодженість переносимості* (англ. portability compliance) – це відповідність програмного забезпечення наявним стандартам переносимості.

Контрольні запитання до розділу

1. Що визначають вимоги до програмного забезпечення?
2. Що таке вимога до програмного забезпечення відповідно до IEEE Standard-у?
3. Як класифікувати вимоги за рівнями?
4. Що таке вимоги, визначені на рівні бізнесу, як їх слід розуміти?
5. Навести приклад вимог на рівні бізнесу.
6. Що таке вимоги, визначені на рівні користувача?
7. Навести приклад вимог на рівні користувача.
8. Що таке вимоги, визначені на рівні системи?
9. Навести приклад вимог на рівні системи.
10. Що визначають функціональні вимоги до програмного забезпечення?
11. Що визначають нефункціональні вимоги до програмного забезпечення?
12. Які властивості повинні мати вимоги до програмного забезпечення?
13. Що таке специфікація вимог програмного забезпечення?
14. У тексті яких стандартів, методологій та довідників згадується специфікація вимог програмного забезпечення?
15. У якому стандарті визначаються зміст та якісні характеристики правильно складеної специфікації вимог до програмного забезпечення?
16. З чого починають розробку специфікації вимог до програмного забезпечення?
17. Що має бути виявлено в процесі аналізу вимог до функціональності програмного забезпечення, що створюється?

18. Що є засобом визначення функціональних вимог?
19. За рахунок визначення чого накладаються обмеження на дизайн та реалізацію програмного забезпечення, що створюється, і виявляються його експлуатаційні якості?
20. Чому окремі нефункціональні вимоги вважають більш значимими і критичними, ніж окремі функціональні вимоги?
21. Які є способи документування вимог?
22. Який спосіб документування вимог використовується в більшості проектів з розробки програмного забезпечення і чому?
23. Дотримуючись яких тверджень можна допомогти покращити якість документування вимог до програмного забезпечення?
24. Яких правил слід дотримуватися під час формулювання вимог природною мовою?
25. Чому для багатьох проектів вимоги змінюються аж впритул до моменту завершення створення системи?
26. Що таке варіант використання?
27. Які бувають варіанти використання залежно від мети їх виконання?
28. Який найпростіший випадок визначення варіантів використання?
29. Як може бути визначений варіант використання залежно від рівня абстракції?
30. За рахунок чого можна наочно зображувати очікувану поведінку системи?
31. Що таке дійова особа на діаграмі варіантів використання?
32. Що саме відображує зв'язок між варіантами використання?
33. Що передбачає зв'язок використання на діаграмі варіантів використання?
34. Що передбачає зв'язок розширення на діаграмі варіантів використання?
35. У чому полягають переваги використання варіантів використання під час визначення функціональних вимог до програмного забезпечення?
36. Що таке технічні обмеження програмного забезпечення?
37. Що саме має бути визначено під час опису зовнішніх інтерфейсів програмного забезпечення, яке створюється?
38. На які окремі вимоги розділяють опис зовнішніх інтерфейсів програмного забезпечення, що створюється?
39. Що визначають вимоги до інтерфейсу користувача?
40. Що визначають вимоги до інтерфейсів до зовнішнього апаратного забезпечення?
41. Що визначають вимоги до інтерфейсів до зовнішнього програмного забезпечення?
42. Що визначають вимоги до комунікаційних інтерфейсів?
43. Що таке атрибути якості?
44. Як класифікуються атрибути якості відповідно до стандарту ISO 9126 «Software Quality Characteristics»?
45. Що визначає функціональність програмного забезпечення?
46. Які атрибути якості характеризують функціональність програмного забезпечення?

47. Що визначає придатність як атрибут якості програмного забезпечення?
48. Що визначає точність як атрибут якості програмного забезпечення?
49. Що визначає здатність до взаємодії як атрибут якості програмного забезпечення?
50. Що визначає захищеність як атрибут якості програмного забезпечення?
51. Що визначає надійність програмного забезпечення?
52. Які атрибути якості характеризують надійність програмного забезпечення?
53. Що визначає безвідмовність як атрибут якості програмного забезпечення?
54. Що визначає відмовостійкість як атрибут якості програмного забезпечення?
55. У чому різниця між безвідмовністю та відмовостійкістю програмного забезпечення?
56. Що визначає відновлюваність як атрибут якості програмного забезпечення?
57. Що визначає поняття використовуємості програмного забезпечення?
58. Які атрибути якості характеризують використовуємість програмного забезпечення?
59. Що визначає зрозумілість як атрибут якості програмного забезпечення?
60. Що визначає легкість вивчення як атрибут якості програмного забезпечення?
61. Що визначає оперуємість як атрибут якості програмного забезпечення?
62. Що визначає привабливість як атрибут якості програмного забезпечення?
63. Що визначає ефективність програмного забезпечення?
64. Які атрибути якості характеризують ефективність програмного забезпечення?
65. Що визначають часові характеристики ефективності як атрибут якості програмного забезпечення?
66. Використання яких ресурсів є атрибутом якості програмного забезпечення?
67. Що визначає поняття супроводжуємості програмного забезпечення?
68. Які атрибути якості характеризують супроводжуємість програмного забезпечення?
69. Що визначає аналізуємість як атрибут якості програмного забезпечення?
70. Що визначає змінюваність як атрибут якості програмного забезпечення?
71. Що визначає стійкість як атрибут якості програмного забезпечення?
72. Що визначає тестуємість як атрибут якості програмного забезпечення?
73. Що визначає переносимість програмного забезпечення?
74. Які атрибути якості характеризують переносимість програмного забезпечення?
75. Що визначає адаптуємість як атрибут якості програмного забезпечення?
76. Що визначає встановлюємість як атрибут якості програмного забезпечення?
77. Що визначає співіснування як атрибут якості програмного забезпечення?
78. Що визначає замінність як атрибут якості програмного забезпечення?

РОЗДІЛ 3
ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Після завершення визначення і документування вимог до програмного продукту, який необхідно розробити в курсовій роботі відповідно до варіанта, необхідно спроектувати його архітектуру.

Проектування архітектури програмного продукту, що розробляється в курсовій роботі, має складатися з наступних кроків:

- визначення класів та відношень між ними на діаграмі класів в нотації UML;
- визначення часової послідовності подій на діаграмі послідовності в нотації UML.

Опис архітектури програмного продукту необхідно виконувати відповідно до правил проектування діаграм в нотації UML в середовищі Microsoft Visio, що викладені в цьому розділі.

При проектуванні використовується *архітектурний стиль проектування*, заснований на визначенні основних компонентів програмного забезпечення і зв'язків між ними.

Архітектура програмного забезпечення – високорівневе визначення структури програмного забезпечення, опис якого містить опис логіки окремих компонентів системи і зв'язків між ними та достатній для проведення робіт з написання коду. Під компонентом у цьому визначенні мається на увазі досить довільний структурний елемент програмного забезпечення, який можна виділити шляхом визначення інтерфейсу взаємодії між цим компонентом і всім, що його оточує.

Проектування архітектури майбутнього програмного забезпечення зазвичай розпочинається вже на етапі формування вимог: визначають загальну структуру кожної архітектурної сутності, виконують їх декомпозицію і з'ясовують інтерфейси взаємодії між ними. Таким чином, відбувається розбиття великої складної системи на більш дрібні частини (модулі), що відповідають певному рівню абстракції.

Слід зауважити, що архітектурний компонент може бути визначений порізному в залежності від обраного архітектурного підходу і ступеня деталізації для опису елементів системи.

3.1. Об'єктно-орієнтоване проектування

Під час проектування архітектури програмного забезпечення з використанням об'єктно-орієнтованого підходу розглядають три основні фактори проектування складних систем:

- абстрагування;
- декомпозиція;
- ієрархія.

Абстракція (англ. abstraction) будь-якої системи полягає у створенні моделі цієї системи, в якій навмисно опущені деякі деталі. Вибір тих деталей, які слід опустити, виконують з огляду на аналіз як самого програмного забезпечення, що розробляють, так і його майбутніх користувачів. Мета полягає в тому, щоб дати можливість користувачам звертати увагу лише на ті деталі системи, які важливі для певного програмного забезпечення, а інші деталі ігнорувати.

Таким чином формується узагальнена ідеалізована модель системи. При цьому незначні на певному рівні абстракції елементи моделі можуть уточнюватися на більш нижчих рівнях, і вже на їх основі об'єкти будуть розрізнятися між собою.

Об'єктно-орієнтована *декомпозиція* (англ. decomposition) – це поділ системи, вибравши в якості критерію декомпозиції приналежність її елементів до різних абстракцій даної предметної області.

Оскільки абстракції визначають у вигляді об'єктів, то під час проведення об'єктно-орієнтованої декомпозиції система розбивається на об'єкти, кожен з яких знаходиться у якомусь певному стані. Обмінюючись повідомленнями, об'єкти об'єднуються у систему для виконання спільних дій.

Ієрархія (англ. hierarchy) в об'єктно-орієнтованому підході до проектування є способом розширення інформативності елементів системи.

Ієрархічна схема, відображаючи функції елементів системи, одночасно показує структуру зв'язків між ними.

Ієрархічні структури системи характеризуються, з одного боку, вертикальним керуванням, коли елементи верхнього рівня мають право втручання і координування роботи елементів нижнього рівня. З іншого боку, дії елементів верхнього рівня залежать від інформації, що отримують в результаті функціонування елементів нижніх ієрархічних рівнів. Таким чином, зверху вниз йде, в основному, керуючий вплив, а знизу вгору – інформація про відповідні рішення і зміни.

Визначення ієрархічних зв'язків в системі не завжди є простим завданням, але після їх визначення структура складної системи так само, як і сама система, стають зрозумілішими.

3.1.1. Проектування архітектури

Розробка архітектури програмного забезпечення продовжується до тих пір, поки не будуть виконуватися наступні умови:

- усі варіанти використання реалізовані у вигляді послідовностей обміну повідомленнями між компонентами системи у рамках їх інтерфейсів;
- набір компонентів системи достатній для забезпечення всієї необхідної функціональності, при цьому він достатньо зручний для супроводу і з точки зору переносимості і не викликає помітних проблем з ефективністю;
- кожен компонент має невеликий і чітко визначений набір вирішуваних завдань, при цьому його інтерфейс чітко визначений і збалансований за розміром.

Таким чином, ґрунтуючись на варіантах використання, проводять аналіз характеристик архітектури та оцінку її придатності.

Добре спроектована архітектура володіє наступними властивостями:

- архітектура являє собою багаторівневу систему абстракцій, які співпрацюють одна з одною на кожному рівні абстрагування, мають чітко визначений інтерфейс взаємодії з зовнішнім світом і ґрунтуються на добре продуманих засобах організації нижнього рівня абстрагування;
- на кожному рівні абстрагування інтерфейс абстракції строго відокремлений від реалізації в такий чин, що зміна реалізації не призводить до необхідності змін у інтерфейсі: змінюючись внутрішньо, абстракції продовжують відповідати очікуванням зовнішніх клієнтів;
- архітектура проста, тобто не містить нічого зайвого: спільна поведінка досягається використанням спільних абстракцій та їх взаємозв'язків.

Що стосується помилок проектування архітектури програмного забезпечення, то чим пізніше буде виявлена помилка, тим дорожче обійдеться її виправлення. Тому для дотримання вимог щодо надійності програмного забезпечення, яке розробляється, необхідно вже на стадії проектування архітектури ретельно опрацьовувати зв'язки між архітектурними компонентами і встановлювати ієрархію їх взаємодії. Компоненти, які найбільш часто використовуються або архітектурно пов'язані з безліччю інших компонентів, мають найбільший вплив на надійність системи. Залежні між собою компоненти спричиняють поширення помилок від компонента, в якому вона відбувається, до інших компонентів.

3.1.2. Моделювання системи в проектуванні

Під час проектування переважно використовують моделювання, оскільки з його допомогою легко реалізувати принципи декомпозиції, абстракції та ієрархії. При цьому розглядають побудову системи в такий чин, щоб вона:

- задовольняла специфікації вимог, що була визначена для програмного забезпечення на попередньому етапі його розробки;

- узгоджувалася з обмеженнями, що накладаються апаратним забезпеченням;
- відповідала вимогам, що висуваються безпосередньо до самого процесу розробки програмного забезпечення, (наприклад, тривалість або вартість розробки).

Мова моделювання – це нотація, здебільшого графічна, яка використовується для опису системи, що моделюється. Нотація є сукупність графічних об'єктів, які використовуються в моделях; вона є синтаксисом мови моделювання.

Уніфікована мова моделювання UML передбачає розробку різних моделей, сукупність яких описує систему, що розробляється. Кожна з таких моделей відноситься до відповідного етапу розробки програмного забезпечення і має власне призначення. При цьому кожна з моделей складається з однієї або декількох UML-діаграм, які можуть бути згруповані наступним чином:

- структурні діаграми;
- діаграми поведінки;
- діаграми взаємодії.

На практиці під час розробки програмного забезпечення використовують не всі з наявних діаграм. Дуже часто буває достатнім використовувати лише невеликий набір UML-діаграм для моделювання системи. Найбільш використовуваними діаграмами є діаграми варіантів використання, які розглядалися в попередньому розділі для визначення специфікації вимог, діаграми класів і діаграми послідовності, які будуть розглянуті далі в цьому розділі для задач проектування.

Мова моделювання UML надає можливість моделювати функціональність системи і її поведінку, що є вкрай необхідним під час проектування архітектури майбутнього програмного забезпечення. Діаграми класів є центральною ланкою в об'єктно-орієнтованому проектуванні, що дозволяють визначити типи об'єктів системи і різного роду зв'язки між ними. А діаграми послідовностей, в свою чергу, визначають взаємодію і спільну роботу об'єктів, і таким чином дозволяють моделювати складну поведінку системи за допомогою повідомлень, якими обмінюються об'єкти один з одним в процесі взаємодії.

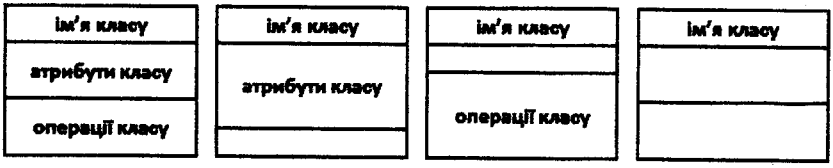
3.2. Визначення класів на діаграмі класів

Діаграми класів визначають класи, інтерфейси і типи даних системи, а також різного роду зв'язки між ними.

Слід зазначити, що не всі визначення нотації UML можуть бути явно реалізовані засобами мови C++. Головне, що слід пам'ятати про мову UML, – це те, що вона є універсальною і не має бути залежною від платформи або реалізації. Саме тому в ній не використовується семантика, специфічна для якоїсь мови програмування. На практиці визначення конкретної семантики домену є однією з робіт проектного архітектора.

3.2.1. Клас у нотації UML

Клас у нотації UML служить для позначення множини об'єктів, які володіють однаковою структурою, поведінкою і відносинами з об'єктами інших класів. Графічно клас зображується у вигляді прямокутника, який додатково розділений горизонтальними лініями на розділи. У цих розділах можуть зазначатися ім'я класу, атрибути (змінні) і операції (методи).

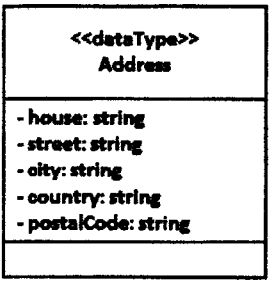


Обов'язковим елементом позначення класу є його ім'я. На початкових етапах проектування програмного забезпечення окремі класи на діаграмі можуть позначатися простим прямокутником із зазначенням тільки імені відповідного класу. По мірі опрацювання окремих компонентів діаграми класів доповнюються атрибутами та операціями. Передбачається, що остаточний варіант діаграми містить найбільш повний опис класів, які складаються з усіх трьох заповнених розділів.

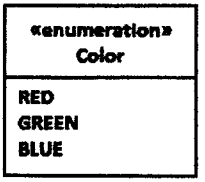
Імені класу може передувати стереотип, який характеризує принципове призначення цього класу і задається окремим рядком одним з ключових слів в лапках:

- *auxiliary* – допоміжний клас, який підтримує інший клас, що зазвичай має стереотип *focus*, за рахунок реалізації додаткової логіки;
- *focus* – основний клас, який визначає основну логіку, підтримувану в допоміжних класах (допоміжні класи можуть бути визначені явно за допомогою стереотипу *auxiliary* чи неявно за допомогою відношення залежності);
- *utility* – клас, який містить лише статичні атрибути та операції;
- *type* – клас, який використовується тільки для специфікації структури та поведінки (але не реалізації) множини об'єктів;
- *interface* – клас, який використовується тільки для опису набору операцій, щоб визначити, що може робити компонент;
- *dataType* – клас, екземпляри якого визначені тільки за їх значеннями (базові елементарні конструкції, що визначені для опису, наприклад, дати, часу, статі, валюти, адреси тощо), тобто основна відмінність *dataType*-класу від звичайного класу полягає в тому, що неможливо мати два екземпляри *dataType*-класу з однаковими значеннями;
- *enumeration* – клас, який визначає перелічувальний тип, включаючи його можливі значення як набір ідентифікаторів.

Наприклад, наступна UML-діаграма має стереотип *dataType* перед іменем класу Address, який означає, що екземпляр цього класу має бути унікальним екземпляром для всієї програми (не існує двох однакових адрес):



Ще один приклад з використання стереотипу перед іменем класу, в якому UML-діаграма визначає перелічуваний тип, що задає три різні значення кольорів:



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми:

```
enum Color { RED, GREEN, BLUE };
```

Під час роботи з класом в програмі може існувати будь-яка кількість його екземплярів. Однак у деяких випадках число екземплярів класу потрібно обмежити, тобто задати кратність класу.

Кратність класу задається виразом, що вказується в правому верхньому куті секції з іменем класу. Зазвичай на практиці використовуються наступні варіанти кратності:

- *кратність 0* (не має екземплярів) – такий клас стає службовим (зі стереотипом *utility*), що містить лише атрибути та операції з областю дії класу, а зберігання інформації, що обробляється службовим класом, забезпечують об'єкти, що його використовують;
- *кратність 1* (клас має рівно один екземпляр) – такий клас називається синглетним і, по суті, не відрізняється від службового (іноді зручніше використовувати синглетні класи, наприклад, коли клас є елементом реального світу, що існує в єдиному екземплярі);

- *фіксована кратність* (клас має фіксоване число екземплярів) – такий варіант не часто, але зустрічається, коли в реальному світі існує завжди фіксоване число екземплярів, наприклад, 8;
- *кратність* * (клас має довільне число екземплярів) – оскільки цей варіант зустрічається найчастіше, він ніяк спеціально не вказується і використовується за замовчуванням.

Клас, у якого не існує безпосередніх екземплярів, хоча у його нащадків їх може бути будь-яка кількість, є абстрактним і потребує, щоб його ім'я на діаграмі класів було виділено курсивом.

Можливість визначити класи, у яких немає нащадків, задається листовими класами, які специфікуються за допомогою завдання властивості *leaf*, що вказується під ім'ям класу у фігурних дужках.

Рідше використовується, хоча і залишається досить корисною, можливість задати клас, який не має батьків. Такий клас називається кореневим і специфікується за допомогою завдання властивості *root*, що також вказується під ім'ям класу у фігурних дужках. Якщо є декілька незалежних ієрархій спадкування, то початок кожної зручно позначати саме в такий чин.

3.2.2. Атрибути класу

У другому зверху розділі прямокутника класу записуються його атрибути. Кожному атрибуту класу відповідає окремий рядок тексту, який має наступний формат:

*[visibility] ['/'] name [':' type] ['[' multiplicity ']'] ['=' default-value] ['{' property-modifier [',' property-modifier] * '}']*

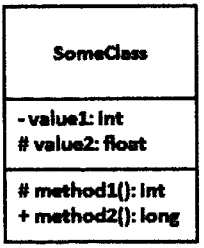
Ім'я атрибута *name* являє собою рядок тексту, який використовується в якості ідентифікатора відповідного атрибута і тому повинен бути унікальним у межах даного класу. Ім'я атрибута є єдиним обов'язковим елементом синтаксичного позначення атрибута.

Квантор видимості атрибута *visibility* може приймати одне з трьох можливих значень і відображається за допомогою спеціальних символів:

- *символ «+»* позначає атрибут з областю видимості типу *public* (загальнодоступний): атрибут із цією областю видимості доступний з будь-якого іншого класу;
- *символ «#»* позначає атрибут з областю видимості типу *protected* (захищений): атрибут із цією областю видимості недоступний для всіх класів, за винятком підкласів даного класу;
- *символ «–»* позначає атрибут з областю видимості типу *private* (закритий): атрибут із цією областю видимості недоступний для всіх класів без винятку.

Квантор видимості може бути опущений. У цьому випадку його відсутність просто означає, що видимість атрибута не вказується.

Наприклад, наступна UML-діаграма визначає клас, що містить атрибути та операції з різними типами області видимості:

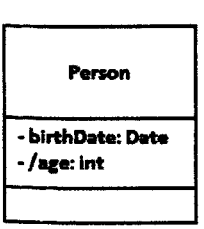


Код, написаний мовою C++ для реалізації зазначеної UML-діаграми:

```
class SomeClass
{
private:
      int value1;

protected:
      float value2;
      int method1();

public:
      long method2();

...

};
```

Символ зворотного слешу «/» позначає, що атрибут є похідним, тобто таким, значення якого обчислюється з іншої інформації, наприклад, шляхом використання значень інших атрибутів.

Наприклад, наступна UML-діаграма визначає клас, що містить атрибут, значення якого обчислюється шляхом використання значення іншого атрибута:



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми, може бути визначений наступним чином:

```
// Person.h
class Person
{
private:
      Date birthDate;
      int age;

...

};

// Person.cpp
...

//---------------------------------------------------------------------------
Person::Person(Date _birthDate) : birthDate(_berthDate)
{
      Date todayDate;
      todayDate.setCurrentDate();
      age = birthDate.getAge(todayDate);

}

...
```

Тип атрибута *type* являє собою вираз, семантика якого визначається мовою специфікації відповідної моделі, тобто визначається залежно від мови програмування, яку будуть використовувати для реалізації даної моделі.

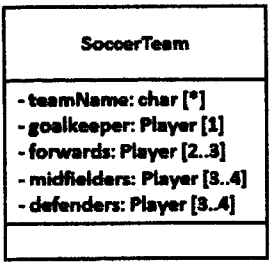
Кратність атрибута *multiplicity* характеризує загальну кількість конкретних значень даного типу, що входять до складу конкретного атрибута. У загальному випадку кратність записується у формі рядка тексту в квадратних дужках після імені відповідного атрибута:

'[' [lower-bound '.. '] upper-bound ']'

де нижня границя *lower-bound* і верхня границя *upper-bound* є додатними цілими числами, які служать для позначення окремого інтервалу цілих чисел від значення нижньої границі до значення верхньої границі.

У якості верхньої границі може використовуватися спеціальний символ «*», який означає довільне додатне ціле число. Іншими словами, це означає необмежене зверху значення кратності відповідного атрибута. Значення кратності з інтервалу знаходяться в монотонно зростаючому порядку без пропуску окремих чисел, що лежать між нижньою і верхньою границями, при цьому відповідні нижні і верхні значення границь включаються до значення кратності. Якщо в якості кратності вказується одне число, то кратність атрибута приймається рівною даному числу. Якщо ж вказується єдиний знак «*», то це означає, що кратність атрибута може бути довільним додатним цілим числом або нулем. Якщо кратність атрибута не вказана, то за замовчуванням приймається її значення, рівне 1..1, тобто рівне одиниці.

Наприклад, наступна UML-діаграма визначає клас, який описує футбольну команду, атрибути якого характеризуються різними значеннями кратності:

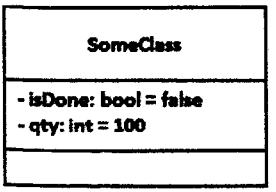


Код, написаний мовою C++ для реалізації зазначеної UML-діаграми, може бути визначений наступним чином:

```
class SoccerTeam
{
private:
      char *teamName;
      Player goalkeeper;
      Player  forwards[3];
      Player midfielders[4];
      Player defenders[4];

...

};
```

Значення за замовчуванням атрибута *default-value* служить для визначення деякого початкового значення для відповідного атрибута в момент створення окремого екземпляра класу.

Наприклад, наступна UML-діаграма визначає клас, в якому для атрибутів задані їх значення за замовчуванням:



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми, може бути визначений наступним чином (зверніть увагу, що механізми мови C++ дозволяють задавати значення за замовчуванням атрибутам класу по-різному):

```
// SomeClass.h
class SomeClass
{
private:
      bool done = false;
      int qty;
public:
      SomeClass();

...

};

// SomeClass.cpp
...

//-------------------------------------------------------------------------
SomeClass:: SomeClass() : qty(100)
{

}

...
```

Під час визначення атрибутів класу можуть бути використані дві додаткові синтаксичні конструкції: підкреслення рядка атрибута і пояснювальний текст у фігурних дужках.

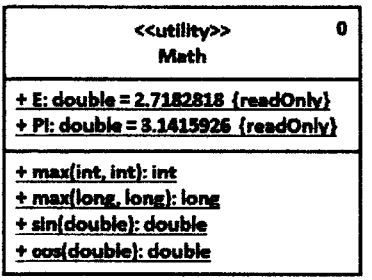
Підкреслення рядка атрибута означає, що відповідний атрибут загальний для всіх об'єктів даного класу, тобто його значення у всіх створюваних об'єктах однакове (аналог ключового слова *static* у мові програмування C++).

Рядок-властивість атрибута *property-modifier* вказує додаткові характеристики, що задають правила визначення значень атрибута в програмі при роботі з даним типом об'єктів:

- *id* позначає, що значення атрибута є частиною ідентифікатора класу, який володіє цим атрибутом;
- *readOnly* позначає, що значення атрибута задається при ініціалізації об'єкта і не може змінюватися (isReadOnly = true);
- *ordered* позначає, що значення атрибута за умови його кратності, більшої за одиницю, впорядковані (isOrdered = true);
- *unique* позначає, що значення атрибута за умови його кратності, більшої за одиницю, не має повторюваних значень (isUnique = true);
- *nonunique* позначає, що значення атрибута за умови його кратності, більшої за одиницю, можуть мати повторювані значення (isUnique = false);
- *sequence* позначає, що значення атрибута за умови його кратності, більшої за одиницю, утворюють впорядковану послідовність (isUnique = false і isOrdered = true).

Відсутність рядка-властивості *property-modifier* за замовчуванням означає, що значення відповідного атрибута може бути змінено в програмі і не має накладених обмежень.

Наступна UML-діаграма визначає клас, який визначений з використання стереотипу *utility* перед іменем класу, отже, клас має містити лише статичні атрибути та операції (на додаток атрибути класу не можуть змінюватися):



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми, може бути визначений наступним чином:

```
// Math.h
class Math
{
public:
      static const double E;
      static const double PI;
      int max(int, int);
      long max(long, long);
      double sin(double);
      double cos(double);

...

};

// Math.cpp
...

const double Math::E = 2.7182818;
const double Math::PI = 3.1415926;

//-------------------------------------------------------------------------
int Math::max( int a, int b )
{
            return a > b ? a : b;
}

...
```

3.2.3. Операції класу

У третьому зверху розділі прямокутника класу записуються операції (методи) класу. Операція являє собою деякий сервіс, що надає кожний екземпляр класу на певну вимогу. Сукупність операцій характеризує функціональний аспект поведінки класу.

Кожній операції класу відповідає окремий рядок тексту, який має наступний формат:

[visibility] operation-name '(' [parameter-list]')' [':' return-spec] ['{' operation-property [',' operation-property] '}']*

Ім'я операції *operation-name* являє собою рядок тексту, що використовується в якості ідентифікатора відповідної операції і тому повинен бути унікальним в межах даного класу. Ім'я операції є єдиним обов'язковим елементом синтаксичного позначення операції.

Квантор видимості операції *visibility* має ті ж самі значення і задає ті ж правила, як і у випадку атрибутів класу.

Список параметрів операції *parameter-list* є переліком розділених комою формальних параметрів, записаних у формі рядка тексту в круглих дужках після імені відповідної операції:

'(' [parameter [',' parameter]]')'*

Кожен з параметрів може бути представлений в наступному форматі:

[direction] parameter-name ':' type-expression ['[' multiplicity ']'] ['=' default-value] ['{' parameter-property [',' parameter-property] '}']*

Напрямок параметра *direction* вказує спосіб передавання значення параметра операції і може бути вказаний ключовим словом *in, out, inout* або *return* зі значенням *in* за замовчуванням у разі, якщо вид параметра не вказується. Ключове слово *in* вказує на те, що значення цього параметра передаються в операцію об'єктом, що викликає операцію. Ключове слово *out* вказує на те, що значення цього параметра передаються об'єкту, що викликає операцію, по закінченню виконання операції. Ключове слово *inout* вказує на те, що значення цього параметра спочатку передаються всередину операції, а потім передаються назовні. Ключове слово *return* вказує на те, що значення переданого в операції параметра буде повернено цією операцією об'єкту, який викликав операцію, при цьому такий вид параметра може бути зазначений лише для одного з параметрів операції.

Ім'я параметра *parameter-name* є ідентифікатором відповідного формального параметра.

Тип параметра *type-expression* є залежним від конкретної мови програмування значенням типу для відповідного формального параметра операції.

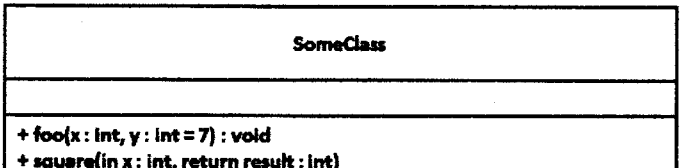
Кратність параметра *multiplicity* характеризує загальну кількість конкретних значень даного типу, що входять до складу зазначеного параметра, і задає ті ж правила, як і у випадку атрибутів класу.

Значення *default-value* в загальному випадку являє собою вираз для визначення значення формального параметра за замовчуванням, синтаксис якого залежить від конкретної мови програмування і підпорядковується прийнятим у ньому обмеженням.

Рядок-властивість параметра операції *parameter-property* вказує додаткові характеристики, що задають такі самі правила визначення значень, як і у випадку атрибутів класу.

Тип поверненого значення операції *return-spec* є залежним від мови реалізації і задає тип значення, що повертається об'єктом після виконання відповідної операції. Двокрапка і вираз типу поверненого значення можуть бути опущені, якщо операція не повертає ніякого значення.

Наступна UML-діаграма визначає клас, в якому для параметрів операцій використовуються різні значення їх напрямку, а також використані значення параметрів за замовчуванням:



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми, може бути визначений наступним чином (зверніть увагу, що механізми мови C++ дозволяють задавати значення напрямку *inout* для параметрів методу по-різному, а реалізація напрямку *out* C++ мовою взагалі неможлива):

```
// SomeClass.h
class SomeClass
{
public:
      void foo(int x, int y = 7);
      int square(int n);
      void textToDate(string t, date &d);
      void toDate(string t, int *yyyy, int *mm, int *dd);

...

};
```

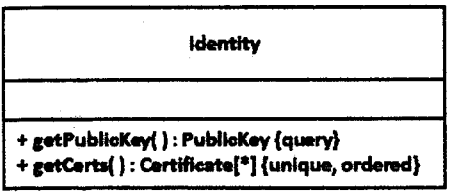
```
// SomeClass.cpp
...

//---------------------------------------------------------------------------
void SomeClass::foo(int x, int y = 7)
{
            cout << "x + y = " << x + y << '\n';
}


//---------------------------------------------------------------------------
int SomeClass::square(int n)
{
      n *= n;
      return n;
}

//---------------------------------------------------------------------------
void SomeClass::textToDate(string t, date &d)
{
      ...
}

//---------------------------------------------------------------------------
void SomeClass::toDate(string t, int *yyyy, int *mm, int *dd)
{
      ...
}

...
```

Рядок-властивість операції *parameter-property* служить для вказування значень властивостей, які можуть бути застосовані до даної операції:

- *redefines opration-name* вказує на те, що дана операція перевизначає успадковану операцію, яка визначена ідентифікатором *opration-name*;
- *query* вказує на те, що виконання даної операції не змінює стану системи, тобто дана операція є просто функцією без побічних ефектів;
- *ordered* вказує на те, що результат, який повертає операція, є впорядкованою послідовністю;
- *unique* вказує на те, що результат, який повертає операція, має унікальне значення для всіх екземплярів класу, до якого належить дана операція;
- *oper-constraint* вказує на додаткове обмеження, яке застосовується до операції і залежить від мови реалізації класу, до якого належить дана операція.

Рядок-властивість не є обов'язковим, він може бути відсутнім, якщо ніякі властивості не специфіковані.

Наступна UML-діаграма визначає клас, в якому для операцій використовуються різні значення рядка-властивості (операція getPublicKey має не змінювати стану системи, а операція getCerts має повертати впорядковану послідовність унікальних значень):



Код, написаний мовою C++ для визначення класу зазначеної UML-діаграми, може бути реалізований наступним чином (реалізацію методів в даному прикладі не наведено, оскільки синтаксис мови C++ не надає для цього випадку якоїсь специфіки):

```
// SomeClass.h
class Identity
{
public:
      PublicKey getPublicKey() const;
      Certificate* getCerts();

...

};
```

3.3. Визначення відношень між класами на діаграмі класів

Крім внутрішньої структури класів, на діаграмі класів також вказують зв'язки (відношення), серед яких розрізняють:

- відношення залежності;
- відношення асоціації;
- відношення узагальнення;
- відношення реалізації.

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

3.3.1. Відношення залежності

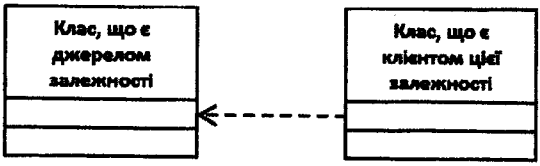
Відношення залежності в загальному випадку використовується, коли деяка зміна одного елемента моделі може вимагати зміни іншого залежного від нього елемента моделі.

Залежність між класами виникає в наступних випадках:

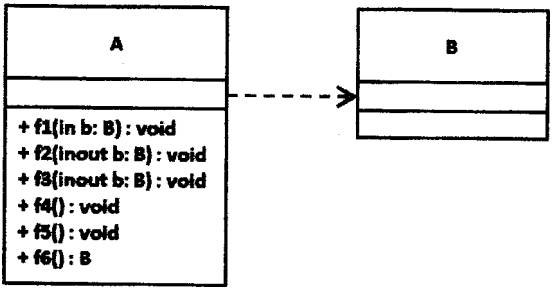
- коли в сигнатурі операції одного класу є аргумент-об'єкт іншого класу;
- коли в методі одного класу є локальний об'єкт іншого класу;
- коли результатом операції одного класу є екземпляр іншого класу.

У всіх цих трьох випадках при виклику операції виникає тимчасовий зв'язок між екземплярами класів, що зникає після закінчення виконання методу.

На діаграмі класів дане відношення зображується пунктирною лінією зі стрілкою на одному з її кінців і пов'язує окремі класи між собою, при цьому стрілка спрямована від класу-клієнта залежності до незалежного класу (класу-джерела):



Наприклад, наступна UML-діаграма визначає два класи, між якими встановлено відношення залежності:



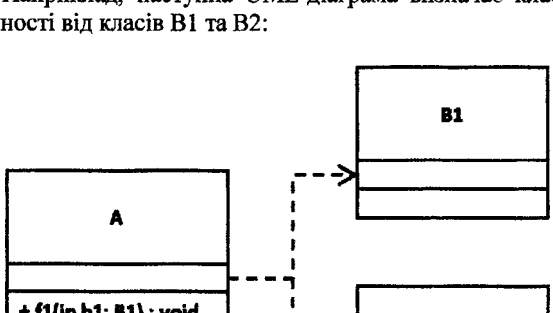
Код, написаний мовою C++ для визначення класів зазначеної UML-діаграми, може бути реалізований наступним чином (наявність будь-якої однієї з подібних операцій, зазначених в прикладі, призводить до встановлення відношення залежності між класами):

```
// A.h
class A
{
public:
      void f1(B b);
      void f2(B *b);
      void f3(Y &b);
      void f4();
      void f5();
      B f6();

...

};
```

```
// A.cpp
...

//---------------------------------------------------------------------------
void A::f1(B b)
{
      ...

      b.Foo();

      ...
}

//---------------------------------------------------------------------------
void A::f2(B *b)
{
      ...

      b->Foo();

      ...
}

//---------------------------------------------------------------------------
void A::f3(Y &b)
{
      ...

      b.Foo();

      ...
}

//---------------------------------------------------------------------------
void A::f4()
{
      B b;
      b.Foo();

      ...
}

//---------------------------------------------------------------------------
void A::f5()
{
      ...

      B::StaticFoo();
}

//---------------------------------------------------------------------------
B A::f6()
{
      ...

      return B();
}

...
```

У якості класу-клієнта і класу-джерела залежності можуть виступати декілька елементів моделі. У цьому випадку одна лінія, що, наприклад, виходить від класу-джерела залежності, розщеплюється в деякій точці на кілька окремих ліній, кожна з яких має окрему стрілку для класу-клієнта.

Наприклад, наступна UML-діаграма визначає клас A, який є клієнтом залежності від класів B1 та B2:



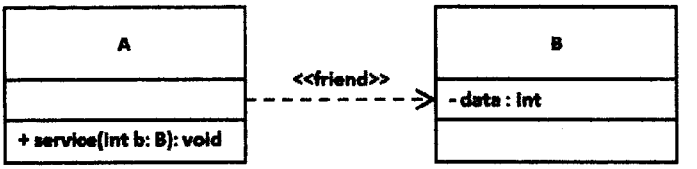
Відношення залежності може мати індивідуальне ім'я, що не є обов'язковим.

Для відношень залежності також передбачені ключові слова, які позначають деякі спеціальні види можливих залежностей:

- *bind* вказує на підстановку параметрів в шаблон: незалежною сутністю є шаблон (клас з параметрами), а залежною – клас, який отриманий з шаблону визначенням шаблонних параметрів;
- *call* вказує на залежність між двома операціями: операція залежного класу викликає операцію незалежного класу;
- *derive* означає, що залежний елемент може бути відновлений за інформацією, що міститься в незалежному елементі (застосовується не тільки до класів, а й до інших елементів моделі, наприклад, атрибутів);
- *friend* вказує на те, що залежний клас має доступ до складових незалежного класу навіть, якщо за загальними правилами видимості такі права у нього відсутні;
- *instantiate* вказує на те, що операції незалежного класу створюють екземпляри залежного класу;
- *use* вказує на залежність самого загального вигляду, яка показує, що залежний клас якимось чином використовує незалежний клас.

Зазначені ключові слова (стереотипи) записуються на діаграмі в лапках поруч зі стрілкою, яка відповідає даній залежності.

Наприклад, наступна UML-діаграма визначає два класи, між якими встановлено відношення залежності, при цьому клас A має доступ до складових класу B незалежно від встановлених правил видимості:



Код, написаний мовою C++ для визначення класів зазначеної UML-діаграми, може бути реалізований наступним чином:

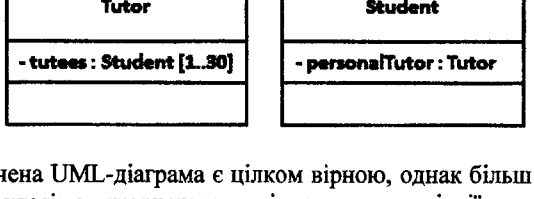
```
// B.h
class B
{
friend class A; // A is a friend of B
private:
      int data;

...

};

//b.h
class A
{
public:
      void service(B b);

...

};

// b.cpp
...

//-----------------------------------------------------------------------
void A::service(B b)
{
      b.data = 0; // legal access due to friendship
}

...
```

3.3.2. Відношення асоціації

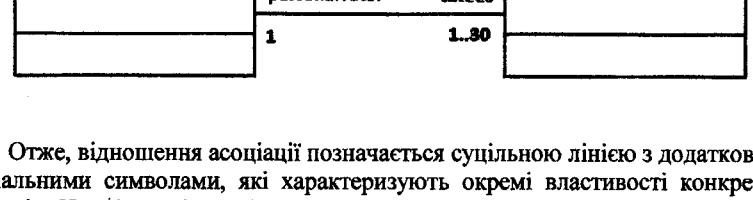
Відношення асоціації відповідає наявності деяких відносин між класами і пов'язане з поняттям атрибута. При наявності асоціації між класами можна вважати, що їх екземпляри з'єднані посиланнями, тобто мають атрибути, значеннями яких є посилання на екземпляри зв'язаних класів.

Як правило, в структурі класу явно вказують атрибути простих типів (числа, символи, рядки, логічні змінні), а атрибути складних типів (значенням яких є екземпляри інших класів) прийнято зображувати як зв'язок асоціації.

Наприклад, наступна UML-діаграма визначає два класи, в кожному з яких є атрибут, що має тип, визначений іншим класом:



Зазначена UML-діаграма є цілком вірною, однак більш доречним є визначення таких класів з використанням відношення асоціації:



Отже, відношення асоціації позначається суцільною лінією з додатковими спеціальними символами, які характеризують окремі властивості конкретної асоціації. На діаграмі класів зображення асоціації (включаючи агрегацію і композицію) характеризуються:

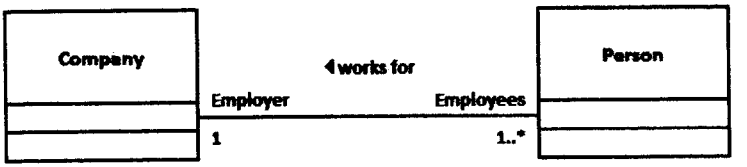
- напрямком;
- ім'ям;
- іменами його кінців-полюсів (задають ролі об'єктів-учасників зв'язку);
- потужностями полюсів.

Напрямок вказує хід повідомлень. За замовчуванням відношення асоціації двонаправлене, тобто повідомлення можуть виходити з будь-якого кінця зв'язку. Якщо введено обмеження за напрямком, то додається стрілка на кінці зв'язку. Іноді перекреслюється хрестом полюс, в бік якого навігація заборонена, тобто не можуть бути передані повідомлення.

Відношення асоціації може мати індивідуальне ім'я, що не є обов'язковим, а полюсам (кінцям) асоціації можуть бути призначені ролі, що також не є обов'язковим.

Наприклад, наступна UML-діаграма визначає два класи з відношенням асоціації між ними, для якого визначено порядок читання «person works for company» (вказує напрям трикутника біля імені асоціації), полюсам відношення

призначені ролі Employer (вказує роботодавця для особи) та Emploee (вказує службовців у компанії):



У наведеному прикладі зображення полюсів асоціації характеризується також визначенням їх потужності.

Потужність показує, як багато об'єктів може брати участь у зв'язку асоціації.

Для кожної асоціації існують два покажчики потужності – по одному на кожному кінці зв'язку. Таким чином, з того боку зв'язку, де приписана потужність, вона позначає кількість об'єктів цього класу, які з'єднані з одним об'єктом класу на іншому кінці зв'язку.

Для позначення потужності використовують зазначення найменшого та найбільшого значення разом або одного значення (якщо мінімум збігається з максимумом чи якщо скорочено записується діапазон 0..*):

- потужність «1» має значення «рівно один»;
- потужність «0..*» (або «*») має значення «нуль або більше»;
- потужність «1..*» має значення «один або багато»;
- потужність «0..1» має значення «нуль або один»;
- потужність «2..4» має значення «від 2-х до 4-х» (заданий діапазон).

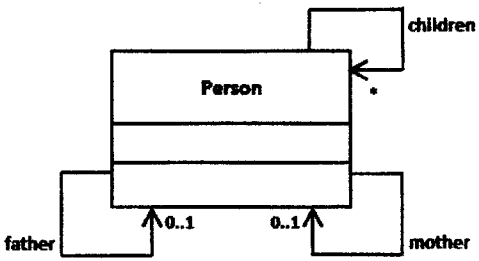
Полюса асоціацій з потужністю «*» мають ще дві характеристики: впорядкованість зв'язаних об'єктів та їх повторюваність. Різні поєднання цих характеристик утворюють чотири типи полюсів:

- *set* тип полюса за замовчуванням, який задає невпорядковану множину об'єктів, що не має однакових елементів;
- *ordered* задає впорядковану множину об'єктів, що не має однакових елементів;
- *bag* задає мультимножину – невпорядковану множину об'єктів, в якій є однакові елементи;
- *sequence* задає послідовність – впорядковану множину об'єктів, в якій є однакові елементи.

Важливо зауважити, що потужності вказують тільки на кінцях відношення асоціації і його підвидах: агрегації і композиції. Для інших зв'язків значення потужності вказувати заборонено.

Відношення асоціації виникає не тільки між лише двома класами (бінарна асоціація): в загальному випадку асоціація n-арна ($n \geq 1$).

Наприклад, наступна UML-діаграма визначає клас, в якому визначений зв'язок асоціації на екземпляр самого себе:



У деяких програмах розробки UML-діаграм можуть бути надані засоби зображення окремих випадків n-арної асоціації (тернарної, 4-арної) за допомогою їх зв'язку ромбом.

Відношенню асоціації може бути приписаний кваліфікатор. Кваліфікатор – це атрибут або набір атрибутів асоціації, значення яких дозволяють вибрати для конкретного об'єкта кваліфікованого класу безліч цільових об'єктів на протилежному кінці з'єднання.

Кваліфікатор зображується у вигляді невеликого прямокутника на полюсі асоціації, що примикає до прямокутника класу. Усередині цього прямокутника (або поруч з ним) вказуються імена і, можливо, типи атрибутів кваліфікатора. Опис кваліфікуючого атрибута асоціації має такий же синтаксис, що й опис звичайного атрибута класу, тільки він не може містити початкового значення.

Основне призначення кваліфікатора – знизити кратність протилежного полюса асоціації, тому загалом він використовується в асоціаціях з кратністю полюсів «один до багатьох» або «багато до багатьох» і стоїть біля полюса, що протилежний полюсу з кратністю «багато». Таким чином, якщо на полюсі асоціації, що протилежний полюсу з кваліфікатором, задана кратність, то вона вказує не допустиму потужність множини об'єктів, приєднаних до полюса зв'язку, а допустиму потужність підмножини, яка визначається при завданні значень атрибутів кваліфікатора.

Наприклад, в наступній UML-діаграмі визначений зв'язок асоціації між двома класами Folder та File, що має кваліфікатор:

Кваліфікатор fileName асоціації папка-файл визначає, що в папці може знаходитися не більше одного файлу з заданим ім'ям. Через наявність кваліфікатора потужність асоціації на полюсі біля класу File дорівнює 0..1, оскільки, фіксуючи на протилежному кінці зв'язку один об'єкт класу Folder і кваліфікатор fileName, можна отримати не більше одного зв'язаного з папкою файлу. Без кваліфікатора потужність була б «0..*».

Спеціальною формою або окремим випадком відносини асоціації є відношення агрегації, яке, в свою чергу, теж має спеціальну форму – відношення композиції.

Відношення агрегації має місце між декількома класами в тому випадку, якщо один з класів являє собою деяку сутність, що включає в себе в якості складових частин інші сутності. Розкриваючи внутрішню структуру системи, відношення агрегації показує, з яких компонентів складається система і як вони пов'язані між собою. З огляду моделі, окремі частини системи можуть виступати як у вигляді елементів, так і у вигляді підсистем, які, у свою чергу, теж можуть утворювати складні компоненти або підсистеми. Це відношення за своєю суттю описує декомпозицію або розбиття складної системи на більш прості складові частини, які також можуть бути розбиті на частини, якщо в цьому виникне необхідність в подальшому.

Графічно відношення агрегації зображується суцільною лінією, один з кінців якої являє собою не зафарбований усередині ромб. Цей ромб вказує на той із класів, який являє собою клас-контейнер («ціле»), а решта класів є його «частинами»:

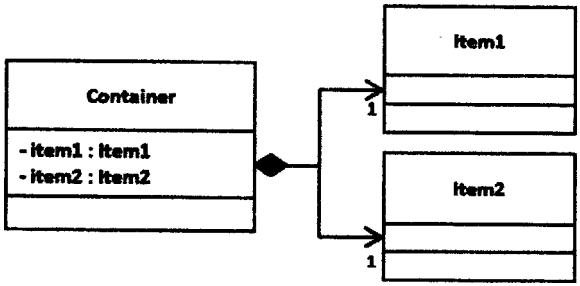
Відношення композиції є окремим випадком відношення агрегації. Це відношення служить для визначення спеціальної форми відносин «частина-ціле», при якій складові частини в деякому розумінні знаходяться всередині цілого. Специфіка взаємозв'язку між ними полягає в тому, що частини не можуть виступати окремо від цілого, тобто зі знищенням цілого знищуються і всі його складові частини. Для композиції справедливі твердження:

- об'єкт-частина в конкретний момент часу належить не більш, ніж одному об'єкту-цілому;
- якщо знищується об'єкт-ціле, то всі його поточні об'єкти-частини також знищуються.

Графічно відношення композиції зображується суцільною лінією, один з кінців якої являє собою зафарбований усередині ромб. Цей ромб вказує на той

із класів, який являє собою клас-контейнер («ціле»), а решта класів є його «частинами»:

Наприклад, наступна UML-діаграма визначає класи, між якими встановлено відношення композиції:



Код, написаний мовою C++ для визначення класів зазначеної UML-діаграми, може бути реалізований наступним чином (зверніть увагу на різницю у визначенні полів та реалізацію конструктора та деструктора):

```
// Container.h
class Container
{
private:
      Item1 item1;
      Item2 *item2;
public:
      Container();
      ~Container();

...

};

// Container.cpp
...

//-------------------------------------------------------------------------
Container::Container()
{
      ...

      item2 = new Item2;

      ...
}
```

```
//---------------------------------------------------------------------
~Container()
{
      ...

      if( item2 != NULL )
            delete item2;

      ...

}

...
```

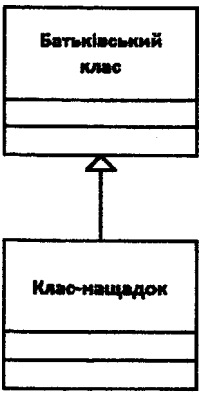
Всі правила зображення на діаграмі класів відношення асоціації поширюються і на форми відношень агрегації та композиції.

3.3.3. Відношення узагальнення

Відношення узагальнення є відношенням між більш загальним елементом (батьком або предком) і більш приватним або спеціальним елементом (нащадком).

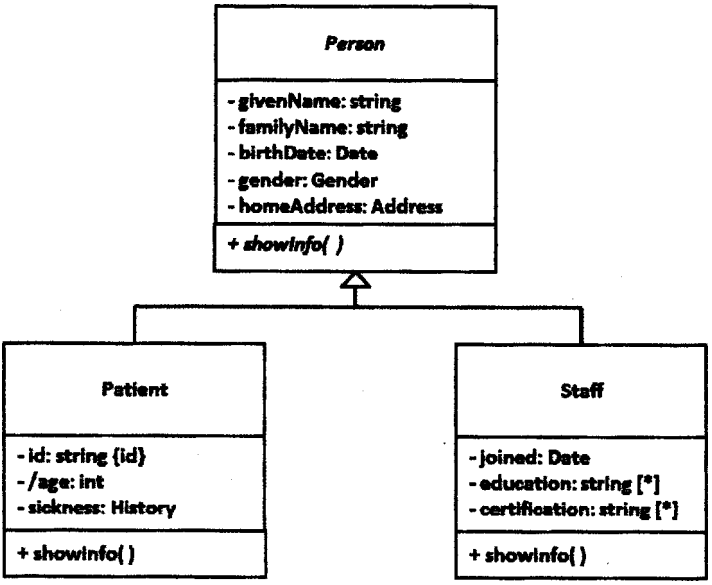
Стосовно діаграми класів відношення узагальнення описує ієрархічну будову класів та успадкування їх властивостей і поведінки. При цьому передбачається, що клас-нащадок має всі властивості і поведінку класу-предка, а також має свої власні властивості і поведінку, які відсутні у класу-предка.

На діаграмі класів відношення узагальнення позначається суцільною лінією з трикутною стрілкою на одному з кінців, де стрілка вказує на більш загальний клас (клас-предок або суперклас), а її відсутність – на більш спеціальний клас (клас-нащадок або підклас):



З метою спрощення позначень на діаграмі класів сукупність ліній, що позначають одне і те ж відношення узагальнення, може бути об'єднана в одну лінію. У цьому випадку окремі лінії сходяться до єдиної стрілки.

Наприклад, наступна UML-діаграма визначає ієрархію класів, у якій базовий клас Person є абстрактним та містить абстрактну операцію showInfo(), а також є два класи-нащадки Patient і Staff:



Код, написаний мовою C++ для визначення класів зазначеної UML-діаграми, може бути реалізований наступним чином:

```
// Person.h
class Person // abstract base class
{
private:
      string givenName;
      string familyName;
      Date birthDate;
      Gender gender;
      Address homeAddress;

public:
      virtual void showInfo() = 0; // pure virtual function

...

};
```

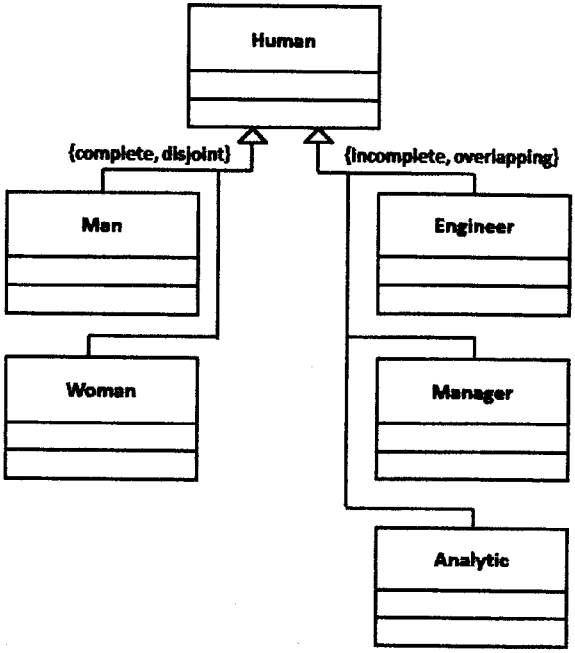
```
// Patient.h
class Patient : public Person
{
private:
      string id;
      int age;
      History sickness;

public:
      void showInfo()

...

};


// Staff.h
class Staff : public Person
{
private:
      Date joined;
      string *education;
      string *certification;

public:
      void showInfo()

...

};
```

Поруч зі стрілкою узагальнення у фігурних дужках може розміщуватися рядок тексту, який вказує на деякі додаткові властивості цього відношення. Цей текст має відношення до всіх ліній узагальнення, що йдуть до класів-нащадків. Іншими словами, зазначена властивість стосується всіх підкласів даного відношення і розглядається як обмеження.

В якості обмежень, які використовуються для відношення узагальнення, можуть бути використані наступні ключові слова:

- *complete* означає, що в даному відношенні узагальнення специфіковані всі класи-нащадки і інших класів-нащадків у даного класу-предка бути не може;
- *incomplete* означає випадок, протилежний першому, тобто передбачається, що на діаграмі вказані не всі класи-нащадки і в подальшому можлива зміна їх переліку, не змінюючи вже побудовану діаграму;
- *disjoint* означає, що класи-нащадки не можуть містити об'єктів, які одночасно є екземплярами двох або більше класів;
- *overlapping* означає, що окремі екземпляри класів-нащадків можуть належати одночасно кільком класам.

Наприклад, наступна UML-діаграма визначає ієрархію класів, в якій використані обмеження на множину узагальнень:



3.3.4. Відношення реалізації

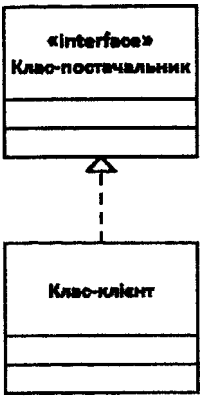
Відношення реалізації – спеціалізоване відношення залежності між двома елементами моделі, в якому один елемент (клієнт) реалізує поведінку, задану іншим елементом (постачальником).

Відношення реалізації зустрічаються між інтерфейсами і класами, що їх реалізують. Інтерфейсом є абстрактний клас, в якому всі складові також абстрактні, тобто такі, які обов'язково повинні з'явитися в класі, що реалізує інтерфейс.

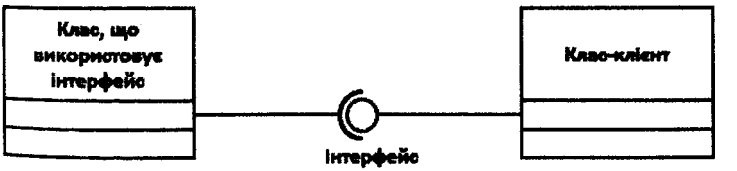
Абстрактні складові, також як і ім'я абстрактного класу, на діаграмі класів виділяються курсивом.

Інтерфейс фактично є описом (без реалізації) групи функцій, які він надає для використання іншому класу. Логіка роботи цих функцій не визначається, а є лише можливість задати неформальний опис того, що від них вимагається. Клас підтримує (або реалізує) інтерфейс, якщо він містить методи, що реалізують всі операції інтерфейсу.

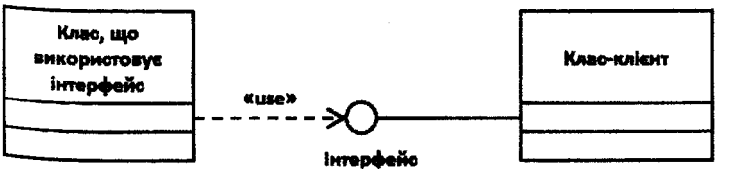
Графічно реалізація представляється так само, як і спадкування, але пунктирною лінією, при цьому стрілка вказує на елемент-постачальник, що задається зі стереотипом інтерфейсу:



Для реалізації інтерфейсів є альтернативна «гніздова» нотація. При її використанні інтерфейс зображується кружком, а зв'язок реалізації – суцільною лінією без стрілки, що йде до нього. В «гніздовій» нотації клас, який використовує інтерфейс, зв'язується з ним або асоціацією з півколом на кінці або залежністю з відповідним стереотипом:



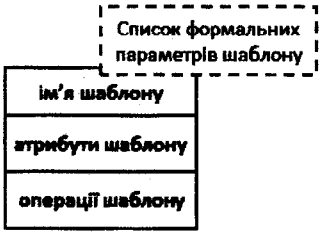
або



У деяких випадках в моделі необхідні класи зі схожою структурою, які відрізняються деякими параметрами – параметрезовані класи (шаблони). Шаблон задає опис безлічі класів з одним або більше невизначеним формальним параметром, тобто це сімейство класів, де кожен клас

відрізняється значеннями цих невизначених параметрів. Таким чином, шаблон визначається в термінах формальних параметрів і, отже, його не можна використовувати безпосередньо як звичайний клас, так як його параметри повинні бути прив'язані до певних значень.

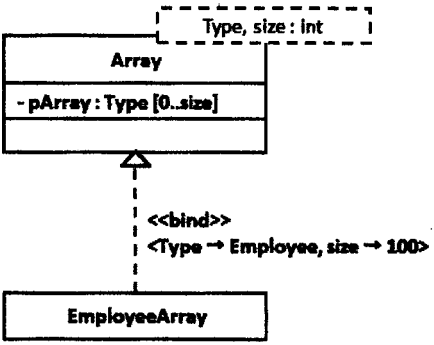
Графічно шаблон зображується прямокутником, до верхнього правого кута якого приєднаний маленький прямокутник з пунктирних ліній. У пунктирному прямокутнику вказується список формальних параметрів шаблону, який не повинен бути порожнім:



Список формальних параметрів шаблону може бути зображений у формі списку, розділеного комами, або представлений одним формальним параметром шаблону в рядку.

Шаблон не може брати участь у більшості звичайних відносин між класами. Існує всього два види відношень, в яких він може брати участь – відношення між шаблоном і класом, що від нього породжений підстановкою параметрів (позначається ключовим словом *bind*) і відношення спрямованої асоціації. Спрямована асоціація повинна йти в напрямку від шаблону (тобто навігація йде в напрямку від шаблону).

Наприклад, наступна UML-діаграма визначає шаблон і клас, що від нього породжений підстановкою параметрів:



Код, написаний мовою C++ для реалізації зазначеної UML-діаграми:

```
template <class Type, int size>
class Array
{
private:
      Type *pArray;

public:
      Array()
      {
            pArray = new Type[size];
      }
      ~Array()
      {
            delete[] pArray;
      }

...

};

void main()
{
      Array< Employee, 100 > EmployeeArray;

      ...

}
```

Діаграми класів можуть застосовуватися і при прямому проектуванні, тобто в процесі розробки нової системи, і при зворотному проектуванні – описі існуючих і використовуваних систем.

У більшості існуючих інструментів UML-моделювання можлива кодогенерація для певної мови програмування (наприклад, C++). Таким чином, діаграма класів – кінцевий результат проектування і відправна точка процесу розробки.

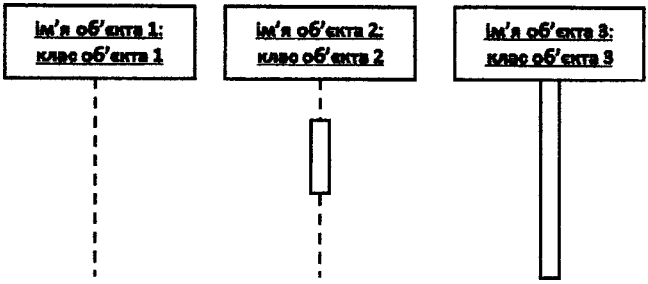
3.4. Розробка часової послідовності подій на діаграмі послідовності

Діаграми послідовності відображають часову послідовність подій, що відбуваються в рамках варіанта використання.

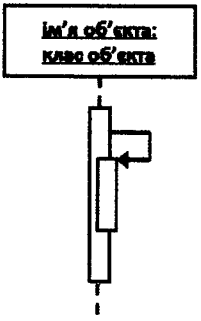
Діаграма послідовності – це діаграма, на якій показані взаємодії об'єктів, упорядковані за часом їхнього прояву в програмному забезпеченні. На діаграмі послідовності зображуються об'єкти, які безпосередньо беруть участь у взаємодії, при цьому ніякі статичні зв'язки з іншими об'єктами не візуалізуються.

3.4.1. Дійові особи і об'єкти

Дійові особи і об'єкти (екземпляри класів) системи представлені на діаграмі лініями життя, які виглядають як прямокутники, від яких вниз проведена пунктирна вертикальна лінія:



Уздовж лінії життя прямокутниками відзначені специфікації виконання (або активації) – періоди, протягом яких відбувається обробка повідомлень. Після отримання повідомлення об'єктом починається його виконання. Під час обробки повідомлень об'єкт може розсилати повідомлення іншим об'єктам або самому собі. Він може бути неактивним під час очікування відповіді на надіслані їм повідомлення, хоча весь цей час активація триває і буде завершена лише після закінчення обробки повідомлення. Протягом однієї активації може бути створена додаткова. Це відбувається, наприклад, тоді, коли об'єкт викликає власну операцію (рекурсивний виклик):



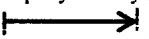
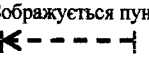
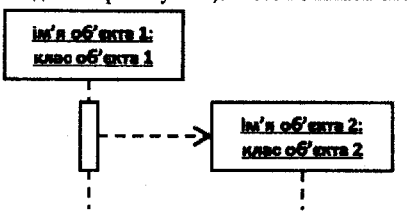
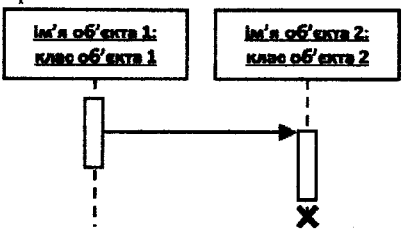
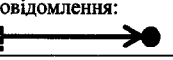
3.4.2. Повідомлення

Кожне повідомлення може бути описано в такому форматі:

[number :][attribute=] operation-name [(argument-list)] [:return-value]

В описі повідомлення номер *number* – порядковий номер повідомлення; змінна *attribute* – явне вказування, де буде збережений результат; ім'я операції *operation-name* – явне вказування викликаної операції і її фактичних параметрів *argument-list*; повернене_значення *return-value* – явне вказування того, що повертається. Будь-яка з частин опису може бути відсутня, крім ім'я операції.

Стрілки, що відповідають повідомленням, які передаються між екземпляром дійової особи і об'єктом або між об'єктами, з'єднують лінії життя відправника і одержувача повідомлення.

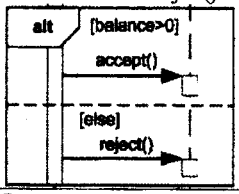
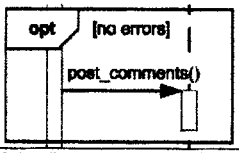
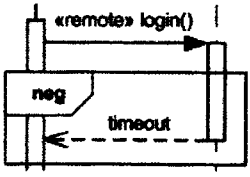
Типи повідомлень	Приклад позначення
Синхронне (виклик процедури)	Позначається суцільною стрілкою з зафарбованим трикутним кінцем: 
Асинхронне (виклик без очікування повернення)	Зображується суцільною стрілкою зі звичайним кінцем: 
Повернення (відповідь на синхронне повідомлення)	Зображується пунктирною стрілкою: 
Повідомлення на створення	Входить в прямокутник,з якого починається лінія життя: 
Повідомлення на видалення	За стрілкою завершується лінія життя, що вказується «хрестиком»: 
Знайдене повідомлення	Зображується без зазначення лінії життя, що відправила повідомлення: 
Втрачене повідомлення	Зображується без зазначення лінії життя, яка приймає повідомлення: 

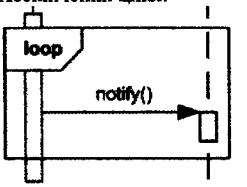
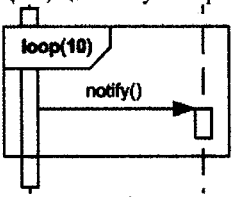
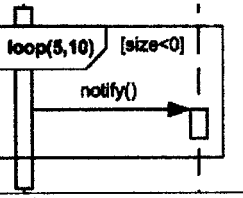
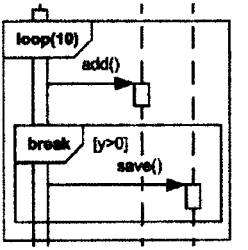
Порядок відправки повідомлень відповідає їх розміщенню на діаграмі зверху вниз. Вище знаходяться повідомлення, відправлені раніше.

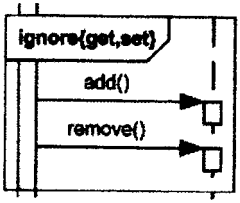
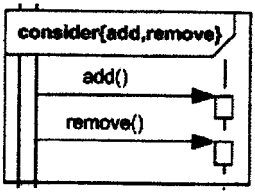
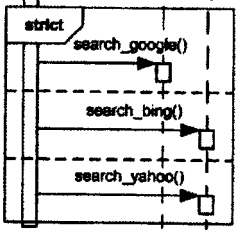
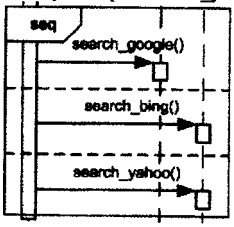
Таким чином, для діаграми послідовності головним є саме динаміка взаємодії об'єктів у часі, яку відображують два виміри діаграми: один вимір простягається зверху вниз у вигляді вертикальних ліній, кожна з яких зображує лінію життя окремого об'єкта; другий – горизонтальна часова вісь, спрямована зліва направо, що відображує взаємодію об'єктів в часі.

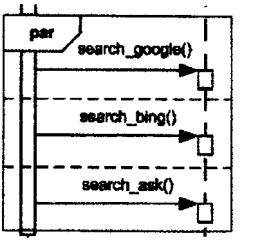
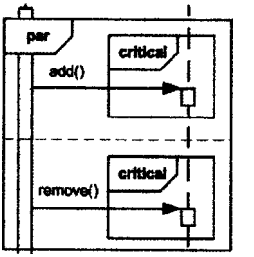
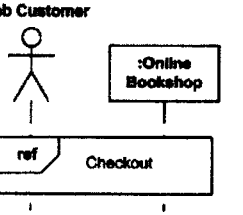
3.4.3. Комбіновані фрагменти

Діаграми послідовності можуть містити комбіновані фрагменти взаємодії, що описують більш складні випадки, ніж простий обмін повідомленнями. Кожен фрагмент такої взаємодії буде містити один або більше операндів взаємодії, що з'єднані оператором взаємодії.

Оператор взаємодії	Приклад використання
Кілька альтернатив *alt*: кожна альтернатива представляється окремим операндом, кожен операнд забезпечується сторожовою умовою, фрагмент перевіряє всі сторожові умови і залежно від істинності сторожових умов обирає один операнд взаємодії і виконує його, а якщо є кілька альтернатив з істинними сторожовими умовами, то вибір здійснюється не детерміновано	Виклик accept() за умови, якщо balance > 0, інакше – виклик reject(): 
Необов'язкова взаємодія *opt*: єдиний заданий операнд виконується тільки тоді, коли сторожова умова істинна, інакше не робиться нічого	Виклик post_comments() за відсутності помилок: 
Невірна (неприпустима) взаємодія neg	Має бути повернено timeout повідомлення, що означає збій системи: 

Оператор взаємодії	Приклад використання
Цикл *loop*: єдиний операнд є тілом циклу, в сторожовій умові може бути зазначено мінімальну та максимальну кількість ітерацій і / або умову зупинки виконання циклу	Нескінчений цикл:  Цикл, що виконується рівно 10 разів:  Цикл, який очікувано буде виконуватися не менше 5 і не більше 10 разів. Якщо сторожова умова [size < 0] стає хибною, то цикл завершується незалежно від мінімального числа зазначених ітерацій: 
Достроковий вихід з циклу *break*: зустрічається всередині loop-фрагмента та забезпечує виконання свого єдиного операнда взаємодії для дострокового виходу з циклу при істинній сторожовій умові	Break завершує цикл за умови, якщо y>0: 

Оператор взаємодії	Приклад використання
Оператор взаємодії ігнорування *ignore*: деякі повідомлення, які не показані у фрагменті, можуть вважатися незначними і неявно ігноруються, якщо вони з'являються у зазначеній послідовності подій	Ігноруються get() і set() повідомлення за їх наявності: 
Оператор взаємодії розглядання *consider*: визначає повідомлення, що повинні бути розглянуті в рамках фрагмента, а будь-які інші повідомлення будуть проігноровані	Розглядаються лише add() та remove() повідомлення, ігноруючи всі інші: 
Суворе *strict* або слабке *seq* впорядкування операндів взаємодії: при суворому впорядкуванні все виконується послідовно (перший операнд, за ним другий), а при слабкому впорядкуванні впорядковуються тільки повідомлення, одержувані кожною лінією життя	Виклики search_google(), search_bing() та searchyahoo() мають бути виконані в суворому порядку слідування:  Виклик search_google() може бути виконаний паралельно з search_bing() та search_yahoo(), але виклик search_bing() має бути перед search_yahoo(): 

Оператор взаємодії	Приклад використання
Паралельно виконувані операнди взаємодії par	Виклики search_google(), search_bing() та search_ask() можуть бути викликані паралельно: 
Критична ділянка *critical*	Add() або remove() можуть бути викликані паралельно, але кожне з цих повідомлень є критичною ділянкою: 
Оператор взаємодії *ref*: використання взаємодії-посилання на іншу діаграму, де детально моделюється один з етапів взаємодії	Web Customer та Bookshop використовують (посилаються на) взаємодії, що визначені на діаграмі Checkout: 

Контрольні запитання до розділу

1. Що таке архітектурний стиль проектування програмного забезпечення?
2. Дати визначення архітектурі програмного забезпечення.
3. Що саме виконується щодо розробки архітектури програмного забезпечення ще на етапі формування вимог?
4. Від чого залежить визначення архітектурних компонентів програмного забезпечення, що розробляється?
5. Які три основні фактори проектування складних систем розглядають під час проектування архітектури програмного забезпечення з використанням об'єктно-орієнтованого підходу?
6. У чому полягає поняття абстракції системи?
7. Що таке об'єктно-орієнтована декомпозиція?
8. Що означає поняття ієрархії під час проектування програмного забезпечення?
9. Якими властивостями має володіти добре спроектована архітектура програмного забезпечення?
10. Які компоненти системи мають найбільший вплив на її надійність?
11. Чому під час проектування програмного забезпечення переважно використовується моделювання системи, що розробляється?
12. Що таке мова моделювання?
13. Розробка чого передбачена уніфікованою мовою моделювання UML?
14. На які три групи поділяються UML-діаграми?
15. Які з UML-діаграм є найбільш використовуваними?
16. Що визначає діаграма класів?
17. Що визначає клас в мові UML, як він позначається графічно?
18. Який елемент є обов'язковим для позначення класу на діаграмі класів у нотації UML?
19. Як можуть позначатися окремі класи на діаграмі класів у нотації UML на початкових етапах проектування програмного забезпечення?
20. Чому не всі визначення нотації UML можуть бути явно реалізовані засобами мови C++?
21. Що характеризує значення стереотипу, яке передує імені класу на діаграмі класів у нотації UML?
22. Як визначити клас на діаграмі класів у нотації UML, який містить лише статичні атрибути та операції?
23. Як визначити клас на діаграмі класів у нотації UML, який використовується тільки для опису набору операції, щоб визначити, що може робити компонент?
24. Як визначити клас на діаграмі класів у нотації UML, який визначає перелічувальний тип?
25. Як у нотації UML обмежити кількість екземплярів класу на діаграмі класів?

26. Як визначити клас на діаграмі класів у нотації UML, у якого не існує безпосередніх екземплярів?
27. Як визначити клас на діаграмі класів у нотації UML, у якого немає нащадків?
28. Як визначити клас на діаграмі класів у нотації UML, у якого немає батьків?
29. Який формат має рядок, що визначає атрибут класу на діаграмі класів відповідно до нотації UML?
30. Що є єдиним обов'язковим елементом синтаксичного позначення атрибута класу на діаграмі класів відповідно до нотації UML?
31. Що визначає квантор видимості атрибута класу на діаграмі класів відповідно до нотації UML, які значення він може приймати?
32. Що значить, що атрибут класу є похідним, як це позначається відповідно до нотації UML?
33. Що характеризує кратність атрибута класу на діаграмі класів відповідно до нотації UML?
34. Що значить, коли в якості верхньої границі для позначення кратності атрибута класу на діаграмі класів відповідно до нотації UML використовується спеціальний символ «*»?
35. Яке значення приймається за замовчуванням, якщо кратність атрибута класу на діаграмі класів відповідно до нотації UML не вказана?
36. Як відповідно до нотації UML на діаграмі класів задати початкове значення для атрибута в момент створення окремого екземпляра класу?
37. Як відповідно до нотації UML на діаграмі класів задати атрибут, загальний для всіх об'єктів даного класу, тобто такий, значення якого буде однаковим для всіх створюваних об'єктів?
38. Які додаткові характеристики визначає рядок-властивість атрибута класу на діаграмі класів відповідно до нотації UML?
39. Що відповідно до нотації UML являє собою операція класу?
40. Що відповідно до нотації UML визначає сукупність операцій класу?
41. Який формат має рядок, що визначає операцію класу на діаграмі класів відповідно до нотації UML?
42. Що є єдиним обов'язковим елементом синтаксичного позначення операції класу на діаграмі класів відповідно до нотації UML?
43. Як відповідно до нотації UML задати список параметрів операції на діаграмі класів?
44. Який формат має рядок, що визначає параметр операції класу на діаграмі класів відповідно до нотації UML?
45. Що відповідно до нотації UML визначає напрямок параметра операції на діаграмі класів?
46. Які додаткові властивості операції визначає рядок-властивість операції класу на діаграмі класів відповідно до нотації UML?
47. Як визначити операцію класу на діаграмі класів у нотації UML, яка перевизначає успадковану операцію?

48. Як визначити операцію класу на діаграмі класів у нотації UML, виконання якої не змінює стану системи?
49. Як визначити операцію класу на діаграмі класів у нотації UML, результатом виконання якої є впорядкована послідовність?
50. Як визначити операцію класу на діаграмі класів у нотації UML, результатом виконання якої є унікальне значення для всіх екземплярів класу, до якого належить дана операція?
51. Які розрізняють зв'язки між класами, що позначаються на діаграмі класів в нотації UML?
52. У яких випадках між класами виникає відношення залежності?
53. Як зображується відношення залежності між класами на діаграмі класів у нотації UML?
54. Чи можуть відповідно до нотації UML в якості класу-клієнта і класу-джерела залежності виступати декілька елементів моделі?
55. Які існують спеціальні види залежностей між класами відповідно до нотації UML?
56. Як у нотації UML визначити на діаграмі класів залежність, відповідно до якої клас має доступ до складових незалежного класу, навіть якщо за загальними правилами видимості такі права у нього відсутні?
57. Як у нотації UML визначити на діаграмі класів, що операції незалежного класу створюють екземпляри залежного класу?
58. У яких випадках між класами виникає відношення асоціації?
59. Як зображується відношення асоціації між класами на діаграмі класів у нотації UML?
60. Що вказує напрямок відношення асоціації відповідно до нотації UML?
61. Що визначає потужність у відношенні асоціації відповідно до нотації UML?
62. Якими додатковими характеристиками володіють полюси асоціацій з потужністю «*» відповідно до нотації UML?
63. Чи може відповідно до нотації UML бути визначено значення потужності для інших зв'язків між класами, окрім асоціації?
64. Як у нотації UML зобразити на діаграмі класів клас, в якому визначений зв'язок асоціації на екземпляр самого себе?
65. Що таке кваліфікатор відношення асоціації відповідно до нотації UML, яке його основне призначення?
66. Як зображується кваліфікатор відношення асоціації між класами на діаграмі класів у нотації UML?
67. Які є підвиди відношення асоціації?
68. У яких випадках між класами виникає відношення агрегації?
69. Як зображується відношення агрегації між класами на діаграмі класів у нотації UML?
70. У яких випадках між класами виникає відношення композиції?
71. Як зображується відношення композиції між класами на діаграмі класів у нотації UML?

72. Чи поширюються правила зображення на діаграмі класів відношення асоціації на форми відношень агрегації та композиції?
73. У яких випадках між класами виникає відношення узагальнення?
74. Як зображується відношення узагальнення між класами на діаграмі класів у нотації UML?
75. Як у нотації UML зобразити на діаграмі класів відношення, в якому специфіковані всі класи-нащадки і інших класів-нащадків у класу-предка бути не може?
76. Як у нотації UML зобразити на діаграмі класів відношення, в якому класи-нащадки не можуть містити об'єктів, які одночасно є екземплярами двох або більше класів?
77. У яких випадках між класами виникає відношення реалізації?
78. Що таке клас, який є інтерфейсом?
79. Як зображується відношення реалізації між класами на діаграмі класів у нотації UML?
80. Як зображується відношення реалізації між класами на діаграмі класів відповідно до альтернативної «гніздової» нотації?
81. Як у нотації UML зобразити на діаграмі класів клас, який є шаблоном?
82. У яких двох видах відносин між класами може брати участь клас, який є шаблоном?
83. Що визначає діаграма послідовності?
84. Що відповідно до нотації UML зображується на діаграмі послідовності?
85. Що таке відповідно до нотації UML специфікації виконання (активації), як вони позначаються на діаграмі послідовності?
86. Як у нотації UML позначити рекурсивний виклик на діаграмі послідовності?
87. Який формат опису повідомлення на діаграмі послідовності відповідно до нотації UML?
88. Які відповідно до нотації UML існують типи повідомлень, що визначаються на діаграмі послідовності?
89. Як у нотації UML позначити виклик без очікування повернення на діаграмі послідовності?
90. Як у нотації UML позначити відповідь на синхронне повідомлення на діаграмі послідовності?
91. Як у нотації UML позначити повідомлення на створення лінії життя на діаграмі послідовності?
92. Як у нотації UML позначити повідомлення на видалення лінії життя на діаграмі послідовності?
93. Що таке знайдене повідомлення відповідно до нотації UML, коли воно виникає?
94. Що таке втрачене повідомлення відповідно до нотації UML, коли воно виникає?
95. У якому порядку виконується відправка повідомлень на діаграмі послідовності відповідно до нотації UML?

96. Як у нотації UML позначити комбінований фрагмент, який визначає кілька альтернатив?
97. Як у нотації UML позначити комбінований фрагмент, який визначає цикл і достроковий вихід із нього?
98. Як у нотації UML позначити комбінований фрагмент, який визначає паралельно виконувані операнди?

РОЗДІЛ 4
РЕАЛІЗАЦІЯ КОДУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Реалізацію програмного продукту в курсовій роботі необхідно виконувати мовою програмування високого рівня C++ в середовищі Microsoft Visual Studio.

Написання програмного коду слід виконувати частинами в чотири етапи:

- визначення класів та програмування меню користувача;
- створення об'єктів та використання контейнерів;
- організація роботи з даними через файл;
- пошук даних у контейнері.

Кожна частина логічно завершена та дозволяє відлагодити та перевірити розроблену на певному етапі програму.

Основною вимогою до програмної реалізації є обов'язкове використання об'єктно-орієнтованого підходу. Критеріями використання об'єктно-орієнтованого підходу є використання класів, механізмів керування пам'яттю і винятками, механізмів успадкування, інкапсуляції та поліморфізму.

Велика частина роботи програмістів пов'язана з написанням програмного коду, тестуванням і відлагодженням програм на одній з мов програмування.

Різні мови програмування підтримують різні стилі програмування (парадигми програмування). Вибір потрібної мови програмування дозволяє скоротити час написання програми і вирішити задачі реалізації алгоритмів програми найбільш ефективно. Різні мови вимагають від програміста різного рівня уваги до деталей під час реалізації, результатом чого часто буває компроміс між простотою і продуктивністю.

Мову програмування можна назвати об'єктно-орієнтованою тільки тоді, якщо у ній реалізовані наступні механізми:

- підтримка об'єктів, тобто абстракції даних, які мають інтерфейс у вигляді іменованих операцій, і власні дані з обмеженим доступом до них;
- об'єкти відносяться до відповідних типів;
- типи можуть успадковувати атрибути надтипів (надкласів).

За цією класифікацією до об'єктно-орієнтованих мов програмування можна віднести мову програмування C++. Цій мові, як об'єктно-орієнтованій мові програмування, характерна наявність:

- абстрактних типів даних;
- приховування реалізації зовнішнього інтерфейсу (інкапсуляція);
- успадкування властивостей і поведінки об'єктів;
- динамічне зв'язування імені зі значенням;
- поліморфізм імен повідомлень;
- автоматичне керування пам'яттю.

Мова програмування визначає синтаксис і початкову семантику вихідного коду програмного забезпечення.

Написання коду на обраній мові програмування може бути виконано в будь-якому текстовому редакторі. Однак для цих цілей сьогодні широко використовуються інтегровані середовища розробки – комплексні програмні рішення для розробки програмного забезпечення.

Інтегровані середовища розробки створені для того, щоб максимізувати продуктивність програміста, надавши йому необхідні інструменти розробки зі схожими інтерфейсами. Тобто інтегроване середовище розробки – це одна програма, в якій відбуватиметься весь процес розробки, і яка надає необхідні функції для модифікації, компілювання, запуску та відлагодження програмного забезпечення.

Деякі інтегровані середовища розробки призначені для використання певної мови програмування (або декількох споріднених мов), надаючи набір можливостей, які більш підходять до парадигми програмування відповідної мови. Такими середовищами є, наприклад, PhpStorm, Xcode, Xojo та Delphi. З іншої сторони, існує чимало більш універсальних середовищ, які є багатомовними, наприклад, Microsoft Visual Studio.

Сьогодні до складу інтегрованих середовищ розробки зазвичай входять:

- редактор коду для введення і редагування тексту програм, який може мати специфічну функціональність, таку, як індексація імен, відображення документації, підсвічування синтаксису, засоби візуального створення призначеного для користувача інтерфейсу;
- відлагоджувач для відлагодження (виявлення і усунення помилок);
- транслятор для перетворення тексту програми в машинний код;
- компонувальник для збірки програми з декількох модулів.

Іноді інтегровані середовища розробки можуть містити також засоби для інтеграції з системами управління версіями і різноманітні інструменти для спрощення конструювання графічного інтерфейсу користувача. Багато сучасних середовищ розробки також містять браузер класів, інспектор об'єктів і інструменти побудови діаграм ієрархії класів для використання під час проведення об'єктно-орієнтованої розробки програмного забезпечення.

З метою полегшення виправлення помилок на етапі компіляції в курсовій роботі необхідно дотримуватися розробки програмного коду в чотири етапи. Після завершення кожного з етапів необхідно провести компіляцію програмного продукту та виправити всі наявні помилки, користуючись засобами інтегрованого відлагоджувача в Microsoft Visual Studio. Основні принципи роботи з відлагоджувачем в Microsoft Visual Studio наведені в додатку Б. Як результат, після виправлення помилок на етапі компіляції розроблений програмний код має успішно компілюватися, а програма має запускатися на виконання.

4.1. Визначення класів та програмування меню користувача

Визначення класів ієрархії, що зазначена у варіанті, та програмування меню користувача є першою частиною реалізації програмного продукту, який розробляється в курсовій роботі.

4.1.1. Визначення класів ієрархії

У першій частині код для ієрархії класів слід розробити з визначенням необхідних для роботи з цими класами полів і методів. Зверху розробленої ієрархії класів має бути абстрактний клас. Поліморфізм слід реалізувати за допомогою віртуальних функцій та використати на цьому етапі, наприклад, для методів відображення екземпляра класу.

Приклад ієрархії класів:

```
//абстрактний клас
class person
{
protected:
      string name;
      int age;
public:
      person(string _name, int _age);
      void setName(string _name);
      string getName()const;
      void setAge(int _age);
      int getAge()const;
      virtual void show()const;
      virtual ~person();
};

//клас «робітник»
class worker : public person
{
private:
      int rang;
public:
      worker(string _name, int _age, int _rang);
      void setRang(int _rang);
      int getRang()const;
      void show()const;
      ~worker();
};
```

```
//клас «службовець»
class employer : public person
{
private:
      string post;
public:
      employer(string _name, int _age, string _post);
      void setPost(string _post);
      string getPost()const;
      void show()const;
      ~employer();
};
```

Визначення класів слід розмістити у файлах *.h, а визначення методів класу поміщається у файлах *.cpp.

Директива препроцесора *#include* додає в текст програми вміст зазначеного файлу в ту точку коду, де вона записана. Текст файлу, що додається, також може містити директиви *#include*. Ця директива може зустрічатися в будь-якому місці коду програми, але зазвичай всі включення розміщуються на початку.

Директива має дві форми:

#include "ім'я_файлу"
#include <ім'я_файлу>

Рядок, що задає *ім'я_файлу*, може складатися або тільки з імені файлу, або з імені файлу та передуючого йому шляху. Якщо *ім'я_файлу* вказано в лапках, то пошук файлу здійснюється відповідно до заданого шляху, а при його відсутності – в поточному каталозі. Якщо *ім'я_файлу* задано в кутових дужках, то пошук файлу проводиться в стандартних каталогах (для їх налаштування служить опція Options → Directories ... → Include Directories).

Під час визначення полів слід враховувати можливість виконання запиту, що має бути розроблений в четвертій частині роботи.

Спочатку для методів усіх класів слід розробити тільки конструктор з параметрами та перевірити код. Ініціалізацію полів класу необхідно виконувати з використанням синтаксису через списки ініціалізації конструкторів. Оскільки розробка наступних частин може потребувати ініціалізації об'єктів не під час створення, а пізніше, в кожному з класів ієрархії слід розробити *set*-методи та *get*-методи до кожного поля класу.

Методи класів, що не змінюють стан об'єкта, для якого вони викликаються, слід оголошувати як const-методи.

Під час реалізації методів ієрархії класів слід пам'ятати про переваги використання принципу спадкування, який дозволяє класам-нащадкам отримувати властивості та поведінку базового класу, доповнюючи їх своїми власними. Таким чином, спадкування дозволить покращити повторне

використання коду шляхом використання в класах-нащадках вже визначених властивостей та методів базового класу.

4.1.2. Меню користувача

Після створення і відлагодження ієрархії класів слід розробити клас програми, що буде виконувати основну роботу:

- створення об'єктів і розміщення їх в контейнері;
- перегляд вмісту контейнера;
- збереження вмісту контейнера у файлі;
- завантаження до контейнера послідовності об'єктів з файлу;
- очищення контейнера;
- сортування послідовності об'єктів у контейнері;
- виконання запиту.

Можливі різні варіанти реалізації класу програми.

Об'єкт контейнера може бути визначений як статичне поле класу програми, і в цьому випадку методи ініціалізації, виконання та звільнення ресурсів визначаються як статичні *public* методи цього ж класу. Інші методи, що вимагають доступу до даних контейнера, визначаються як статичні *private* методи.

У іншому випадку об'єкт контейнера може бути визначений як нестатичне поле класу програми (це може бути покажчик на об'єкт контейнера). У цьому випадку методи класу програми визначаються як нестатичні.

Виклик меню користувача слід розмістити у функції виконання класу програми. Меню має передбачати виклик функцій – членів класу програми для виконання різного виду робіт.

Приклад організації меню користувача:

```
void Run()
{
      int i;
      char s[10];

      do
      {
            system("cls");
            cout<<"-------------- MENU --------------"<<endl;
            cout<<"<1>.Create worker`s object"<<endl;
            cout<<"<2>.Create employer`s object"<<endl;
            cout<<"<3>.Show the factory"<<endl;
            cout<<"<4>.Remove the factory"<<endl;
            cout<<"<5>.Save to the file"<<endl;
            cout<<"<6>.Load from the file"<<endl;
            cout<<"<7>.Sort objects"<<endl;
            cout<<"<8>.Do request"<<endl;
            cout<<"<9>.Leav the program"<<endl<<endl;

            cin.getline(s,10);
            i=atoi(s);
```

```
            switch(i)
            {
                  case 1:
                  {
                        system("cls");
                        CreateWorker();
                        system("pause");
                        break;
                  }

                  //і так далі...

                  default:
                  {
                        if(i>10 || i<1)
                        {
                              cout<<"Your choise is not correct..."<<endl;
                              system("pause");
                        }
                        break;
                  }

            }
      }
      while (i!=10);
}
```

На цьому етапі розробки програмного продукту функції мають бути реалізовані у вигляді заглушок і кожен їх виклик має супроводжуватися виведенням повідомлення.

У функції main() мають бути реалізовані:

- функція ініціалізації програми: створення порожнього контейнера для подальшого зберігання послідовності об'єктів, виділення інших ресурсів;
- функція виконання програми: вивід на екран меню і виклик фупкції роботи через меню;
- звільнення ресурсів.

4.2. Створення об'єктів та використання контейнерів

Після відлагодження меню користувача слід реалізувати функції створення об'єктів і додавання їх в контейнер.

4.2.1. Класифікація контейнерів

Використання контейнерів під час розробки великих програмних систем, які повинні мати можливість швидко змінюватися в залежності від потреб бізнесу, прискорює швидкість створення таких систем і суттєво полегшує процес її підтримки.

Контейнер задає спосіб організації зберігання даних і являє собою тип, який дозволяє інкапсулювати в собі об'єкти інших типів.

За способом зберігання даних контейнери прийнято розділяти на *послідовні* та *асоціативні*.

Послідовним є такий контейнер, який забезпечує зберігання кінцевої кількості елементів у вигляді безперервної послідовності. Позиція елементів такого контейнера не пов'язана з його значенням, а залежить від часу і місця додавання елемента до контейнера.

Асоціативний контейнер призначений для зберігання пар ключ-значення. Такий контейнер передбачає ефективний доступ до його елементів за значенням ключа на відміну від послідовного контейнера, який більш ефективний у доступі до елементів за їх відносною або абсолютною позицією в контейнері. Асоціативні контейнери можуть бути реалізовані так, що зберігають лише унікальні ключі (якщо контейнер може містити щонайбільше один елемент для кожного значення ключа), в іншому випадку говорять, що контейнер підтримує рівні ключі.

За типом даних, які зберігаються в контейнері, останні поділяють на *однорідні* та *неоднорідні*.

Однорідний контейнер – це такий контейнер, в якому зберігаються об'єкти строго одного типу.

Неоднорідний контейнер – це такий контейнер, в якому можуть зберігатися об'єкти різних типів.

Залежно від принципу побудови контейнери бувають *статичні* та *динамічні*.

Статичний контейнер – це контейнер, склад якого повністю визначається на етапі компіляції. Тобто на етапі компіляції для статичного контейнера визначаються кількість елементів і їх типи, але не самі значення цих елементів.

Динамічний контейнер – це контейнер, склад якого частково або повністю визначається на етапі виконання програми.

Однією з особливостей контейнерів є те, що будь-який вид контейнера може бути агрегацією інших видів контейнерів. Наприклад, статичний контейнер може складатися з динамічних і, відповідно, навпаки.

Найбільш популярними контейнерами є:

- зв'язний список (linked list);
- стек (stack);
- черга (queue);
- черга з пріоритетом (priority queue);
- дек (deque);
- динамічний масив (dynamic array);
- дерево (tree).

Кожен вид контейнера забезпечує свій набір операцій над даними, які зберігаються в ньому. Вибір виду контейнера залежить від того, що потрібно робити з даними в програмі. Наприклад, за необхідності часто додавати та видаляти елементи в середині послідовності елементів контейнера слід використовувати списки, а якщо додавання елементів виконується головним чином в кінець або початок контейнера, – дек.

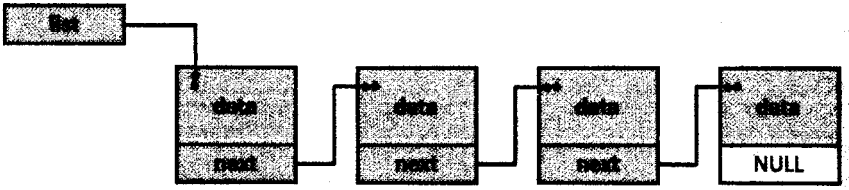
Для розробки контейнерів, принаймні більшої частини з них, мовою C++ необхідні конструкції з використанням масивів, записів (struct), об'єднань (union), посилань або покажчиків. Наприклад, двозв'язний список може бути побудований з використанням записів і покажчиків: кожен запис (вузол списку) зберігає дані і покажчики на «лівий» і «правий» вузли.

4.2.2. Зв'язний список

Однозв'язний список – це такий контейнер, в якому кожен його окремий елемент (вузол) в явному вигляді зберігає інформацію про розміщення свого сусіднього елемента.

Таким чином в C ++ кожен елемент однозв'язного списку крім того, що містить самі дані, також зберігає покажчик на наступний або попередній елемент (найчастіше – на наступний).

Графічне зображення однозв'язного списку:

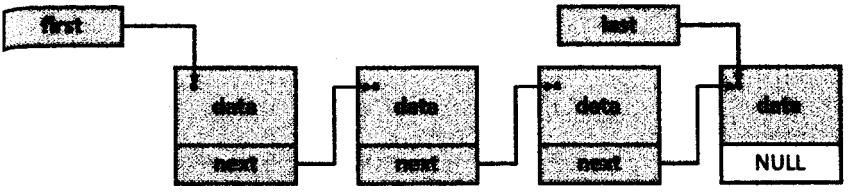


На наведеному зображенні кожен з блоків – це окремий елемент (вузол) списку, на наявність даних в якому вказує data, а покажчик next зберігає інформацію про розміщення сусіднього елемента. На наведеному зображенні в кожному вузлі списку покажчик вказує на наступний вузол, таким чином у зображеному однозв'язному списку можна пересуватися тільки в сторону його кінця, оскільки дізнатися адресу попереднього елемента, спираючись на вміст поточного вузла, неможливо. Останній елемент списку не вказує на жоден інший елемент, про що говорить значення покажчика NULL. Покажчик list є заголовним елементом, який вказує на початок списку, він не містить дані, а тільки посилання на перший елемент.

Робота з останнім елементом однозв'язного списку може бути реалізована по-різному.

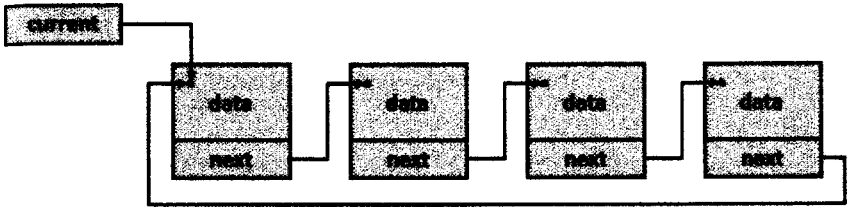
Найпростіший спосіб, який був визначений вище, полягає у встановленні значення покажчика на сусідній елемент в останньому елементі списку рівним NULL. Це означає, що наступний сусідній елемент списку відсутній.

Другий спосіб полягає у визначенні спеціального елемента, що є кінцевим вузлом списку. Цей елемент є покажчиком, який вказує на останній вузол списку. В такий чин для роботи зі списком використовують два спеціальні елементи: покажчик first, який вказує на перший елемент списку, і покажчик last, який вказує на останній елемент списку:



Зазначені два способи реалізації визначають так званий однозв'язний лінійний список.

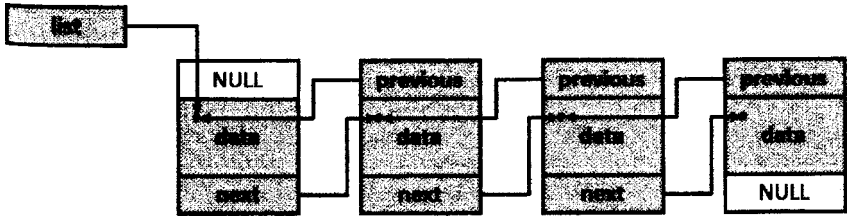
Ще один спосіб полягає у встановленні значення покажчика на сусідній елемент в останньому елементі списку таким, щоб воно вказувало на перший елемент списку:



У цьому випадку реалізація визначає так званий кільцевий (круговий) однозв'язний список. Поняття першого та останнього елемента контейнера для такої реалізації не має сенсу. Щоб спростити повний прохід за всіма елементами такого списку, використовують спеціальний елемент: покажчик current, що вказує на поточний елемент в якомусь зафіксованому стані списку.

Однозв'язний список – це не найзручніший тип зв'язного списку, оскільки з одного елемента можна потрапити лише в один із сусідніх, а отже, рух відбувається лише в один бік. Коли крім покажчика на наступний елемент є ще й покажчик на попередній, то такий список називається двозв'язним.

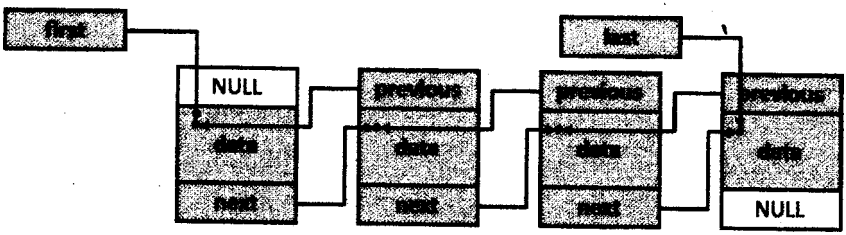
Графічне зображення двозв'язного списку:



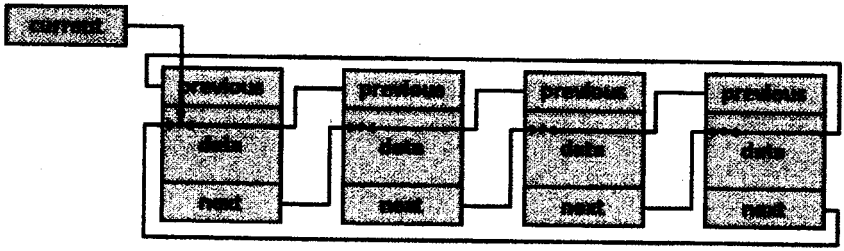
На наведеному зображенні так само, як і для однозв'язного списку, кожен з блоків – це окремий елемент списку, на наявність даних в яких вказує data. Покажчик next зберігає інформацію про розміщення наступного сусіднього елемента, а покажчик previous – інформацію про розміщення попереднього

сусіднього елемента. Покажчик next останнього елемента так само, як і покажчик previous першого елемента зображеного двозв'язного списку, не вказують на інші елементи. Покажчик list є заголовним елементом, який вказує на початок списку, він не містить дані, а тільки посилання на перший елемент.

За аналогією до однозв'язного, в двозв'язному списку робота з покажчиками на останній і на перший елементи може бути реалізована по-різному. У найпростішому випадку значення покажчиків для крайових елементів встановлюються рівними NULL, як було зображено на попередньому рисунку. Ця реалізація визначає так званий двозв'язний лінійний список так само, як і наступна, з визначенням двох спеціальних покажчиків, що вказують на початковий first і кінцевий last вузли списку:



Схожою на реалізацію для випадку однозв'язного списку є й реалізація кільцевого двозв'язного списку. У цьому випадку значення покажчика next останнього елемента зберігає адресу першого елемента, а значення покажчика previous першого елемента зберігає адресу останнього елемента списку:



Зв'язні списки дозволяють ефективно виконувати операції додавання і видалення елементів для будь-якої позиції в послідовності елементів.

Визначення зв'язного списку відрізняється від визначення масиву, для якого наступний елемент знаходиться в пам'яті поряд з попереднім елементом. Під час реалізації масиву необхідно заздалегідь знати, скільки елементів буде в ньому зберігатися. Це необхідно для того, щоб статично виділити безперервну ділянку пам'яті під масив або розробити деяку схему його розширення (або скорочення) для розміщення більшої (або меншої) кількості елементів. У зв'язному списку кожен вузол є окремим елементом, тому для простої його реалізації розподіл пам'яті під кожен вузол виконується окремо. Під час

додавання в список нового елемента під вузол виділяється пам'ять, а потім адреса цієї ділянки пам'яті заноситься до списку. Під час видалення вузла зі списку видаляється значення адреси ділянки пам'яті, де зберігається вузол, після чого звільняється займана ним пам'ять. Таким чином, у зв'язному списку елементи можуть бути розкидані по різних ділянках пам'яті, а отже, розмір зв'язного списку можна не встановлювати заздалегідь.

Розподіл пам'яті під елементи масиву являє собою операцію класу O(1), оскільки всі елементи знаходяться в одному безперервному блоці пам'яті. Для зв'язного списку пам'ять під вузли розподіляється окремо, отже, це операція класу O(n). Навіть якщо не враховувати швидкодію, подібна поведінка може призвести до небажаної фрагментації пам'яті.

Через те, що кожен елемент повинен зберігати покажчик на сусідній елемент у випадку однозв'язного списку, реальний розмір вузла такого контейнера збільшується на розмір покажчика (для 32-х розрядної операційної системи це 4 байта), а у випадку двозв'язного списку – на розмір двох покажчиків.

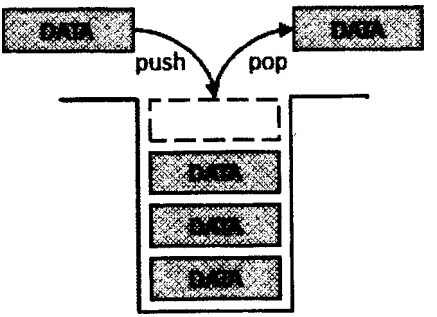
У масиві доступ до n-ого елемента вимагає проведення простих арифметичних обчислень, оскільки всі елементи містяться в одному безперервному блоці пам'яті. А в списку отримання доступу до n-ого елемента досягається лише через перебір всіх елементів, починаючи від крайового елемента і доки не буде досягнений n-ий.

4.2.3. Стек

Стек – контейнер, доступ до елементів якого організований за принципом LIFO (від англ. Last-In-First-Out – «останнім прийшов – першим вийшов»). Принцип LIFO застосовується в тих випадках, коли останні додані до контейнера дані повинні бути першими видалені (оброблені).

У відповідність до принципу LIFO, для стека повинна бути визначена точка взаємодії з ним (так звана вершина стека), через яку буде здійснюватися додавання та видалення елементів.

Графічно зобразити стек зручно у вигляді вертикального списку, операції над елементами якого відбуваються строго з одного кінця контейнера:



В залежності від вимог до програмного забезпечення, що розробляється, реалізація стека може бути виконана по-різному. В більшості випадків стек базується або на звичайному масиві, або на однозв'язному чи двозв'язному списку.

У випадку, якщо максимальний розмір стека відомий заздалегідь, можна реалізувати стек за допомогою масиву. У такій реалізації стек складається з масиву елементів і покажчика на його вершину. Операції додавання та видалення в такому випадку можуть бути реалізовані більш ефективно, ніж з використанням зв'язних списків. Під час додавання елемента до стека спочатку покажчик на вершину стека збільшується на одне значення, а потім робиться запис елемента на місце, яке визначається новим значенням покажчика. Операція видалення елемента полягає в зменшенні покажчика вершини на одне значення.

Реалізація стека на масивах є раціональною з огляду на можливість економії пам'яті, що використовується в списках під зберігання покажчиків для зв'язування елементів списку один з одним. Недоліком є необхідність резервування максимально можливого розміру стека на стадії компіляції програми. При цьому не виключається можливість переповнення стека або навпаки – неекономного витрачання пам'яті, оскільки в процесі роботи програми стек може заповнюватися лише частково.

Стек є надзвичайно зручним контейнером для багатьох задач програмування. З його допомогою організовані виконання рекурсії, управління динамічною пам'яттю, транслятори арифметичних виразів, семантичні та синтаксичні аналізатори тощо.

4.2.4. Черга

Черга – контейнер, доступ до елементів якого організований за принципом FIFO (від англ. First-In-First-Out – «першим прийшов – першим вийшов»), для якого характерно, що всі елементи a_1, a_2, ..., a_{n-1}, додані до контейнера раніше за елемент a_n, повинні бути видалені перш, ніж буде видалено елемент a_n.

У відповідність до принципу FIFO, для черги повинні бути визначені дві точки взаємодії з нею: початок черги, звідки відбувається видалення елементів, і кінець черги, куди відбувається додавання елементів.

Графічне зображення черги:

В залежності від вимог до програмного забезпечення, що розробляється, реалізація черги може бути виконана по-різному. Так само, як і у випадку зі

стеком, в більшості випадків реалізація черги базується або на звичайному масиві, або на однозв'язному чи двозв'язному списку.

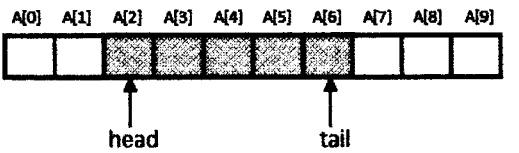
Черга може бути реалізована на базі звичайного масиву. У такій реалізації черга складається з масиву елементів і двох покажчиків, які інтерпретуються як індекси масиву і визначають голову та хвіст черги. У початковому стані черга порожня і обидва покажчики мають нульове значення.

Під час зміни вмісту черги за рахунок вилучення або додавання елементів змінюється значення покажчиків на її голову або хвіст.

Під час додавання елемента до черги значення покажчика на її хвіст змінюється так, щоб вказувати на наступну вільну комірку в черзі. Після цього нові дані записуються в елемент черги, адреса якого визначається новим значенням покажчика на хвіст черги. Вилучення існуючих даних відбувається з елемента, адреса якого визначається значенням покажчика на голову черги. Після цього покажчик на голову змінюється таким чином, щоб вказувати на наступний зайнятий елемент в контейнері.

У найпростішому випадку, коли максимальний розмір черги відомий заздалегідь, значення довжини масиву апріорі встановлює граничний розмір черги, що задається під час виділення пам'яті під масив. Для забезпечення коректної роботи черги в такому випадку необхідне виконання умови, що значення покажчика на голову черги не має перевищувати значення покажчика на хвіст черги. Така реалізація черги називається чергою з лінійним фіксованим буфером.

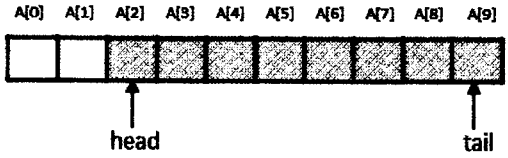
Наприклад, розглянемо логічну структуру черги з лінійним фіксованим буфером, яка реалізована на базі масиву A:



Наведена на зображенні черга має граничний розмір в 10 елементів, під які виділено масив A відповідної довжини. У деякий момент часу, якому відповідає стан черги на рисунку, в контейнері розміщено 5 елементів. Очевидно, що для досягнення такого стану до черги було додано 7 елементів і вилучено 2 з них.

За означенням реалізація черги з лінійним фіксованим буфером має обмежене застосування, оскільки не дозволяє додавати нові елементи, коли хвіст черги вичерпаний, навіть якщо на початку буфера черги є вільні комірки після вилучення головних елементів, що можуть бути використані для розміщення нових елементів.

Цей тупиковий стан переповнення черги з лінійним фіксованим буфером показаний на наступному рисунку, де 8 елементів займають кінцеві елементи буфера черги, а на його початку є 2 вільні комірки:



Спроба додати ще один елемент до такої черги призведе до її переповнення.

Щоб мати можливість використовувати вільні комірки на початку буфера черги, які утворилися після вилучення головних елементів, можна скористатися реалізацією черги, що відома як черга з кільцевим фіксованим буфером. Кільцевий фіксований буфер слід розглядати як замкнутий лінійний буфер, який після заповнення його до кінця може заповнюватися спочатку, якщо там є вільні комірки.

Жорстка умова коректності роботи черги з лінійним фіксованим буфером у черзі з кільцевим буфером стає необов'язковою, тобто стан, коли покажчик на хвіст випереджає покажчик на голову черги, дозволяється.

Використовувати реалізацію черги з кільцевим фіксованим буфером рекомендується тоді, коли потрібно добитися високої продуктивності і економного витрачання пам'яті, а також у випадках, коли заздалегідь відомо, що операції додавання і видалення елементів зустрічаються рівноймовірно.

Як і для стека, для реалізації черги найкращим виявляється підхід, який базується на ідеї використання зв'язних списків.

Черга в програмуванні використовується, як і в реальному житті, коли потрібно зробити якісь дії в порядку їх надходження, виконавши їх послідовно. Зокрема, черга використовується в таких задачах, як послідовність обробки подій в операційних системах, буферизація завдань принтера в системі вхідних і вихідних потоків, організація буфера введення з клавіатури, моделювання процесів обслуговування клієнтів тощо.

4.2.5. Черга з пріоритетом

Черга з пріоритетом працює як звичайна черга, однак внутрішній порядок елементів залежить від їх пріоритету. Той елемент, чий пріоритет вище, додається в початок черги, а чий пріоритет нижче – в її кінець. Таким чином, коли додається новий елемент, він може відразу стати найпершим в черзі на видалення (обробку).

Класичний спосіб реалізації черги з пріоритетом полягає у використанні бінарної (двійкової) купи.

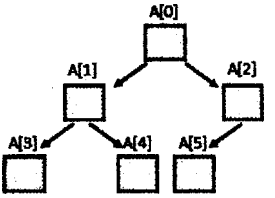
Двійкова купа є масивом, який можна розглядати як майже повне двійкове дерево. У такому дереві будь-який батько не менше, ніж кожен з його нащадків. При цьому на всіх рівнях, крім, можливо, останнього, дерево повністю заповнене (заповнений рівень – це такий рівень, що містить максимально можливу кількість вузлів).

Таким чином, двійкова купа – це двійкове дерево, для якого виконуються умови:

- значення будь-якої вершини не менше, ніж значення її нащадків;
- глибина листя (відстань до кореня) відрізняється не більше, ніж на один рівень;
- останній рівень заповнюється зліва направо.

Для побудови двійкової купи початковий масив ділиться навпіл, при цьому друга його половина вже приймається за правильно побудовану двійкову купу. Потім послідовно беруться елементи з першої половини і додаються до двійкової купи на потрібні місця. Дійсно, для другої половини початкового масиву основна властивість двійкової купи виконується автоматично. Вірніше буде сказати, що ця властивість не порушується, оскільки для елементів другої половини просто не існує нащадків. Отже, немає сенсу для елементів другої половини будувати двійкову купу, послідовно додаючи кожен елемент, оскільки в алгоритмі просто не буде з чим порівнювати поточний елемент.

Найзручніше помістити двійкову купу в масив. При цьому розподіл індексів масиву по вузлах дерева буде виглядати наступним чином:



На етапі додавання елементів в першу половину двійкової купи, щоб виконати додавання елемента, потрібно поміняти його з найбільшим із нащадків, якщо останній перевершує значення елемента. Потім те ж саме необхідно виконати по відношенню вже до нових його нащадків.

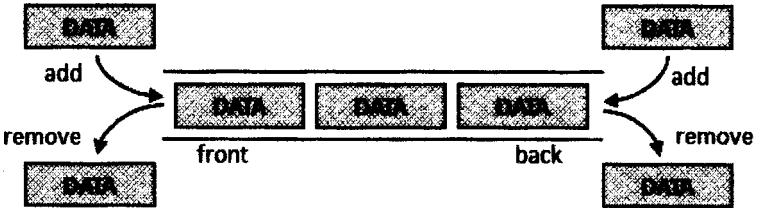
Тобто для того, щоб кожен вузол двійкової купи був більше своїх нащадків, кожен елемент масиву a[i] повинен бути більше або дорівнювати елементам a[2i+1] і a[2i+2]. Таким чином, для побудови першої половини двійкової купи перебирають елементи початкового масиву з N елементів в циклі справа наліво для i = N/2, N/2–1, …, 1. Якщо нерівності $x_i > x_{2i+1}$ та $x_i > x_{2i+2}$ не виконуються, то виконується перестановка x_i з найбільшим з нащадків. Перестановки завершуються при виконанні нерівностей $x_i > x_{2i+1}$ і $x_i > x_{2i+2}$.

Пріоритетні черги використовуються в наступних алгоритмах: алгоритм Дейкстри, алгоритм Прима, дискретно-подієвого моделювання, алгоритм Гаффмана, пошук за першим найкращим збігом, управління смугою пропускання.

4.2.6. Дек

Дек (від англ. double ended queue – «двостороння черга») – контейнер типу «список», що функціонує одночасно за двома принципами організації даних FIFO і LIFO. Тобто цей контейнер являє собою послідовність елементів, в яку елементи можна додавати і видаляти в довільному порядку з двох сторін. Перший і останній елементи дека відповідають його входу і виходу.

Графічне зображення дека:



Розрізняють обмежені різновиди дека: дек з обмеженим входом і дек з обмеженим виходом. Перший допускає додавання елементів тільки в один з кінців дека, а другий – видалення елементів тільки з одного з кінців дека.

Дек може бути реалізований як статичний контейнер на базі масиву і як динамічний контейнер на базі зв'язного списку. Найбільш тривіальним визначенням дека є визначення через двозв'язний список, в який заборонено додавання елементів в середину контейнера. Оскільки для дека, як і для черги, передбачена робота з обома кінцями контейнера, то доцільно використовувати для реалізації дека ті самі підходи, що застосовувалися і для черги.

Незважаючи на підхід до реалізації дека, в певний момент часу виконання поелементних операцій над цим контейнером передбачає можливість визначення однієї з його сторін як активної.

Дек є найбільш гнучким і, як наслідок, найбільш складним контейнером, що надає широкий доступ до формування послідовності, в якій будуть зберігатися необхідні набори даних. З усіх розглянутих раніше лінійних контейнерів саме дек є найбільш універсальним. Накладаючи додаткові обмеження на операції з початком і / або кінцем дека, можна здійснювати моделювання стека і черги.

Задачі, що вимагають використання контейнера дека, зустрічаються в обчислювальній техніці і програмуванні набагато рідше, ніж завдання, які реалізуються за допомогою стека або черги. Зразком використання дека може бути, наприклад, якийсь термінал, в який поступають команди, кожна з яких виконується певний час. Якщо ввести наступну команду, не дочекавшись, доки закінчиться виконання попередньої, то вона встане в чергу і почне виконуватися, як тільки звільниться термінал. Ця частина може бути реалізовано за принципом FIFO як контейнер черга. Якщо ж додатково ввести операцію відміни останньої введеної команди, то це потребує використання контейнера дека.

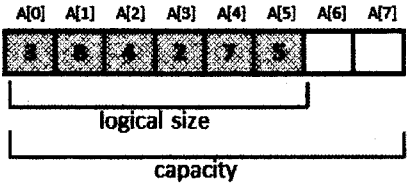
4.2.7. Динамічний масив

Динамічні масиви використовуються для розміщення в пам'яті та обробки послідовності даних, розмір яких неможливо визначити на момент написання програми.

Семантика масиву передбачає його фізичну структуру у вигляді безперервного блоку пам'яті, що розділений на окремі елементи, адреси яких утворюють арифметичну прогресію. Отже, за індексом елемента процесор може дуже швидко визначити відповідну адресу в пам'яті і зробити операцію читання або запису значення. Час виконання такої операції не залежить від розміру масиву, а значить, алгоритмічна складність такої операції становить O(1).

Існує безліч алгоритмів реалізації динамічних масивів. Найбільш поширеною реалізацією є реалізація зміни розміру масиву шляхом переміщення його в динамічній пам'яті.

У цій реалізації під масив виділяється безперервний блок пам'яті, розмір якого більше за необхідний, тобто більше за так званий логічний розмір масиву (англ. size). Кількість елементів, яка фактично може бути розміщена в цій пам'яті, називається ємністю (англ. capacity) динамічного масиву. Таким чином, динамічний масив – це масив фіксованого розміру, в якому зайнята тільки частина його елементів:



Операція збільшення розміру масиву, якщо новий розмір не перевищує ємності, просто змінює лічильник довжини масиву до потрібного розміру. З самим масивом ніяких змін при цьому не відбувається.

Операція збільшення розміру, для якої новий розмір перевищує ємність, призводить до переміщення масиву в пам'яті. Для цього виділяється новий блок пам'яті, розмір якого перевищує поточний розмір масиву, при цьому відповідно оновлюється значення ємності масиву. Далі поточний вміст масиву копіюється у новий виділений блок пам'яті, а значення покажчика на дані масиву змінюється на нове. Останнім кроком виконується звільнення раніше виділеного під масив блоку пам'яті.

Зменшення розміру масиву також може призводити до переміщення його в пам'яті. Це відбувається, якщо в результаті операції зменшення елементів масиву відсоток зайнятої під них пам'яті падає нижче певного значення. Переміщення при цьому проводиться за тією ж схемою, що й у разі збільшення масиву.

Ефективність зазначеної реалізації динамічного масиву буде залежати від параметрів, які визначають умови переміщення масиву в пам'яті. До таких параметрів відносяться величина приросту ємності під час збільшення розміру масиву, мінімальне значення ємності, відсоток мінімального заповнення масиву.

Приріст ємності під час збільшення розміру масиву може задаватися або відносною величиною (певна частка від логічної довжини масиву), або абсолютним приростом понад необхідну довжину масиву. Чим більше цей параметр, тим пізніше при тому самому режимі заповнення масиву потрібно буде виконувати наступне переміщення в пам'яті. Однак зі збільшенням значення цього параметра збільшується вірогідність того, що виділена пам'ять залишиться невикористаною.

Для підвищення ефективності, щоб не виконувати частих переміщень невеликих масивів у пам'яті, задається мінімальне значення ємності. Практично визначення мінімального значення ємності під час реалізації динамічного масиву означає, що всі масиви розміром менші за задану мінімальну ємність в дійсності будуть втрачати зайву пам'ять.

Відсоток мінімального заповнення масиву визначає, коли буде відбуватися переміщення після зменшення розміру масиву. Чим більше це значення, тим частіше при зменшенні розміру масиву буде відбуватися переміщення. Чим воно менше, тим більше пам'яті під час зменшення розміру масиву буде залишатися зайнятою, але не буде при цьому використовуватися.

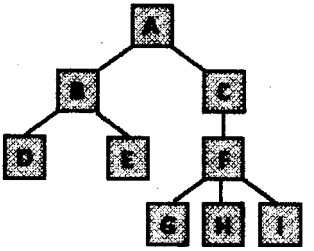
Жоден із способів визначення параметрів, що визначають умови переміщення масиву в пам'яті, не гарантує максимальної ефективності для всіх можливих режимів роботи з масивом. Тому, незважаючи на особливості реалізації контейнера динамічного масиву, для компенсації негативних ефектів доцільною може бути розробка операцій збільшення та зменшення елементів масиву з можливістю прямого встановлення його ємності. У такому випадку, якщо під час використання контейнера динамічного масиву буде точно відомо, до яких розмірів збільшиться або зменшиться масив в результаті тієї чи іншої операції, можна буде прямо вказати необхідну кінцеву ємність в операції зміни його розміру.

4.2.8. Дерево

Дерево – це сукупність елементів, які називаються вузлами (при цьому один з них визначається як корінь), і відносин (батько-нащадок), що утворюють ієрархічну структуру вузлів, які мають відношення «один до багатьох».

Самий верхній вузол дерева називається коренем. Верхній вузол для нижнього вузла називається батьком, а нижній вузол для верхнього – нащадком. Вузли, що не мають нащадків, називаються термінальними вузлами або листям дерева. Нетермінальні вузли називаються внутрішніми вузлами дерева. Два вузли дерева з'єднуються гілкою. Дерево без гілок з одним вузлом – це пусте або нульове дерево.

Графічне зображення дерева:



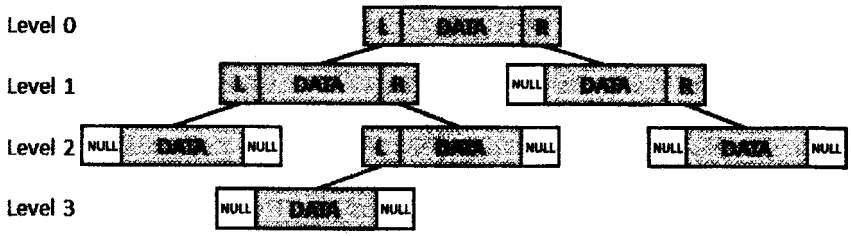
Для дерева, зображеного на рисунку, коренем є вузол A, листям – вузли D, E, G, H, I, внутрішніми вузлами – вузли B, C, F. Якщо окремо розглянути, наприклад, вузол F, то він є нащадком вузла C і водночас виступає батьком по відношенню до вузлів G, H, I.

Важливими метричними характеристиками дерев є ступінь розгалуження і рівень їх вузлів. Ступенем розгалуження вузла дерева є число вузлів, для яких він є батьком. Рівень вузла дерева визначається довжиною шляху, утвореного послідовністю вузлів, які з'єднують його з коренем. Довжина шляху вимірюється числом вузлів в ньому. Таким чином, корінь дерева лежить на нульовому рівні, а рівень будь-якого іншого вузла дерева має рівень на одиницю більше рівня свого батька. Максимальний рівень будь-якої вершини дерева називається її глибиною (від кореня до вузла) або висотою (від вузла до максимально віддаленого листа). Звідси глибина кореня дорівнює нулю, а максимальний рівень всіх з вершин дерева називається глибиною дерева.

Для дерева, зображеного на рисунку раніше, корінь A і вузол B мають ступінь розгалуження 2, ступінь розгалуження вузла C дорівнює 1, а вузла F дорівнює 3. Ступені інших вузлів дорівнюють 0, тому що вони є листям дерева. Що стосується рівня, то корінь A має рівень 0, вузли B і C – рівень 1, вузли D, E, F – рівень 2, а вузли G, H, I – рівень 3.

Узагальненням поняття дерева є поняття лісу. Лісом (або бором) називають впорядковану множину непересічних дерев. Простота перетворення дерева в ліс і навпаки свідчить про близькість цих двох понять. Видалення кореня перетворює дерево в ліс, і навпаки, додавання одного вузла перетворює ліс в дерево, в якому цей вузол стає коренем. Ось чому інколи для позначення лісу з N дерев вживають термін: дерево з N-кратним коренем.

Якщо у кожного вузла дерева є не більше двох нащадків, то таке дерево називається двійковим або бінарним. Графічно двійкове дерево можна зобразити як послідовність елементів, кожен з яких може бути пов'язаний з двома (лівим і правим) наступними елементами:



Кожен елемент (вузол) двійкового дерева має два покажчики: на «лівий» вузол-нащадок L і на «правий» вузол-нащадок R лівої та правої гілок вузла відповідно. Ліва та права гілки кожного вузла бінарного дерева породжують незалежні ліве та праве піддерево відповідно. Для термінальних вузлів, які не мають нащадків, значення покажчиків L та R дорівнюють NULL.

Повним бінарним деревом рівня n називається таке бінарне дерево, в якому кожен вузол рівня n є листом, а кожен вузол рівня, меншого за n, має непусті ліве і праве піддерева.

Для більш ефективної реалізації алгоритмів пошуку використовуються дерева двійкового пошуку. Дерево двійкового пошуку – це таке дерево, в якому значення всіх лівих нащадків менші за значення, що зберігає їх батьківський вузол, а значення всіх правих нащадків – більші. Ця властивість називається характеристичною властивістю дерева двійкового пошуку і виконується для будь-якого вузла, включаючи корінь. Для реалізації бінарного дерева пошуку значення вузлів має бути асоційоване з ключами, для яких визначена операція порівняння вузлів дерева. У конкретних реалізаціях це може бути пара ключ-значення або саме значення вузла може виступати в якості ключа. Враховуючи все вищесказане, пошук вузла в двійковому дереві пошуку можна здійснити, рухаючись від кореня в ліве або праве піддерево залежно від значення ключа піддерева.

Визначення дерева як визначення будь-якого рекурсивного об'єкта містить базисну і рекурсивну частини. Базисна частина, яка визначає корінь дерева, є нерекурсивним твердженням. Рекурсивна частина визначається так, щоб її можна було звести до базисної виконанням послідовності повторних застосувань. Тобто дерево з числом вузлів $n > 1$ рекурсивно визначається через дерева з числом вузлів менше, ніж n, доки не буде досягнуто базисного кроку, на якому дерево складається з єдиного вузла – його кореня. Рекурсивне визначення дерева дозволяє більш простим способом формалізувати його структуру і алгоритми його обробки.

4.2.9. Реалізація контейнера

Для реалізації контейнера в курсовій роботі слід розробити клас контейнера відповідно до свого варіанта. Для організації класу елемента (вузла) контейнера найдоцільніше розробити структуру.

На цьому етапі в класі контейнера слід розробити функції, що необхідні для роботи із заданим контейнером. Незалежно від логіки, що передбачена конкретним контейнером, обов'язковими до реалізації є операції:

- додавання елемента в контейнер (пам'ять під черговий елемент контейнера слід виділяти динамічно);
- друку вмісту контейнера на екран;
- видалення елемента з контейнера;
- видалення контейнера (знищення всіх його елементів).

Першою доцільно реалізувати операцію додавання елементів до контейнера. Приклад реалізації функції меню, в якій організовано створення об'єкта з подальшим його додаванням в контейнер для зберігання:

```
void CreateWorker()
{
      Person* p;

      string cur_worker = "John Daw";
      int cur_age = 23;
      int cur_rang = 5;

      p = new worker( cur_worker, cur_age, cur_rang );
      collection->push_back(p);

}
```

Створивши таку функцію, слід переконатися, що об'єкти дійсно створюються і поміщаються в контейнер. Для цього буде доцільним реалізувати операцію друку вмісту контейнера на екран, тобто виконати перегляд контейнера.

Далі слід реалізувати операцію видалення елемента контейнера відповідно до логіки, за якою організовано контейнер.

Останньою слід реалізувати операцію видалення контейнера, в якій елементи можуть видалятися послідовно через виклик операції видалення окремого елемента контейнера, що була розроблена напередодні.

Таким чином, на цьому етапі виконаними є реалізація пунктів меню 1 – 4 без введення даних з клавіатури.

4.3. Організація роботи з даними через файл

Наступним етапом реалізації програмного продукту є, по-перше, розробка ініціалізації об'єктів через введення даних з клавіатури і, по-друге, розробка функції для збереження об'єктів у файлі і завантаження їх з файлу.

4.3.1. Ініціалізація об'єктів

Під час введення даних з клавіатури необхідно врахувати введення рядків, що містять пробіли. Для введення таких рядків слід використовувати метод getline().

Приклад введення рядків і чисел:

```
void CreateEmployer()
{
      Person* p;
      p = new Employer();
      p->Input();

      collection->push_back(p);
}

void Employer::Input()
{
      cout<<"Input employer`s full name: ";
      getline(cin, name, '\n');

      cout<<"Input employer`s age: ";
      cin>>age;

      cout<<"Input employer`s post";
      cin>>post;
}
```

4.3.2. Серіалізація і десеріалізація об'єктів

Під час збереження об'єктів у файлі виникає декілька проблем.

По-перше, треба не просто записати поля об'єкта у файл, а виконати так звану серіалізацію об'єкта. Під серіалізацією в програмуванні розуміють переведення (у нашому випадку об'єкта) в послідовність бітів (які і записуються у файл). Рішення цієї задачі простим копіюванням даних об'єкта у файл не завжди можливе (наприклад, якщо об'єкт містить покажчики або посилання на інші об'єкти). Для вирішення цієї проблеми необхідно написати свої функції для серіалізації об'єктів, в яких враховуються всі зв'язки об'єкта з іншими об'єктами.

Друга проблема пов'язана з тим, що, оскільки в контейнері зберігаються поліморфні об'єкти різних класів, то і у файл повинні записуватися об'єкти різних класів. Отже, файл повинен бути поліморфним. Основна складність при роботі з таким файлом пов'язана з тим, що об'єкти різних класів мають різну структуру. Необхідно мати механізм для виконання різних функцій серіалізації залежно від типу об'єкта. Одним із способів рішення цієї задачі є запис перед полями об'єкта ідентифікатора, що визначає тип об'єкта (наприклад, це може бути ціле число).

Реалізація методів запису даних об'єкта у файл і зчитування даних з файлу до об'єкта може бути виконана за наступною схемою, при цьому необхідно перенавантажити для класів ієрархії операції запису в потік ostream << і зчитування з потоку istream >>:

```
//Серіалізація
void WritePersons(/*параметром є вказівник ptr на контейнер*/)
{
      //Спочатку необхідно перевірити, чи не порожій контейнер,
      //якщо контейнер порожній,
      //то вивести про це повідомлення і вийти з функції
```

```
      //Далі необхідно створити потік
      //і відкрити файл для запису
      ofstream out("persons.dat");

      //Далі необхідно перевірити стан потоку на можливість роботи з ним
      if (!out.is_open())
      {
            cout << "Error opening file!" << endl;
            return;
      }

      //Якщо все гаразд, то далі необхадно пройти по вмісту контейнера і
      //для кожного об'єкта виконати наступні дії:
      if(typeid(*(ptr->el))==typeid(Worker))//(el-вказівник на поточний об'єкт)
      {
            out<<1<<' '; //запис у файл 1
            out<<(*((Worker*)(ptr->el))); //серіалізація об'єкта Worker
      }
      if(typeid(*(ptr->el))==typeid(Employee))
      {
            out<<2<<' '; //запис у файл 2
            out<<(*((Employee*)(ptr->el))); //серіалізація об'єкта Eemployer
      }
}
```

```
//Десеріалізація
int ReadPersons(/*параметром є вказівник ptr на контейнер*/)
{
      //Спочатку необхідно створити потік
      //і відкрити файл для читання
      ifstream in;
      in.open("persons.dat");

      //Далі необхідно перевірити стан потоку на можливість роботи з ним
      if (!in.is_open())
      {
            cout << "Error opening file!" << endl;
            return;
      }

      //Далі необхідно створити тимчасові змінні для організації роботи
      Worker* w;
      Employer* e;
      int k, count=0;

      //Далі, поки не кінець файлу, виконувати:
      while(!in.eof())
      {
            //Зчитуємо ознаку типу об'єкта
            in>>k;

            if(k==1) //Це об'єкт Worker
            {
                  w=newWorker();
                  in>>(*w); //Зчитуємо його та розміщуємо в контейнері
                  ptr->add(w);
                  count++;
            }

            if(k==2)//Це об'єкт Employer
```

```
            {
                  e=new Employer();
                  in>>(*emp); //Зчитуємо його та розміщуємо в контейнері
                  ptr->add(e);
                  count++;
            }

            //Далі необхідно перевірити випадок хибної ознаки типу об'єкта

      }

      return count; //функція повертає кількість створених об'єктів
}
```

4.4. Пошук даних у контейнері

На наступному етапі розробки програмного продукту слід відповідно до свого варіанта розробити функцію для пошуку певного елемента в контейнері. Запит на пошук необхідно організувати в діалоговому режимі з користувачем.

4.4.1. Алгоритми пошуку в контейнерах

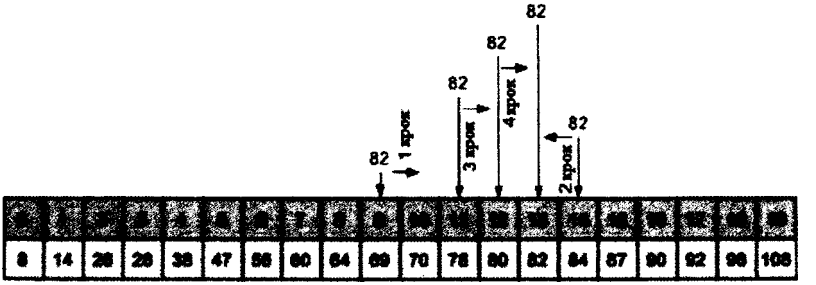
Лінійний пошук (англ. linear search) є найпростішим з відомих. Його суть полягає в послідовному перегляді елементів в деякому порядку, який гарантує перегляд всієї множини елементів (наприклад, перегляд черги від її головного елемента до останнього елемента в її хвості). Якщо в ході перегляду елементів буде знайдений шуканий елемент, то алгоритм пошуку припиняється з позитивним результатом; якщо ж після перегляду всієї множини шуканий елемент не буде знайдений, то алгоритм пошуку має видати негативний результат.

Найпростіший спосіб прискорити алгоритм пошуку полягає в модифікації алгоритму лінійного пошуку за рахунок усунення операції перевірки на закінчення елементів контейнера. Ця перевірка здійснюється кожен раз під час звернення до чергового елемента. Щоб позбутися виконання перевірки на кожному кроці алгоритму пошуку в кінець контейнера додається ще один елемент, значення якого дорівнює шуканому. В такому випадку перевірка на закінчення контейнера здійснюється лише за умови збігу чергового елемента з шуканим. Якщо шуканий елемент знаходиться всередині контейнера, то пошук закінчується вдало і елемент вважається знайденим. Якщо шуканий елемент виявився останнім, то пошук закінчується невдачею, тобто шуканого елемента в контейнері немає.

Бінарний (дихотомічний) пошук (англ. binary search) передбачає, що множина елементів зберігається, як деяка відсортована (наприклад, за зростанням) послідовність елементів, до яких можна отримати прямий доступ за допомогою індексу. Фактично мова йде про те, що множина елементів зберігається в масиві, який відсортований. Суть методу полягає в тому, що наявність шуканого елемента перевіряється через його порівняння з елементом, який розташований посередині області пошуку. Областю пошуку називається

частина масиву, в якій імовірно знаходиться шуканий елемент. Спочатку областю пошуку є всі елементи масиву від першого до останнього. У зазначеній області визначається елемент, який розташований посередині. Використання масиву в зазначеному методі забезпечує миттєвий доступ до елемента за його індексом. Далі виконується порівняння значення визначеного посередині області елемента з шуканим значенням. Залежно від того, більше воно чи менше шуканого значення, подальший пошук буде відбуватися в правій або лівій половинах масиву. Якщо значення виявиться рівним шуканому, то пошук завершено. Як можна бачити, область пошуку на кожному кроці алгоритму буде скорочуватися вдвічі.

Схема бінарного пошуку елемента, значення якого дорівнює 82, у відсортованому за зростанням масиві із 20 елементів:



Як видно з рисунка, на кожному кроці алгоритму пошуку значення знайденого середнього в області пошуку елемента порівнюється з 82. В залежності від результату порівняння, пошук продовжується в ділянці масиву, що відповідає лівій або правій половині поточної області пошуку.

Виходячи з опису алгоритму бінарного пошуку, можна прийти до висновку, що малоймовірно, щоб він використовувався для зв'язних списків. Тим не менш, реалізація бінарного пошуку для зв'язних списків може бути виправданою. У загальному випадку перехід за посиланням виконується набагато швидше, ніж виклик функції порівняння. Це означає, що мінімізація викликів функції порівняння може привести до прискорення пошуку в контейнері. Очевидно, що для реалізації алгоритму необхідно знати заздалегідь кількість елементів, що містяться в контейнері.

Виконуючи операцію пошуку в дереві, доводиться досліджувати елементи, що зберігаються ієрархічно. Спосіб дослідження вузлів дерева, при якому кожен вузол відвідується тільки один раз, називається *обходом дерева* (англ. tree traversal). При цьому в результаті обходу дерева виходить лінійна розстановка його вузлів.

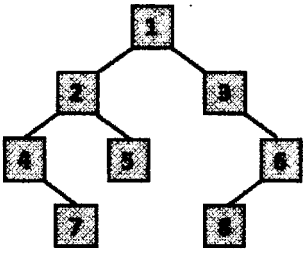
Описані нижче алгоритми обходу відносяться до двійкових дерев, але вони можуть бути узагальнені і для інших дерев.

Залежно від траєкторій виділяють два типи обходу:

– горизонтальний (в ширину);
– вертикальний (в глибину).

Під горизонтальним мається на увазі обхід дерева за рівнями (level-ordered). Під час такого обходу спочатку відвідуються всі вузли поточного рівня, після чого здійснюється перехід на нижній рівень.

Графічне зображення послідовності відвідування вершин бінарного дерева під час горизонтального обходу:



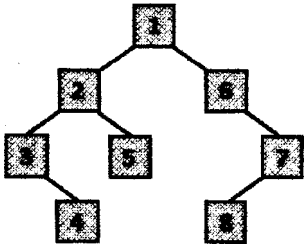
Обхід дерева в ширину може бути реалізований за допомогою черги. Спочатку в порожню чергу додається корінь дерева, далі, поки черга не буде порожньою, із черги виштовхується черговий вузол і обробляється, після чого в чергу додаються нащадки цього вузла.

Під час вертикального обходу порядок обробки поточного вузла і вузлів його лівого та правого піддерев можуть бути різними. За цією ознакою виділяють три варіанти вертикального обходу, кожний з яких визначається рекурсивно:

– прямий порядок (префіксний, pre-ordered);
– зворотний порядок (постфіксний, post-ordered);
– симетричний порядок (інфіксний, in-ordered).

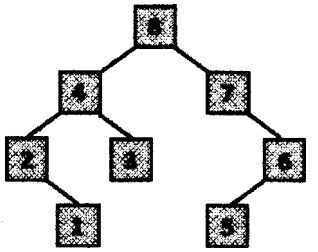
Прямий порядок передбачає виконання наступної послідовності кроків: відвідати корінь → відвідати ліве піддерево → відвідати праве піддерево. Тобто в такому порядку обходу кожна вершина відвідується до того, як будуть відвідані її діти.

Графічне зображення послідовності відвідування вершин бінарного дерева під час прямого обходу:



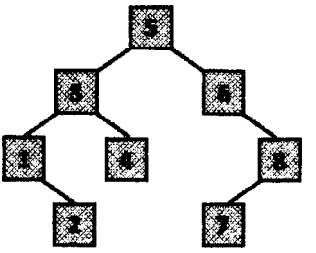
Зворотний порядок передбачає виконання наступної послідовності кроків: відвідати ліве піддерево → відвідати праве піддерево → відвідати корінь. Тобто в такому порядку кожна вершина відвідується лише після того, як будуть відвідані її діти.

Графічне зображення послідовності відвідування вершин бінарного дерева під час зворотного обходу:



Симетричний порядок передбачає виконання наступної послідовності кроків: відвідати ліве піддерево → відвідати корінь → відвідати праве піддерево. У такому порядку кожна вершина відвідується між відвіданням лівої та правої дитини. Такий порядок особливо часто застосовується в двійкових деревах пошуку, тому що дає можливість обходу вершин у порядку збільшення їхніх порядкових номерів.

Графічне зображення послідовності відвідування вершин бінарного дерева під час симетричного обходу:



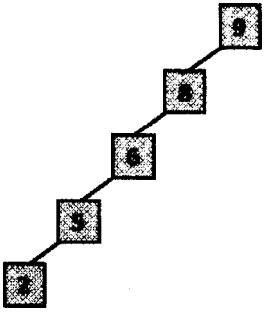
Як вже було зазначено, бінарні дерева пошуку призначені для швидкого доступу до даних. Для того, щоб в бінарному дереві пошуку в середньому було потрібно виконати найменшу кількість операцій для здійснення пошуку елемента в ньому, дерево має бути збалансованим, тобто таким, в якому висота лівого і правого піддерев відрізняються не більше, ніж на одиницю.

Збалансоване бінарне дерево пошуку буде мати мінімальну висоту в порівнянні з іншими деревами з такою ж кількістю вузлів, при цьому операція пошуку в такому дереві матиме складність порядку $O(\log_2 n)$.

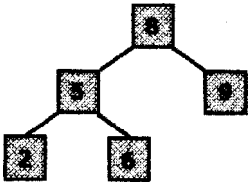
Під час додавання або видалення вузлів збалансоване дерево може виродитися і доступ до даних істотно сповільниться, в найгіршому випадку операція пошуку матиме складність O(n).

Для підтримки дерева в збалансованому стані під час додавання або видалення вузлів необхідно виконувати перебудову дерева (операцію балансування).

Екстремально незбалансоване бінарне дерево пошуку:



Те саме дерево, що й на попередньому рисунку, але після виконання операції балансування:



В ідеалі операція балансування має призводити до випадку, коли операція пошук в дереві має логарифмічну складність. Однак, оскільки операція балансування вимагає додаткового часу на її виконання, під час частого додавання або видалення вузлів підтримувати дерево в збалансованому стані стає не вигідним через великі затрати часу на балансування. Виходом з такої ситуації є менш строге визначення збалансованості дерева (висота лівого і правого піддерев може відрізнятися не на одиницю, а на якусь більшу за значенням константну величину).

4.4.2. Реалізація пошуку певного елемента в контейнері

У курсовій роботі під час розробки функції для пошуку певного елемента в контейнері необхідно реалізувати алгоритм пошуку.

Для прискорення виконання пошуку необхідно попередньо відсортувати елементи контейнера будь-яким з методів сортування. У випадку розробки

операції пошуку в двійковому дереві пошуку дерево попередньо необхідно перебудувати, виконавши операцію балансування.

Сортування елементів контейнера слід виконувати за ознакою, з якою пов'язаний запит. Виходячи із запиту, також слід обрати і ознаку для визначення ключа вузлів бінарного дерева пошуку.

Сортування контейнера, так само, як і балансування бінарного дерева, мають бути реалізовані як метод розробленого класу контейнера.

Контрольні запитання до розділу

1. У якому випадку мову програмування можна назвати об'єктно-орієнтованою?
2. За якими характерними ознаками мову C++ можна віднести до об'єктно-орієнтованої мови програмування?
3. Що таке інтегровані середовища розробки, з якою метою вони створені?
4. Що зазвичай входить до складу інтегрованих середовищ розробки?
5. Яку функціональність може мати редактор коду інтегрованого середовища розробки?
6. Як працювати з відлагоджувачем коду в інтегрованому середовищі розробки Microsoft Visual Studio?
7. Для чого призначений транслятор інтегрованого середовища розробки?
8. Для чого призначений компонувальник інтегрованого середовища розробки?
9. Навіщо використовують системи управління версіями, які з них найбільш поширені під час розробки програмного забезпечення?
10. У який чин може бути реалізований поліморфізм, використовуючи мову програмування C++?
11. Навіщо під час розробки програм мовою C++ визначення класів розміщують у файлах *.h, а визначення методів класу – у файлах *.cpp?
12. Які дві форми визначення імені файлу для підключення має директива препроцесора *#include*?
13. Коли доцільно виконувати ініціалізацію полів класу з використанням синтаксису через списки ініціалізації конструкторів?
14. Чому методи класів, що не змінюють стан об'єкта, для якого вони викликаються, слід оголошувати як const-методи?
15. Яким чином спадкування дозволяє покращити повторне використання коду в класах-нащадках під час реалізації коду мовою програмування C++?
16. Як реалізувати мовою програмування C++ клас, який буде мати лише один екземпляр в програмі?
17. Чому використання контейнерів під час розробки великих програмних систем прискорює швидкість їх створення і суттєво полегшує процес її підтримки?

18. Що задає контейнер?
19. Яким чином розділяють контейнери за способом зберігання даних?
20. Що таке послідовний контейнер?
21. Для чого призначені асоціативні контейнери?
22. Яким чином розділяють контейнери за типом даних, які зберігаються в ньому?
23. Що таке однорідний контейнер?
24. Як реалізувати мовою C++ неоднорідний контейнер?
25. Яким чином розділяють контейнери залежно від принципу їх побудови?
26. Що таке статичний контейнер?
27. Що таке динамічний контейнер?
28. Що значить, що контейнер є агрегацією інших видів контейнерів?
29. Перелічити найбільш популярні контейнери.
30. Що таке однозв'язний список?
31. Як реалізувати однозв'язний список мовою програмування C++?
32. Якими різними способами може бути реалізована робота з останнім елементом однозв'язного списку?
33. Чим лінійний однозв'язний список відрізняється від кільцевого (кругового) однозв'язного списку?
34. Що таке двозв'язний список?
35. Як реалізувати двозв'язний список мовою програмування C++?
36. Чим лінійний двозв'язний список відрізняється від кільцевого (кругового) двозв'язного списку?
37. Чим визначення зв'язного списку відрізняється від визначення масиву?
38. На скільки збільшується реальний розмір вузла зв'язного списку?
39. Як у зв'язному списку отримати доступ до n-ого елемента?
40. Що таке контейнер стек?
41. У чому полягає принцип організації даних LIFO?
42. Що таке вершина стека?
43. Як реалізувати стек мовою програмування C++?
44. Чому реалізація стека на масивах вважається раціональною?
45. Що таке контейнер черга?
46. У чому полягає принцип організації даних FIFO?
47. Як реалізувати чергу мовою програмування C++?
48. Що таке реалізація черги з лінійним фіксованим буфером?
49. Що таке реалізація черги з кільцевим фіксованим буфером?
50. Який підхід до реалізацій стека та черги вважається найкращим?
51. Що таке контейнер черга з пріоритетом?
52. Як реалізувати чергу з пріоритетом мовою програмування C++?
53. Що таке бінарна (двійкова) купа?
54. Як відбувається побудова двійкової купи?
55. У яких алгоритмах використовується черга з пріоритетом?
56. Що таке контейнер дек?
57. Як реалізувати дек мовою програмування C++?

58. Які розрізняють обмежені різновиди дека?
59. Яка сторона дека називається активною?
60. Які додаткові обмеження на операції з початком і / або кінцем дека необхідно виконати для здійснення моделювання на його основі стека і черги?
61. Що таке контейнер динамічний масив?
62. Яку фізичну структуру передбачає семантика масиву?
63. Як реалізувати динамічний масив мовою програмування C++?
64. Яка реалізація динамічного масиву є найбільш поширеною, в чому полягає її суть?
65. Що таке логічний розмір динамічного масиву?
66. Що визначається поняттям ємності динамічного масиву?
67. Чим може бути заданий приріст ємності під час збільшення розміру динамічного масиву?
68. У якому випадку операція додавання елемента до динамічного масиву призводить до переміщення його в пам'яті?
69. У якому випадку операція видалення елемента динамічного масиву призводить до переміщення його в пам'яті?
70. Від чого залежить ефективність реалізації зміни розміру динамічного масиву шляхом переміщення його в динамічній пам'яті?
71. З якою метою задається мінімальне значення ємності динамічного масиву?
72. Чому жодна з реалізацій динамічного масиву не може гарантувати максимальну ефективність для всіх можливих режимів роботи з цим контейнером?
73. Що таке контейнер дерево?
74. Як реалізувати дерево мовою програмування C++?
75. Що називається коренем контейнера дерево?
76. Який вузол дерева називається батьком, а який нащадком?
77. Що таке термінальні вузли (листя) дерева?
78. Що таке внутрішні вузли дерева?
79. Що таке пусте (нульове) дерево?
80. Що визначає ступінь розгалуження і що визначає рівень вузлів дерева?
81. Що таке глибина і що таке висота контейнера дерево?
82. Що означає поняття лісу (бору)?
83. Яке дерево називається двійковим (бінарним)?
84. Яке бінарне дерево називається повним бінарним деревом рівня n?
85. Що таке дерево двійкового пошуку, з якою метою воно використовується?
86. Які особливості визначення дерева як рекурсивного об'єкта?
87. Що розуміють під поняттям серіалізації в програмуванні?
88. Що розуміють під поняттям десеріалізації в програмуванні?
89. Які особливості перенавантаження операцій запису в потік ostream << і зчитування з потоку istream >>?
90. У чому полягає ідея лінійного пошуку даних в контейнері?
91. Який найпростіший спосіб прискорити лінійний пошук?

92. У чому полягає ідея бінарного (дихотомічного) пошуку даних в контейнері?
93. Що називається обходом дерева?
94. Які виділяють типи обходу дерева залежно від траєкторій руху в ньому?
95. Як реалізувати обхід дерева в ширину мовою програмування C++?
96. Які є три варіанти вертикального обходу дерева, в чому їх різниця і особливості реалізації мовою програмування C++?
97. Що таке збалансоване, незбалансоване, екстремально незбалансоване бінарне дерево пошуку?
98. До чого в ідеальному випадку має призводити операція балансування бінарного дерева пошуку?
99. Які існують алгоритми балансування бінарного дерева пошуку?

РОЗДІЛ 5
ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Перед початком тестування програмного продукту, що розробляється в курсовій роботі, необхідно розробити тест-план, який буде містити набір тестових сценаріїв для повного покриття вимог відповідно до визначеної специфікації.

З метою формалізації сутностей, на яких буде проводитися подальше тестування, в курсовій роботі слід розробити діаграму об'єктів із зазначенням їх стану в програмному продукті під час проведення тестування.

Тестування програмного продукту, що розробляється в курсовій роботі, має складатися з проведення:

- модульного тестування роботи контейнера;
- системного тестування.

Опис тестових сценаріїв та діаграми об'єктів необхідно виконувати, дотримуючись правил, які викладені в цьому розділі.

Тестування програмного забезпечення – це вид діяльності, пов'язаний з виконанням процесів, спрямованих на виявлення (доказ наявності) помилок (невідповідностей, неповноти, двозначностей тощо) в системі, яка визначається поточним станом програмного забезпечення.

Процес тестування відноситься, в першу чергу, до перевірки коректності програмної реалізації системи, тобто визначення відповідності реалізації системи до вимог. Таким чином, тестування – це кероване виконання програми з метою виявлення невідповідностей між її поведінкою та вимогами.

5.1. Методологія тестування програмного забезпечення

Виконання тестування програмного забезпечення спрямоване на виявлення існуючих в програмному коді дефектів. Під дефектом розуміють ділянку програмного коду, виконання якої за певних умов призводить до несподіваної поведінки системи (тобто такої поведінки, що не відповідає вимогам до системи).

Несподівана поведінка системи може призводити до збоїв в її роботі або взагалі до відмови системи в цілому. У цьому випадку говорять про суттєві дефекти програмного коду. Деякі дефекти викликають незначні проблеми, що

не порушують процес функціонування системи, а лише дещо ускладнюють роботу з нею. У цьому випадку говорять про середні або малозначні дефекти.

Завдання тестування – це визначення умов, при яких виявляються дефекти системи і протоколювання цих умов. До завдань тестування зазвичай не входить виявлення конкретних дефектних ділянок програмного коду і ніколи не входить виправлення дефектів – це завдання з відлагодження коду, яке виконується за результатами тестування системи.

Мета застосування процедури тестування програмного коду – мінімізація кількості дефектів, особливо суттєвих, в кінцевому продукті. Слід пам'ятати, що тестування саме по собі не може гарантувати повну відсутність дефектів в програмному коді системи.

На сьогодні існує безліч підходів до вирішення завдання тестування програмного забезпечення. Слід відзначити, що ефективне тестування складних програмних продуктів часто не зводиться до чіткого слідування визначеним процедурам. Кінцевою метою будь-якого процесу тестування є забезпечення такого поняття, як якість, з урахуванням всіх або найбільш критичних для даного конкретного випадку складових.

5.1.1. Тестування системи як білої, чорної або сірої скриньки

Всі методи тестування можуть бути розділені на два класи з точки зору розгляду системи, яку тестують, – як чорної або як білої (скляної) скриньки. Окремо ще розглядають тестування системи як сірої скриньки, що є деякою комбінацією останніх двох.

Основна ідея в тестуванні системи *як чорної скриньки* полягає в тому, що всі матеріали, які доступні тестувальнику, – це вимоги до системи, що описують її поведінку, а також сама система. У цьому випадку під час тестування працювати з системою можна тільки подаючи на її входи деякі зовнішні впливи і спостерігаючи на виходах деякий результат. Всі внутрішні особливості реалізації системи приховані від тестувальника. Таким чином, система і являє собою «чорну скриньку», правильність поведінки якої по відношенню до вимог і належить перевірити. З точки зору програмного коду, чорна скринька може представляти собою набір класів (або модулів) з відомими зовнішніми інтерфейсами, але недоступним кодом реалізації.

Основне завдання тестування під час розгляду системи як чорної скриньки полягає в послідовній перевірці відповідності поведінки системи вимогам. Оскільки специфікація вимог є єдиним джерелом інформації для проведення тестування чорної скриньки, таке тестування часто називають тестуванням за вимогами.

Під час тестування системи *як білої скриньки* тестувальник має доступ не тільки до вимог до системи, її входів і виходів, а й до її внутрішньої структури (тобто до програмного коду).

Доступність програмного коду розширює можливості тестувальника тим, що він може бачити відповідність ділянок програмного коду вимогам, а отже, бачити, чи на весь програмний код існують вимоги. Програмний код, для якого

відсутні вимоги, називають кодом, що непокритий вимогами. Такий код є потенційним джерелом неадекватної поведінки системи. Крім того, прозорість системи дозволяє поглибити аналіз її ділянок, які викликають проблеми (інколи буває, що одна проблема нейтралізує іншу проблему і вони ніколи не виникають одночасно).

Таким чином, тестування системи як білої скриньки не є альтернативним варіантом до тестування чорної скриньки. Скоріше воно його доповнює, дозволяючи виявляти інший клас помилок.

Під час тестування системи як *сірої скриньки* використовуються інструменти, спрямовані на розуміння властивостей програми та оточення, з яким вона взаємодіє. Такий підхід може бути використаний для перевірки чорної скриньки з метою збільшення продуктивності тестування і продуктивності аналізу помилок.

З іншого боку, тестування сірої скриньки – це використання неповної або передбачуваної інформації про структуру системи або її архітектуру для подальшої концентрації або розширення тестування системи як чорної скриньки.

5.1.2. Тестування на різних фазах розробки програмного забезпечення

В залежності від фази розробки програмного забезпечення та глибини тестового покриття, розрізняють наступні види тестування:

- димове тестування;
- тестування нової функціональності;
- регресійне тестування;
- повторне тестування;
- приймальне тестування.

Димове тестування (англ. smoke testing) полягає в проведенні короткого циклу тестів, які виконуються для підтвердження того, що після оновлення коду (нового коду або відредагованого старого) програмне забезпечення встановлюється, запускається і виконує основні свої функції.

Під час проведення *тестування нової функціональності* (англ. new feature testing) виконують такий набір тестів, який перевіряє роботи нових функцій, що з'явилися в конкретній версії програмного забезпечення.

Регресійне тестування (англ. regression testing) – це такий вид тестування, який спрямований на перевірку змін, зроблених в програмному забезпеченні або навколишньому середовищі (наприклад, після міграції на іншу операційну систему, базу даних, сервер), і підтвердження того факту, що існуюча раніше функціональність працює як і раніше. Регресійними можуть бути як функціональні, так і нефункціональні тести.

Повторне тестування (англ. re-testing) – це таке тестування, під час якого перевіряється, що виправлення помилок, а також будь-які пов'язані з цим зміни в коді програмного забезпечення не вплинули на інші модулі і не викликали появи нових дефектів. У більшості випадків повторне тестування виконують для підтвердження успішності виправлення знайдених помилок.

Приймальне тестування (англ. acceptance testing) – це вид тестування, який спрямований на перевірку програмного забезпечення з точки зору кінцевого користувача. У приймальному тестуванні можна виділити наступні підвиди:

- *альфа-тестування* (англ. alpha testing), яке виконується на стороні організації, що займається розробкою програмного забезпечення, з можливим частковим залученням кінцевих користувачів з метою періодичної перевірки програмного забезпечення на завершальних стадіях його розробки;
- *бета-тестування* (англ. beta testing), яке виконується поза межами організації, що займається розробкою програмного забезпечення, з активним залученням кінцевих користувачів з метою отримання зворотного зв'язку від кінцевого користувача під час роботи зі стабільною версією програмного забезпечення;
- *гамма-тестування* (англ. gamma testing), що є фінальною стадією тестування перед випуском програмного забезпечення і спрямоване на перевірку виправлення незначних дефектів, що були виявлені під час проведення бета-тестування (як правило, виконується з максимальним залученням кінцевих користувачів).

5.2. Етапи тестування, їх документація

У більшості випадків тестування програмного забезпечення виконується протягом всього життєвого циклу його розробки. Це обумовлює те, що під час тестування створюється різного роду документація тестування.

Основне призначення документування процесу тестування полягає в забезпеченні гарантій того, що тестування виконується відповідно до обраних критеріїв оцінки якості, а також того, що всі аспекти поведінки системи протестовані. Крім того, документація процесу тестування використовується при внесенні змін до системи для перевірки того, що як стара, так і нова функціональність працюють коректно.

5.2.1. Тест-вимоги

Початковий етап тестування полягає в формуванні тест-вимог, які відповідають вимогам до системи, що розробляється. В документі, що визначає тест-вимоги, докладно описують, які аспекти поведінки системи повинні бути протестовані. Тобто основна мета специфікації тест-вимог полягає у визначенні, яка функціональність системи повинна бути протестована.

У найпростішому випадку одній вимозі відповідає одна тест-вимога. Однак найчастіше тест-вимоги ще більше деталізують формулювання вимог до програмного забезпечення.

5.2.2. Тест-план

Тест-вимоги визначають, що повинно бути протестовано, але не визначають, як це повинно бути зроблено. Саме тестування проводиться за так званими тестовими сценаріями (варіантами тестування). Кожен тестовий сценарій перевіряє одну ситуацію в роботі системи, а вся сукупність тестових сценаріїв повинна повністю перевіряти всю функціональність системи. Основним джерелом інформації для створення тестових сценаріїв є різного роду документація на систему, наприклад, специфікація функціональних і нефункціональних вимог.

Описи всіх тестових сценаріїв, необхідних для тестування системи, об'єднують в документ, який називається тест-планом. Існує кілька причин для об'єднання опису тестових сценаріїв в єдиному документі:

- єдина схема ідентифікації та роботи з тестовими сценаріями;
- можливість об'єднання тестових сценаріїв в групи за змістом;
- зручність організації внесення змін до тестових сценаріїв;
- визначення послідовності проведення тестування.

Оскільки тестові сценарії пишуть на основі вимог до системи, що розробляють, під час тестування необхідно впевнитися в тому, що для кожної вимоги існує хоча б один тестовий сценарій. Це досягається введенням єдиної схеми ідентифікації (наприклад, використовуючи наскрізну нумерацію), а також введенням посилань на вимоги, на основі яких написано тестовий сценарій.

Раціональним є об'єднання тестових сценаріїв, призначених для перевірки одних і тих самих модулів системи, в групи. Це пов'язано з тим, що для таких сценаріїв, як правило, дуже схожими є дані, що подаються на вхід системи. Отже, групування може дозволити виявляти помилки в самих тестах.

Під час розробки програм неминуче доводиться змінювати тестові сценарії. Документування тестових сценаріїв в єдиному джерелі дозволяє легко виявляти, які сценарії повинні бути змінені або видалені, а в яких групах необхідне створення нових сценаріїв для перевірки нової або зміненої функціональності системи.

Одна з найважливіших властивостей тестового сценарію – це його автономність. Це означає, що результат виконання тестового сценарію не повинен змінюватися в залежності від того, які тести виконувалися до нього. Як правило, автономність тестових сценаріїв досягається за рахунок повної реініціалізації системи перед виконанням кожного нового тестового сценарію. Однак для економії часу виконання всіх тестів, вони можуть бути об'єднані в послідовності, в яких кожен наступний тестовий сценарій використовує стан системи, отриманий в результаті виконання попереднього тесту. Такі зв'язані тестові сценарії повинні бути окремо позначені в документації для того, щоб зберегти коректний порядок їх слідування.

Таким чином, на підставі тест-вимог створюються тест-план – документ, який містить докладний покроковий опис того, як повинні бути протестовані тест-вимоги.

Форма подання тест-плану, в першу чергу, залежить від того, яким чином тест-план буде використовуватися в процесі тестування. Під час проведення ручного тестування зручно формувати тест-план у вигляді текстового документу, в якому окремі розділи є описом тестових сценаріїв. Кожен тестовий сценарій в такому випадку включає в себе перерахування послідовності дій, які необхідно виконати для проведення тестування, а також очікувану відповідь системи на ці дії. Така форма тест-плану незручна для автоматизованого тестування, оскільки опис природною мовою практично не піддається формалізації. Для автоматизованого тестування тестові сценарії зручно записувати на якійсь формальній мові. У цьому випадку можливо безпосереднє використання тест-планів як вхідних даних для середовища тестування.

Критерієм якості тест-плану є покриття всіх вимог для перевірки правильності функціонування програмної реалізації. Бажаною характеристикою тест-плану є перевірка виконання всіх гілок схеми програмної реалізації.

5.2.3. Звіт з тестування

За результатами виконання тестів створюються звіти про виконання тестування (вони можуть створюватися або автоматично, або вручну). Зазвичай звіти містять інформацію про те, які невідповідності вимогам були виявлені в результаті тестування, а також інформацію про частку покриття програмного коду, яка була задіяна в результаті виконання тестування системи.

За виявленими в результаті тестування невідповідностями визначають проблеми, причини виникнення яких аналізують розробники коду.

У результаті тестування зазвичай виявляються два типи проблем:

– невідповідність поведінки системи вимогам;

– неадекватна поведінка системи в ситуаціях, не передбачених вимогами.

Проблеми першого типу зазвичай викликають зміни програмного коду, набагато рідше – зміни вимог. Зміна вимог в даному випадку може знадобитися, зважаючи на їх суперечливість (кілька різних вимог описують різні моделі поведінки системи в одній і тій самій ситуації) або некоректність (вимоги не відповідають дійсності).

Проблеми другого типу однозначно вимагають зміни вимог з огляду на їх неповноту. Тобто у вимогах явно пропущена ситуація, що призводе до неадекватної поведінки системи. При цьому під неадекватною поведінкою може розумітися як повний крах системи, так і взагалі будь-яка поведінка, що не описана у вимогах.

Зміни в систему вносяться тільки після всебічного вивчення звітів тестування і локалізації проблем, що викликали невідповідність вимогам.

Процес тестування повторюється до тих пір, поки не буде досягнутий прийнятний рівень якості програмної системи.

5.3. Тестові сценарії (варіанти тестування)

Тестовий сценарій (англ. test case) – це мінімальний (атомарний) компонент тесту, розроблений з метою перевірки тієї чи іншої властивості або поведінки програмного забезпечення. Таким чином, тест являє собою набір з одного або декількох тестових сценаріїв.

Як правило, тестовий сценарій націлений тільки на один елемент об'єкта тестування. Чим менше у тестовому сценарії покриття функціональності, тим чіткіше область пошуку причини у разі знайденої помилки.

5.3.1. Форма визначення тестового сценарію

Будь-який тестовий сценарій обов'язково має включати в себе:

- унікальний ідентифікатор;
- назву (загальний опис);
- ідентифікатор пов'язаної вимоги;
- передумову;
- тестові кроки;
- очікуваний результат;
- післяумову.

Унікальний ідентифікатор тестового сценарію необхідний для зручної організації зберігання і навігації за всіма створеними сценаріями.

Назва тестового сценарію коротко (декількома словами) описує його призначення, тобто тему чи ідею, що визначають його суть.

Ідентифікатор пов'язаної вимоги має визначати вимогу, що буде тестуватися тестовим сценарієм, при цьому з однією вимогою може бути пов'язано декілька тестових сценаріїв.

Передумова – опис умов, які не мають прямого відношення до функціональності, що перевіряється, але повинні бути виконані. Тобто це список дій, які призводять систему до стану, придатного для проведення основної перевірки, або список умов, виконання яких говорить про те, що система знаходиться в придатному для проведення основного тесту стані.

Тестові кроки – опис послідовності дій, яка повинна привести нас до очікуваного результату. Тобто це список дій, що переводять систему з одного стану в інший для отримання результату, на підставі якого можна зробити висновок про задовільність реалізації поставленим вимогам до програмного продукту. Ця частина є унікальною для кожного тестового сценарію.

Тестові кроки пишуться у вигляді списку, переліку кроків, що проходить тестувальник.

Очікуваний результат – це результат, який очікують побачити після виконання тестових кроків.

Кожен з виконуваних тестових сценаріїв може давати один з трьох можливих результатів:

- позитивний результат, якщо фактичний результат дорівнює очікуваному результату (у цьому випадку тест вважається пройденим);

- негативний результат, якщо фактичний результат не дорівнює очікуваному результату (у цьому випадку знаходять помилку);
- блокування виконання тесту, якщо після одного з кроків продовження тесту неможливе (у цьому випадку так само знаходять помилку).

Післяумова – це список дій, що переводять систему в початковий стан (тобто стан до проведення тесту). Післяумова не є обов'язковою частиною, а швидше за все, є правилом хорошого тону: «насмітив – прибери за собою».

5.3.2. Особливості написання тестових сценаріїв

Під час написання тестового сценарію слід пам'ятати, що він має перевіряти лише одну конкретну вимогу до системи, для якої повинен бути тільки один очікуваний результат. У тестовому сценарії не повинно бути залежностей від інших тестових сценаріїв, нечіткого формулювання кроків або очікуваного результату, відсутності необхідної для проходження тестового сценарію інформації, зайвої деталізації.

Слід уникати залежності від інших тестових сценаріїв, оскільки зв'язаний з якимось тестовий сценарій може бути змінений. У цьому випадку стане незрозуміло, як виконувати тестовий сценарій, в якому є посилання на змінений тест. Так само, через наявність залежності між тестовими сценаріями у тестувальника може виникнути відчуття, що поведінка програми вже призводить до очікуваного стану у випадку виконання лише частини залежних тестових сценаріїв.

Під час написання тестових сценаріїв слід враховувати як позитивні, так і негативні тестові випадки. Позитивний тестовий сценарій використовує тільки коректні дані і перевіряє те, що функціональність програми, яка тестується, виконується коректно. Негативний тестовий сценарій оперує як коректними, так і некоректними даними (як мінімум, має бути використано один некоректний параметр), і ставить за мету перевірку виняткових ситуацій, а також перевіряє, що функціональність програми, яка тестується, не виконується при роботі з некоректними даними.

5.3.3. Класи даних для тестових сценаріїв

У залежності від даних, які подаються на вхід системи, і класу дефектів, на виявлення яких спрямований тестовий сценарій, розрізняють тестування:

- допустимих даних;
- граничних даних;
- відсутності даних;
- повторного введення даних;
- хибних даних.

Тестування допустимих даних пов'язане з тим, що дефекти в програмному забезпеченні найчастіше проявляються під час роботи з нестандартними даними. Однак перед пошуком таких дефектів необхідно перевірити коректність роботи програми у випадку, коли на вхід системи подають допустимі дані, які передбачені специфікацією вимог.

Тестування граничних даних – це окремий вид тестування допустимих даних, передача яких в систему може виявити її дефект (наприклад, числа, значення яких є граничними для їх типу, рядки граничної або нульової довжини). Зазвичай за допомогою тестування граничних даних виявляються дефекти, пов'язані з арифметичним порівнянням чисел або з ітераторами циклів.

Тестування відсутності даних проводиться для виявлення дефектів, які можуть виникнути у випадку, якщо системі не передаються жодні дані або передаються дані нульового розміру.

Тестування повторного введення даних проводиться для виявлення дефектів, які можуть виникнути у випадку, якщо на вхід системи передають ті самі дані. У цьому випадку про дефекти свідчать відмінності у даних, які отримують на виході. Зазвичай дефекти такого типу проявляються в результаті того, що система не оновлює значення внутрішніх змінних до початкового стану, або в результаті помилок округлення.

Тестування хибних даних перевіряє поведінку системи під час передачі їй даних, які не передбачені вимогами (наприклад, під час введення невірних символів або навіть під час надто великої швидкості введення інформації).

5.4. Декомпозиція системи під час її тестування

Оскільки сучасне програмне забезпечення має великі розміри програмного коду, під час проведення його тестування використовується метод декомпозиції.

Відповідно до цього методу система може тестуватися, виходячи з різного рівня ізольованості її компонентів. У свою чергу, розрізняють наступні види тестування системи:

- модульне тестування;
- інтеграційне тестування;
- системне тестування.

5.4.1. Модульне тестування

Модульне тестування (англ. unit testing) полягає в окремому тестуванні кожного модуля коду програми. Під час його проведення система розбивається на окремі модулі (класи, простори імен тощо), що мають певну відповідність вимогам або зовнішнім інтерфейсам.

Під час тестування відносно невеликого модуля розміром 100-1000 рядків коду є можливість перевірити якщо не всі, то, принаймні, більшість з наявних логічних гілок в його реалізації, шляхів в графі залежностей даних, граничних значень параметрів тощо. Відповідно до цього будуються критерії тестового покриття: покриті всі логічні гілки, всі оператори, всі граничні значення тощо.

Модульне тестування зазвичай виконується для кожного незалежного програмного модуля і є, мабуть, найбільш поширеним видом тестування, особливо для систем малих і середніх розмірів.

Основними принципами модульного тестування є:

- написання окремих невеликих за розміром тестів, які мають запускатися на виконання в автоматичному режимі;
- підтримка незалежності тестів один від одного і від порядку їх виконання;
- структура тестів має повторювати структуру проекту, однак самі тести мають бути відокремлені від коду основного проекту.

Знаходження помилок в коді програмного забезпечення має призводити до написання нових тестів за умови відсутності покриття модульними тестами відповідної ділянки коду. У випадку існування останніх, їх код має бути перевірений на наявність помилок, оскільки код модульних тестів також, як і основний код програми, може містити помилки.

Для забезпечення ефективної розробки модульних текстів існують спеціальні тестові платформи (фреймворки), які в більшості своїй забезпечують наступне:

- надання функціональності для побудови наборів тестів;
- наявність зручних макросів для перевірок умов в коді;
- реєстрацію помилок і формування звітів в залежності від встановленого рівня критичності знайдених дефектів;
- можливості моніторингу процесу тестування в часі.

Якщо розглядати основні вимоги, що висуваються до зазначених тестових платформ, то можна відмітити необхідність підтримки кросплатформності, забезпечення гнучкості під час виконання основних функцій, а також легкість в освоєнні та використанні забезпечуваного функціоналу.

Серед численних тестових платформ, які надають можливість створювати модульні тести для програмного коду, написаного мовою C++, найбільш поширеними є:

- CppUnit;
- Boost::Test;
- Google C++ Testing Framework.

Основні принципи розробки модульних тестів з використанням тестової платформи Boost::Test наведені в додатку В.

5.4.2. Інтеграційне тестування

Перевірка коректності всіх модулів, на жаль, не гарантує коректності функціонування системи модулів. Тому після виконання модульного тестування виконується збірка окремих модулів в більш великі і виконується *інтеграційне тестування* (англ. integration testing), коли система будується поетапно, а групи модулів додаються поступово.

Основною метою інтеграційного тестування є підтвердження того факту, що результати взаємодії між двома і більше компонентами відповідають функціональним вимогам до системи.

Інтеграційне тестування в якості вихідних даних використовує модулі, над якими вже було проведено модульне тестування, і групує їх в більш великі сутності, над якими будуть виконуватися тести. Таким чином, в процесі виконання інтеграційного тестування визначаються помилки в побудові модулів системи, яку тестують.

5.4.3. Системне тестування

Повністю реалізований програмний продукт піддається *системному тестуванню* (англ. system testing). На цьому етапі тестувальника цікавить не коректність реалізації окремих функцій і методів, а вся програма в цілому, як її бачить кінцевий користувач.

Вихідною інформацією для проведення різних видів системного тестування є вимоги до програмного забезпечення, що розробляється. Ґрунтуючись на вимогах, які тестують, розрізняють функціональне системне тестування та різні види нефункціонального системного тестування в залежності від нефункціональної вимоги, яка тестується. Під час системного тестування проводять далеко не всі з можливих видів нефункціонального тестування – конкретний їх набір залежить від системи, яку тестують. Так, найбільш часто під час проведення системного тестування зовнішніх інтерфейсів доводиться зіштовхуватися з тестуванням інтерфейсу користувача, а серед тестування атрибутів якості – з тестуванням захищеності, надійності, продуктивності, установки (інсталяції) та конфігураційним тестуванням.

Функціональне тестування (англ. functional testing) призначене для підтвердження того, що вся система в цілому веде себе відповідно до очікувань користувача, які формалізовані у вигляді системних вимог до системи.

У ході цього виду тестування перевіряються всі функції системи з точки зору її користувачів (як користувачів-людей, так і інших програм, що користуються системою).

Часто функціональне тестування асоціюють з тестуванням, під час якого систему розглядають як чорну скриньку. Однак, розглядаючи систему як білу скриньку, також можна перевіряти коректність реалізації функціональності.

Під час проведення функціонального тестування критерієм його повноти є повнота покриття тестами системних функціональних вимог.

Тестування інтерфейсу користувача (англ. user interface testing) спрямоване на перевірку сукупності засобів і методів, за допомогою яких користувач взаємодіє з програмним забезпеченням.

Під сукупністю засобів маються на увазі засоби виведення інформації користувачеві з пристрою та засоби введення інформації користувачем у пристрій.

Під сукупністю методів інтерфейсу користувача мається на увазі набір правил, закладених розробником програми, згідно з якими сукупність дій

користувача повинна привести до необхідної реакції пристрою та виконання необхідного завдання.

У залежності від особливостей програмного забезпечення, що тестується, на предмет перевірки інтерфейсу користувача, може бути проведено:

- тестування зовнішнього вигляду інтерфейсу користувача;
- тестування форм взаємодії елементів інтерфейсу з користувачем;
- перевірка доступу до внутрішньої функціональності системи за допомогою призначених для цього елементів інтерфейсу користувача;
- перевірка часу відгуку системи на різні операції, що виконуються користувачем.

Тестування зовнішнього вигляду інтерфейсу користувача спрямоване на перевірку загальних принципів розміщення елементів інтерфейсу користувача на екранних формах.

Тестування форм взаємодії елементів інтерфейсу з користувачем спрямоване на перевірку інформації, що виводиться користувачу системою, та інформації, що вводиться користувачем до системи. Тестування форми виведення інформації користувачу перевіряє зміст повідомлень, що виводяться системою, їх шрифтове та кольорове оформлення, а також випадки, для яких має виводитися те або інше повідомлення. Тестування форми введення інформації перевіряє, в якому вигляді інформація надходить від користувача до системи, при цьому перевіряють коректний формат введеної інформації, а також реакцію системи на некоректне введення.

Перевірка доступу до внутрішньої функціональності системи з елементів інтерфейсу користувача спрямована на тестування зв'язку внутрішньої логіки системи та елементів інтерфейсу. Тобто має бути перевірена коректність реакції системи на інформацію, введену користувачем певним елементом інтерфейсу.

Проведення перевірки часу відгуку системи на різні операції, що виконуються користувачем, пов'язане з тим, що підсвідомо користувач сприймає операцію тривалістю понад однієї секунди як тривалу. Якщо в цей момент система не повідомляє користувача про те, що вона виконує деяку операцію, користувач починає вважати, що система зависла або працює в неправильному режимі. Тестування більшості з таких випадків передбачає перевірку появи відповідних інформаційних повідомлень, наприклад, індикатора прогресу виконання операції (в цьому випадку значення граничного часу виконання операцій і рівномірність збільшення значення індикатора прогресу повинні перевірятися відповідними тестовими сценаріями).

Слід зауважити, що не завжди можна однозначно визначити кількість тестових сценаріїв, які знадобляться для тестування вимог до інтерфейсу користувача. Це пояснюється тим, що вимоги до інтерфейсу користувача часто здаються занадто очевидними для їх точного формулювання. Така неконкретність викликає необхідність розробки великої кількості тестових сценаріїв для тестування окремої вимоги. Більш того, деякі вимоги до інтерфейсу користувача можуть виявитися непридатними для тестування або їх тестування буде значно ускладнено. Таким чином, під час проведення аналізу

вимог до інтерфейсу користувача необхідно чітко уявляти, який елемент інтерфейсу і яким чином буде перевірятися, яка його характеристика буде вимірюватися в ході тестування.

Тестування захищеності (англ. security testing) застосовується, якщо програмне забезпечення призначене для зберігання або обробки даних, вміст яких являє собою таємницю певного роду (особисту, комерційну, державну тощо). Зазвичай до такого програмного забезпечення висуваються підвищені вимоги безпеки, які мають бути перевірені під час тестування безпеки системи.

У ході тестування захищеності перевіряється, що інформація не губиться, не пошкоджується, її неможливо підмінити, а також до неї неможливо отримати несанкціонований доступ, в тому числі за допомогою використання вразливості в самій програмній системі. Для цього доцільним буде провести тестування:

- запобігання доступу до «чужої» інформації, перевіривши розмежування і контроль доступу системою;
- запобігання доступу до залишкової інформації після видалення об'єктів з пам'яті, перевіривши очищення і захист пам'яті;
- збереження рівня секретності поза системою, перевіривши маркування і захист інформації, що передається в зовнішній світ;
- надання доступу тільки санкціонованим користувачам і забезпечення відмови в доступі всім іншим, перевіривши ідентифікацію та аутентифікацію;
- реєстрації в спеціальному журналі всіх подій системи, пов'язаних з безпекою для подальшого аналізу, перевіривши аудит подій системою;
- захищеності інформації з певним рівнем впевненості з огляду на те, як спроектована система і яку вона має архітектуру, перевіривши гарантії проектування та архітектури системи;
- функцій по забезпеченню безпеки системою у всіх режимах її роботи, провівши тестування безпеки в різних режимах роботи системи;
- засобів контролю коректності всіх правил розмежування доступу і системи безпеки в цілому, а також засоби їх відновлення під час збоїв, перевіривши цілісність і відновлення засобів захисту.

Загалом прийнято проводити сертифікацію програмних систем, призначених для зберігання даних для службового користування, секретних даних і цілком секретних даних. Існує ряд стандартів з технічного та експортного контролю, що регламентують властивості програмних систем по забезпеченню необхідного рівня безпеки і по відсутності недокументованих можливостей, які можуть бути використані зловмисником для несанкціонованого доступу до таких даних. Незважаючи на те, що процес сертифікації є самостійним і звичайно відбувається після тестування, вимоги стандартів по сертифікації програмного забезпечення можуть бути використані під час проведення тестування системи.

Тестування надійності (англ. reliability testing) застосовується, оскільки для коректної роботи системи в будь-якій ситуації необхідно впевнитися в тому, що вона відновлює свою функціональність і продовжує коректно працювати після будь-якої проблеми, яка перервала її роботу.

Збої в роботі системи мають незначну тривалість в часі і можуть бути усунені без тривалих процедур відновлення. Як правило, збій викликає короткочасне псування даних користувача без припинення роботи всієї системи в цілому. Наслідки збою можуть бути істотними з точки зору користувача, особливо, якщо дані є критично важливими, однак безперебійна робота системи не порушується.

Відмова – більш серйозний прояв дефекту в системі, під час якого вся система або її частина виходять з ладу, тобто порушується працездатний стан системи, в якому всі аспекти функціонування системи відповідають вимогам. У разі відмови системи для її повернення до нормального функціонування потрібне втручання оператора. Для програмного забезпечення причиною відмови часто є приховані дефекти, що виявляються тільки з плином великого проміжку часу (переповнення внутрішнього лічильника часу, переповнення даних тощо).

Аварія – відмова системи, під час якої система виходить з ладу таким чином, що відновлення її працездатного стану або неможливе, або займає багато часу. Слід відмітити, що для програмного забезпечення можна уникнути виникнення аварійних ситуацій за допомогою повного дублювання системи.

Таким чином, збої і відмови є причиною ситуацій, в яких працездатний стан системи порушується тимчасово, а аварії є причиною ситуацій, в яких працездатний стан системи порушується назавжди або на тривалий термін.

Для тестування надійності (безвідмовності, відмовостійкості, відновлюваності) системи імітуються збої обладнання або навколишнього програмного забезпечення, також імітують збої, викликані зовнішніми факторами. Під час аналізу поведінки системи в цьому випадку необхідно звертати увагу на два фактори – мінімізацію втрат даних в результаті збою і мінімізацію часу між збоєм і продовженням нормального функціонування системи.

Під час тестування систем реального часу особливе місце займає *тестування продуктивності* (англ. performance testing), тобто тестування вимог, що стосуються ефективності використання системних ресурсів програмним забезпеченням. Під час такого тестування виявляють здатність системи забезпечувати відповідну (допустиму) продуктивність з урахуванням займаних ресурсів у заданих умовах.

Тестування продуктивності дозволяє виявляти вузькі місця в системі, які проявляються в умовах підвищеного навантаження або за умови нестачі системних ресурсів. У цьому випадку за результатами тестування проводиться

доопрацювання системи, змінюються алгоритми виділення і розподілу системних ресурсів.

У звітах за даним видом тестування зберігають часові характеристики системи, яку тестують, найбільш розповсюдженими серед яких є кількість оброблених в одиницю часу запитів, часові інтервали між початком обробки кожного наступного запиту, рівномірність часу відповіді системи в різні моменти часу. Такі часові характеристики дозволяють досліджувати здатність програмного забезпечення забезпечити відповідний (допустимий) час відгуку, час обробки та пропускну здатність під час виконання його функцій у заданих умовах. Також у звітах зберігають такі показники, як завантаження апаратного та системного програмного забезпечення (кількість циклів процесора, виділеної пам'яті, кількість вільних системних ресурсів тощо). За такими показниками досліджують здатність програмного забезпечення використовувати відповідну (припустиму) кількість і тип ресурсів під час виконання його функцій у заданих умовах.

Зазвичай тестування продуктивності виконують при різних рівнях навантаження на систему та на різних конфігураціях обладнання. В залежності від цього розрізняють:

- *навантажувальне тестування* (англ. load testing) – це тестування, яке проводиться з метою визначення максимального рівня навантаження системи за рахунок проведення аналізу поведінки системи, що тестується, під час збільшеного навантаження на неї (наприклад, збільшеного числа одночасно працюючих користувачів або числа транзакцій);
- *стресове тестування* (англ. stress testing) – це тестування, яке проводиться з метою визначення надійності системи, що тестується, під час екстремальних або непропорційних навантаженнях і оцінює продуктивність системи у випадку, якщо поточне навантаження значно перевищить очікуваний максимум;
- *об'ємне тестування* (англ. volume testing) – це тестування, яке проводиться з метою визначення, чи здатне програмне забезпечення обробляти великі об'єми даних без помилок, тобто оцінювання продуктивності системи, що тестується, під час збільшення об'єму даних, які обробляються системою;
- *тестування стабільності* (англ. stability testing) – це тестування, яке проводиться з метою визначення, чи програмне забезпечення витримає очікуване навантаження протягом тривалого часу, і спрямоване на виявлення деградації продуктивності, що виражається в зниженні швидкості обробки інформації та збільшенні часу відповіді системи, яка тестується.

Тестування установки (англ. installation testing) – це тестування, що спрямоване на виявлення дефектів, які впливають на виконання процесу установки (інсталяції) програмного забезпечення. Тобто таке тестування

спрямоване на перевірку успішної інсталяції, налаштування, а також оновлення або видалення програмного забезпечення з комп'ютера.

Сьогодні найбільш поширене встановлення програмного забезпечення за допомогою інсталяторів (спеціальних програм, які також потребують належного тестування). У реальних умовах інсталяторів може не бути. У цьому випадку доводиться самостійно виконувати встановлення програмного забезпечення, використовуючи документацію у вигляді інструкцій або *readme* файлів, де крок за кроком описані всі необхідні дії та перевірки.

У загальному випадку тестування установки має перевіряти різні сценарії та аспекти роботи інсталятора в таких ситуаціях, як:

- нове середовище, в якому програмне забезпечення раніше не було встановлено;
- повторне встановлення програмного забезпечення з метою усунення помилок в його роботі;
- повторний запуск встановлення програмного забезпечення після помилки, яка призвела до неможливості продовження його встановлення;
- оновлення поточної версії програмного забезпечення;
- зміна поточної версії програмного забезпечення на більш стару.

Конфігураційне тестування (англ. configuration testing) застосовується, оскільки сьогодні більша частина програмного забезпечення призначена для використання на самому різноманітному обладнанні. Незважаючи на те, що сьогодні особливості реалізації периферійних пристроїв приховують драйвери операційних систем, які мають уніфікований з точки зору прикладних систем інтерфейс, все одно залишаються проблеми сумісності (як програмної, так і апаратної).

Конфігураційне тестування – спеціальний вид тестування, спрямований на перевірку роботи програмного забезпечення з різними конфігураціями системи.

Під час проведення конфігураційного тестування необхідно перевірити, що програмне забезпечення правильно працює на будь-якому підтримуваному апаратному забезпеченні разом з іншими встановленими на ньому програмними системами.

Необхідно також перевірити, що система продовжує стабільно працювати під час «гарячої» заміни будь-якого підтримуваного пристрою на аналогічний. При цьому не повинно відбуватися збоїв в роботі в момент як під час самої заміни, так і після початку роботи з новим пристроєм.

5.5. Екземпляри класів та відношення між ними на діаграмі об'єктів

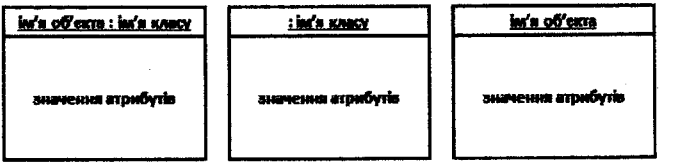
Діаграма об'єктів показує, як конкретні екземпляри класу пов'язані між собою під час виконання, в будь-який момент часу вона складається з тих самих елементів, що і діаграма класів. Інакше кажучи, діаграма об'єктів

представляє собою миттєвий знімок потоку подій, що відбуваються в програмному забезпеченні в цілому або в якомусь його окремому модулі.

5.5.1. Об'єкт на діаграмі об'єктів

Об'єкт є окремим екземпляром класу, який створюється на етапі виконання програми, і має своє власне ім'я та конкретні значення атрибутів.

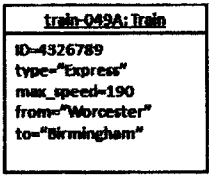
Для графічного зображення об'єкта на діаграмі об'єктів використовується такий самий символ прямокутника, що і для класів на діаграмі класів. Відмінність полягає у визначенні імен об'єктів, які обов'язково підкреслюються, при цьому запис імені об'єкта являє собою рядок тексту «ім'я об'єкта: ім'я класу», що розділений двокрапкою. Ім'я об'єкта може бути відсутнім, в цьому випадку передбачається, що об'єкт є анонімним, і двокрапка вказує на дану обставину. Відсутнім може бути і ім'я класу, тоді вказується просто ім'я об'єкта.



Атрибути кожного з об'єктів, зображених на діаграмі, мають конкретні значення.

Об'єктів одного класу в програмі може бути як завгодно багато і всі вони матимуть однаковий набір атрибутів, що визначений в класі, однак на діаграмі об'єктів значення атрибутів у кожного об'єкта свої і вони характеризують стан системи в певний момент часу (в ході виконання програми значення атрибутів об'єктів можуть змінюватися). Таким чином, імена атрибутів об'єктів на діаграмі об'єктів повинні відповідати атрибутам, які визначені у класі об'єкта або в будь-якому з батьківських класів ієрархії.

Конкретні значення атрибутів об'єктів вказуються після символу рівності «=», якому передує ім'я атрибута:

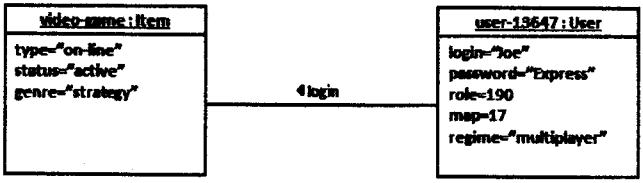


5.5.2. Зв'язки між об'єктами

Під час розробки діаграми об'єктів потрібно пам'ятати, що кожен об'єкт являє собою екземпляр відповідного класу, а відносини між об'єктами визначаються за допомогою зв'язків, які є екземплярами відповідних відносин між класами. Отже, зв'язок між двома об'єктами на діаграмі об'єктів існує тільки тоді, коли існує зв'язок між відповідними класами.

Зв'язок між об'єктами на діаграмі позначається лінією без напрямку, яка показує , що один об'єкт може передавати повідомлення (викликати операції) іншому об'єкту.

Поруч з лінією може бути вказана стрілка з ім'ям операції, що спричиняє визначений на діаграмі стан об'єктів:



Контрольні запитання до розділу

1. Що таке тестування програмного забезпечення?
2. Що перевіряють під час проведення тестування програмного забезпечення?
3. На що спрямовано виконання тестування програмного забезпечення під час його розробки?
4. До чого може призвести несподівана поведінка системи?
5. У чому полягає завдання тестування програмного забезпечення під час його розробки?
6. Що зазвичай не входить до завдань тестування програмного забезпечення під час його розробки?
7. Яка мета застосування процедури тестування програмного коду під час його розробки?
8. Яка основна ідея в тестуванні системи як чорної скриньки?
9. Що є основним завданням тестування під час розгляду системи як чорної скриньки?
10. Яка основна ідея в тестуванні системи як білої скриньки?
11. Що є основним завданням тестування під час розгляду системи як білої скриньки?
12. Що таке тестування системи як сірої скриньки?
13. Які розрізняють види тестування в залежності від фази розробки програмного забезпечення та глибини тестового покриття?
14. Що таке димове тестування?

15. У чому особливості проведення тестування нової функціональності?
16. Що таке регресійне тестування?
17. З якою метою виконують повторне тестування?
18. Чим повторне тестування відрізняється від регресійного?
19. Що таке приймальне тестування?
20. Які існують підвиди приймального тестування?
21. Яке тестування проводиться спочатку: альфа-тестування чи гамма-тестування?
22. У чому полягає початковий етап тестування?
23. Що таке тест-вимога?
24. Яка мета розробки специфікації тест-вимог?
25. Що є основним джерелом інформації для створення тестових сценаріїв?
26. Що таке тест-план, які причини його використання?
27. Що означає поняття автономності тестового сценарію?
28. Що є критерієм якості тест-плану?
29. Яка інформація міститься у звітах про виконання тестування?
30. До чого призводить виявлення під час тестування невідповідностей поведінки системи вимогам?
31. До чого призводить виявлення під час тестування неадекватної поведінки системи в ситуаціях, не передбачених вимогами?
32. До яких пір повторюється процес тестування програмного забезпечення під час його розробки?
33. Що таке тестовий сценарій (варіант тестування)?
34. Що має бути описано під час визначення тестового сценарію?
35. Що визначають передумова, тестові кроки, очікуваний результат та післяумова виконання тестового сценарію?
36. Які існують можливі результати виконання тестового сценарію?
37. Чому під час розробки тестових сценаріїв слід уникати залежності від інших тестових сценаріїв?
38. Що таке позитивні, а що таке негативні тестові випадки?
39. Які розрізняють види тестування в залежності від даних, які подаються на вхід системи, і класу дефектів, на виявлення яких спрямований тестовий сценарій?
40. З перевіркою чого пов'язано тестування допустимих даних?
41. Що таке тестування граничних даних?
42. З якою метою проводиться тестування відсутності даних?
43. З якою метою проводиться тестування повторного введення даних?
44. Що перевіряється під час тестування хибних даних?
45. Чому під час проведення тестування програмного забезпечення використовується метод декомпозиції?
46. Які існують види тестування в залежності від рівня ізольованості компонентів системи, яку тестують?
47. Що таке модульне тестування, з якою метою його виконують?
48. У чому полягають основні принципи модульного тестування?

49. До чого має призводити знаходження помилок в коді програмного забезпечення, для якого підтримується виконання модульного тестування?
50. Що забезпечують тестові платформи для розробки модульних тестів?
51. Які найбільш поширені тестові платформи, що надають можливість створювати модульні тести для програмного коду, написаного мовою C++?
52. Що таке інтеграційне тестування, з якою метою його виконують?
53. Що таке системне тестування, з якою метою його виконують?
54. Що є вихідною інформацією для проведення різних видів системного тестування?
55. Для чого призначено функціональне системне тестування?
56. Що є критерієм повноти проведення функціонального тестування?
57. Для чого призначено тестування інтерфейсу користувача?
58. Що може бути перевірено під час проведення тестування інтерфейсу користувача програмного забезпечення, від чого це залежить?
59. Тестування яких атрибутів якості найбільш часто виконують під час проведення системного тестування?
60. На що спрямовано тестування зовнішнього вигляду інтерфейсу користувача?
61. На що спрямовано тестування форм взаємодії елементів інтерфейсу з користувачем?
62. На що спрямована перевірка доступу до внутрішньої функціональності системи через елементи інтерфейсу користувача?
63. На що спрямована перевірка часу відгуку системи на різні операції?
64. Чим пояснюється те, що не завжди можна однозначно визначити кількість тестових сценаріїв, які знадобляться для тестування вимог до інтерфейсу користувача?
65. Для чого призначено тестування захищеності програмного забезпечення?
66. Перевірку чого виконують під час проведення тестування захищеності програмного забезпечення?
67. З якою метою проводять сертифікацію програмних систем, призначених для зберігання даних для службового користування, секретних даних і цілком секретних даних?
68. Для чого призначено тестування надійності програмного забезпечення?
69. Що таке збої, відмови, аварії в роботі програмного забезпечення, в чому їх різниця?
70. На які фактори необхідно звертати увагу під час аналізу поведінки системи у випадку проведення тестування її надійності?
71. Для чого призначено тестування продуктивності програмного забезпечення?
72. Що дозволяє виявляти проведення тестування продуктивності програмного забезпечення?
73. Яка інформація має міститися у звітах з тестування продуктивності програмного забезпечення?

74. Які розрізняють підвиди тестування продуктивності в залежності від виконання тестів при різних рівнях навантаження на систему та на різних конфігураціях обладнання?
75. Для чого призначено тестування установки програмного забезпечення?
76. Що таке інсталятор програмного забезпечення?
77. Які сценарії та аспекти роботи інсталяторів перевіряються під час проведення тестування установки програмного забезпечення?
78. Для чого призначено конфігураційне тестування програмного забезпечення?
79. Що необхідно перевірити під час проведення конфігураційного тестування програмного забезпечення?
80. Для чого призначена діаграма об'єктів?
81. Як графічно зображується об'єкт на діаграмі об'єктів у нотації UML?
82. Яким чином вказуються конкретні значення атрибутів об'єктів на діаграмі об'єктів у нотації UML?
83. За яких умов між двома об'єктами існує зв'язок на діаграмі об'єктів у нотації UML?
84. Як графічно позначається зв'язок між об'єктами на діаграмі об'єктів у нотації UML?

РОЗДІЛ 6
ОФОРМЛЕННЯ ЗВІТНОЇ ДОКУМЕНТАЦІЇ

За результатами розробки програмного забезпечення із використанням об'єктно-орієнтованого підходу у курсовій роботі необхідно підготувати звіт. У визначену керівником дату звіт має бути зданий на перевірку. Для перевірки необхідно здати як пояснювальну записку (роздрукований звіт), так і електронний звіт, оформлені відповідно до вимог, які викладені в цьому розділі.

Пояснювальна записка до курсової роботи обов'язково повинна бути підтверджена підписами студента та керівника роботи.

6.1. Вимоги до оформлення пояснювальної записки

Пояснювальну записку до курсової роботи слід оформляти, дотримуючись вимог до звітів в сфері науки і техніки ДСТУ 3008–95 «Документація. Звіти в сфері науки і техніки. Структура і правила оформлення».

Текст пояснювальної записки друкують на одному боці білого паперу формату А4 (210х297 мм). Весь текст пояснювальної записки необхідно вирівнювати за шириною сторінки. Абзацний відступ повинен бути однаковим впродовж усього тексту пояснювальної записки і дорівнювати 1,25 см. Відступи в тексті перед та після абзаців робити не треба.

Текст пояснювальної записки необхідно друкувати, залишаючи поля таких розмірів: ліве – не менше 25 мм, праве – не менше 10 мм, верхнє – не менше 20 мм, нижнє – не менше 20 мм.

У всьому тексті, включаючи заголовки, слід використовувати 14-й кегль, шрифт гарнітури Times New Roman та полуторний інтервал. У таблицях, у написах на рисунках, у підрисункових підписах можна використовувати 12-й кегль та одинарний інтервал.

Під час виконання пояснювальної записки необхідно дотримуватись рівномірної щільності, контрастності й чіткості зображення впродовж усього тексту.

Очікуваний обсяг тексту пояснювальної записки до курсової роботи – 25-30 сторінок (в облікований обсяг не включають список використаних джерел та додатки).

6.1.1. Структурний поділ пояснювальної записки

Пояснювальну записку умовно поділяють на вступну частину, основну частину та додатки.

Вступна частина повинна містити такі структурні елементи:

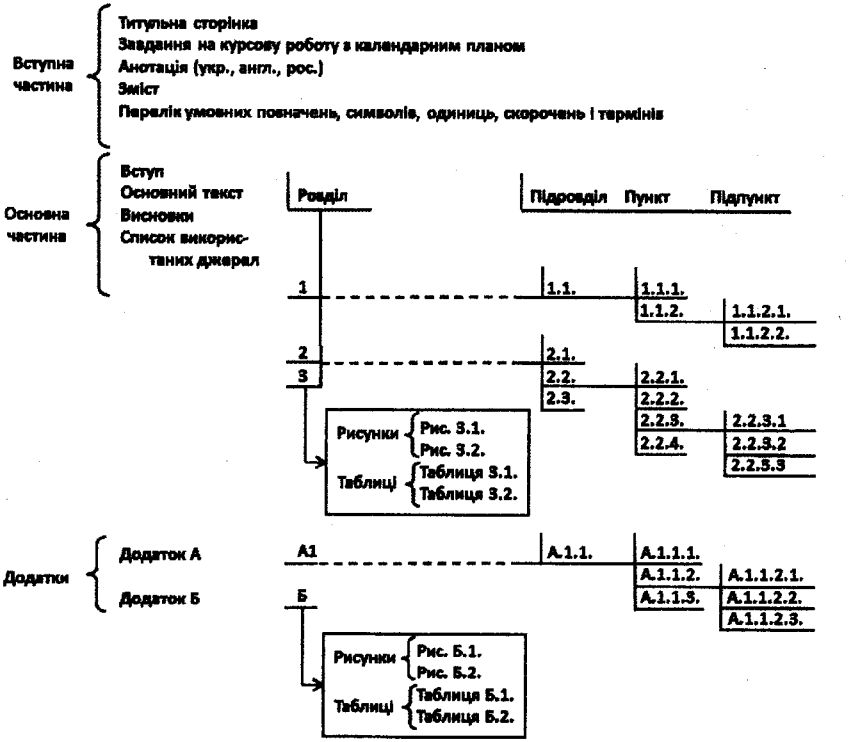
- титульна сторінка (титул);
- завдання на курсову роботу;
- календарний план;
- анотація;
- зміст;
- перелік умовних позначень, символів, одиниць, скорочень і термінів.

Основна частина повинна містити такі структурні елементи:

- вступ;
- розділи, які розкривають основний зміст курсової роботи;
- висновки;
- список використаних джерел.

Суть пояснювальної записки викладають, поділяючи матеріал на розділи, які, в свою чергу, поділяються на підрозділи, пункти та підпункти.

Структурна схема курсової роботи:



6.1.2. Змістовний склад структурних елементів

Титульна сторінка

Титульна сторінка пояснювальної записки до курсової роботи повинна містити:

– повне найменування університету;

– найменування кафедри, де виконана робота;

– назву роботи;

– прізвище, ім'я, по батькові автора;

– науковий ступінь, вчене звання, прізвище, ім'я, по батькові викладача, який є науковим керівником курсової роботи;

– місто і рік написання курсової роботи.

Титульну сторінку слід оформити відповідно до форми в додатку Г.

Завдання на курсову роботу з календарним планом

Завдання з календарним планом слід оформити відповідно до наведеної в додатку Д форми. Календарний план роздруковують на зворотній стороні аркушу із завданням на курсову роботу.

Анотація

Анотація призначена для попереднього ознайомлення з роботою. Вона в стислій, інформативній формі повинна відображати суть виконаної роботи та містити конкретні відомості про одержані результати і висновки.

Текст анотації повинен бути написаний трьома мовами: українською, російською, англійською. Тексти анотації, написані конкретною мовою, оформлюються на окремих пронумерованих сторінках кожний та розміщуються після завдання на курсову роботу з календарним планом.

Текст анотації до курсової роботи складається з двох частин.

Перша частина повинна містити інформацію про мету та завдання, використані методики, термін та місце виконання роботи, одержані результати, а також включати перелік ключових слів.

Друга частина повинна містити дані про об'єм роботи, кількість у ній таблиць, рисунків та цитованих у роботі джерел.

Текст анотації має бути стислим та інформативним. Оптимальний об'єм анотації – 8-15 рядків.

Приклади оформлення анотацій наведені в додатку Е.

Зміст

Зміст розташовують безпосередньо після анотації, починаючи з нової сторінки.

До змісту включають перелік умовних позначень, символів, одиниць, скорочень і термінів (повну назву); вступ; послідовно перелічені назви усіх розділів і підрозділів пояснювальної записки; висновки; список використаної літератури; назви додатків і номери сторінок, які містять початок матеріалу.

Приклад оформлення змісту наведений в додатку Ж.

Перелік умовних позначень, символів, одиниць, скорочень і термінів

Усі прийняті в курсовій роботі умовні позначення, символи, одиниці, скорочення і терміни пояснюють у переліку, який вміщують безпосередньо після змісту, починаючи з нової сторінки. Незалежно від цього, за першої появи цих елементів у тексті наводять їх розшифровку.

Перелік повинен розташовуватись стовпцем. Ліворуч в абетковому порядку наводять умовні позначення, символи, одиниці, скорочення і терміни, праворуч через тире – їх детальну розшифровку. Спочатку наводять скорочення українською мовою, потім – іноземними (з перекладом на мову написання пояснювальної записки).

Вступ

У вступі до курсової роботи коротко викладають мету роботи та розкривають світові тенденції розв'язання поставлених задач. Завдання вступної частини – зорієнтувати читача у тематиці праці, пояснити, чим важливим або цікавим є звернення до теми використання об'єктно-орієнтованого підходу під час розробки програмного забезпечення. Потім слід перерахувати у вигляді декількох конкретних пунктів основні завдання, які поставлені в курсовій роботі. Рекомендований обсяг вступу – 1 сторінка.

Перший розділ. Специфікація вимог

У першому розділі має бути наведена специфікація вимог програмного продукту, який розробляється в курсовій роботі.

Функціональні та нефункціональні вимоги до програмного продукту, який необхідно розробити в курсовій роботі, оформлюються в окремих підрозділах. Кожна вимога до програмного продукту оформлюється у вигляді окремого пункту, назва якого відповідає назві вимоги.

Кожен пункт, що характеризує окрему функціональну вимогу, має складатися з її специфікації, що подається як опис варіанта використання та відповідної йому діаграми в нотації UML. Опис варіанта використання слід наводити в короткій формі у вигляді таблиці, в якій визначено унікальний ідентифікатор і назва варіанта використання, його мета, дійові осіби, тип варіанта використання (основний, допоміжний або додатковий) і його короткий опис. Після таблиці слід навести діаграму варіанта використання в нотації UML, що оформлюється як рисунок.

Кожен пункт, що характеризує окрему нефункціональну вимогу, подається текстом в довільному форматі з обов'язковим зазначенням унікального ідентифікатора вимоги, дотримуючись правил опису технічних обмежень, зовнішніх інтерфейсів та атрибутів якості.

Другий розділ. Структура та логіка роботи

У другому розділі мають бути визначені логічна та фізична структури програмного продукту, який розробляється в курсовій роботі, а також наведена логіка його роботи.

Логічна структура програмного продукту оформлюється у вигляді окремого (першого) підрозділу і має включати визначення класів з детальним описом їх призначення. Для визначення класів програмного продукту слід навести діаграму класів у нотації UML з обов'язковим зазначенням коментарів до кожного з класів. Діаграма класів у нотації UML оформлюється як рисунок. Опис класів програмного продукту слід оформляти у вигляді окремих пунктів підрозділу для кожного окремого класу. В кожному пункті опису класів слід визначити призначення класу та навести опис його закритого, захищеного та відкритого інтерфейсів у вигляді таблиць. Опис кожного з типів інтерфейсу має складатися з таблиці для опису полів та з таблиці для опису методів у разі їх наявності в класі. Формат таблиці для опису полів інтерфейсу:

№ з/п	Назва	Тип	Призначення
1			
2			

Формат таблиці для опису методів інтерфейсу:

№ з/п	Сигнатура	Вхідні параметри	Повернене значення	Призначення
1				
2				

Окремим пунктом слід також описати в таблицях такого ж формату глобальні (не члени класів) функції та змінні за умови їх наявності в програмному продукті, який розробляється в курсовій роботі.

Фізична структура програмного продукту оформлюється у вигляді окремого (другого) підрозділу і має включати опис у вигляді таблиці розподілу класів, змінних, функцій та будь-яких інших компонентних структур по файлах. Формат таблиці для опису фізичної структури:

№ з/п	Назва файлу	Компоненти		Зовнішні компоненти	
		назва	опис	назва	опис
1					
2					

Таким чином, для кожного файлу слід зазначити визначені в його коді компоненти та їх призначення, а також, у разі використання, вказати зовнішні стандартні бібліотеки та файли, що підключаються, із зазначенням їх призначення.

Логіка роботи програмного продукту оформлюється у вигляді окремого (третього) підрозділу і має включати визначення часової послідовності подій, що відбувається під час роботи з програмним продуктом. Для визначення часової послідовності подій слід навести діаграму послідовності в нотації UML. Діаграма послідовності в нотації UML оформлюється як рисунок.

Третій розділ. Тестування

У третьому розділі мають бути визначені сценарії та дані для проведення тестування програмного продукту, який розробляється в курсовій роботі, а також наведені поетапні результати його роботи.

Сценарії тестування оформлюються у вигляді окремого (першого) підрозділу і мають включати опис набору варіантів тестування до програмного продукту, за яким можливе проведення перевірки функціональних та нефункціональних вимог. Кожен варіант тестування оформлюється у вигляді окремого пункту, назва якого відповідає назві варіанта тестування. Опис варіанта тестування слід наводити у вигляді таблиці, в якій визначено унікальний ідентифікатор і назву варіанта тестування, ідентифікатор пов'язаної з ним вимоги до програмного продукту, передумову, тестові кроки, очікуваний результат та післяумову його виконання.

Тестові дані оформлюються у вигляді окремого (другого) підрозділу і мають включати опис даних для проходження набору всіх визначених варіантів тестування. Для опису тестових даних слід навести діаграму об'єктів у нотації UML. Діаграма об'єктів у нотації UML оформлюється як рисунок. Після діаграми об'єктів слід навести копію з екрану вмісту файлу, в якому зберігаються тестові дані. На діаграмі об'єктів та у файлі має бути не менше 10 об'єктів, що є елементами контейнера, з описом реальних даних для перевірки роботи програми.

Поетапні результати роботи програмного продукту оформлюються у вигляді окремого (третього) підрозділу і мають бути подані як копії з екрану роботи програми під час виконання кожного з усіх пунктів меню користувача.

Висновки

У висновках наводять оцінку одержаних результатів роботи або її окремого етапу (негативних також). Висновки мають продемонструвати ступінь реалізації поставленої мети та завдань. На підставі отриманих результатів у висновках можна надати рекомендації, що визначають потрібні, на думку автора, подальші вдосконалення з приводу архітектури та коду розробленої в курсовій роботі програми. Обсяг висновків не повинен перевищувати двох сторінок.

Список використаних джерел

Після розділу висновків у роботі подають список використаних джерел. В цьому розділі пояснювальної записки подається нумерований список всіх джерел, на які існують посилання в роботі. У відповідних місцях тексту мають бути посилання.

Додатки

У додатках до курсової роботи має бути наведений повний текст коду програми (лістинг), у якому складні або ключові моменти повинні бути прокоментовані. Написання коду має бути оформлено відповідно до правил, наведених в додатку З.

У додатках також необхідно навести повний текст коду, розроблених для тестування роботи контейнера модульних тестів.

У всьому тексті, що відповідає коду, слід використовувати 12-й кегль, шрифт гарнітури Courier New та одиничний інтервал.

6.1.3. Оформлення структурних елементів пояснювальної записки

Заголовки структурних елементів пояснювальної записки і заголовки розділів слід розташовувати посередині рядка (без врахування абзацного відступу) і друкувати великими літерами без крапки в кінці, не підкреслюючи.

Розділи і підрозділи повинні мати заголовки.

Номер розділу ставлять після слова "РОЗДІЛ" без крапки в кінці. Потім з нового рядка друкують заголовок розділу великими літерами по центру сторінки без крапки в кінці.

Заголовки підрозділів і пунктів слід починати з абзацного відступу і друкувати маленькими літерами, крім першої великої, не підкреслюючи, без крапки в кінці. Якщо заголовок складається з двох і більше речень, їх розділяють крапкою.

Кожен структурний елемент пояснювальної записки і розділ слід починати з нової сторінки.

Відстань між назвою структурного елемента пояснювальної записки, а також між назвою розділу і подальшим текстом має дорівнювати двом порожнім рядкам.

Не допускається розміщувати назву розділу, підрозділу, а також пункту й підпункту в нижній частині сторінки, якщо після неї розміщено тільки один рядок тексту. Потрібно, щоб внизу сторінки лишалося мінімум два рядки тексту.

У змісті номери та назви розділів, підрозділів та пунктів друкують таким самим шрифтом, як і в тексті пояснювальної записки, додержуючись полуторного інтервалу. Проміжок між назвою та номером сторінки заповнюється крапками.

6.1.4. Нумерація сторінок та частин у пояснювальній записці

Сторінки слід нумерувати арабськими цифрами, додержуючись наскрізної нумерації впродовж усього тексту пояснювальної записки. Номер сторінки проставляють у правому верхньому куті сторінки без крапки в кінці.

Титульну сторінку і завдання на курсову роботу з календарним планом включають до загальної нумерації сторінок, але номери сторінок на них не проставляють. Завдання на курсову роботу з календарним планом враховується як одна сторінка (друга сторінка пояснювальної записки).

Розділи, підрозділи, пункти, підпункти пояснювальної записки слід нумерувати арабськими цифрами. Розділи повинні мати порядкову нумерацію в межах викладення суті роботи і позначатися арабськими цифрами без крапки в кінці, наприклад, 1, 2, 3 і т. д.

Підрозділи повинні мати порядкову нумерацію в межах кожного розділу. Номер підрозділу складається з номера розділу і порядкового номера підрозділу, відокремлених крапкою, наприклад, 1.1., 1.2. і т. д.

Пункти повинні мати порядкову нумерацію в межах кожного підрозділу. Номер пункту складається з номера розділу, порядкового номера підрозділу та порядкового номера пункту, відокремлених крапкою, з крапкою після номера, наприклад, 1.1.1., 1.1.2. і т. д. Потім у тому ж рядку наводять заголовок пункту (пункт може не мати заголовка).

Підпункти нумерують у межах кожного пункту за такими ж правилами, як пункти.

6.1.5. Переліки

Переліки, за потреби, можуть бути наведені всередині пунктів або підпунктів.

Перед переліком ставлять двокрапку.

Перед кожною позицією переліку слід ставити малу літеру української абетки з дужкою або, не нумеруючи, дефіс (перший рівень деталізації).

Для подальшої деталізації переліку слід використовувати арабські цифри з дужкою (другий рівень деталізації).

Переліки першого рівня деталізації друкують малими літерами з абзацного відступу, другого рівня – відступом відносно місця розташування переліків першого рівня.

6.1.6. Скорочення

У роботі бажано використовувати терміни, рекомендовані існуючими нормативними документами. Скорочення слів і абревіатури також повинні бути загальноприйнятими.

Замість скорочень «і т.д.» (і так далі), «і т.ін.» (і таке інше), «подібні» в роботі рекомендується використовувати термін «тощо».

Якщо в роботі прийнято особливу систему скорочування слів або назв, її подають у розділі «Перелік умовних позначень, символів, одиниць, скорочень і термінів». Повний запис лексеми, що скорочується, або словосполучення треба

наводити тоді, коли її вперше згадують у тексті, після неї в дужках подають її скорочення (абревіатуру), у подальших згадуваннях рекомендовано вживати прийняте скорочення без відмінкових закінчень. Якщо відсутність відмінкових закінчень спричиняє неоднозначне розуміння положення роботи, лексему, що скорочується, або словосполучення подають повністю.

6.1.7. Рисунки

Рисунки блок-схем та діаграм у нотації UML слід розміщувати в пояснювальній записці безпосередньо після тексту, де вони згадуються вперше, або на наступній сторінці.

На всі рисунки мають бути посилання в тексті пояснювальної записки.

Рисунки мають мати підпис, що розміщують під ілюстрацією. Підпис звичайно має чотири основних елементи:

- найменування графічного сюжету, що позначається скороченим словом «Рис.»;
- порядковий номер, який вказується без знаку номера арабськими цифрами;
- тематичний заголовок, що містить текст із якомога стислою характеристикою зображеного;
- експлікацію, яка не є обов'язковою частиною підпису рисунка і будується так: деталі сюжету позначають цифрами, які виносять у підпис, супроводжуючи їх текстом.

Приклад оформлення рисунка, в підписі якого є експлікація:

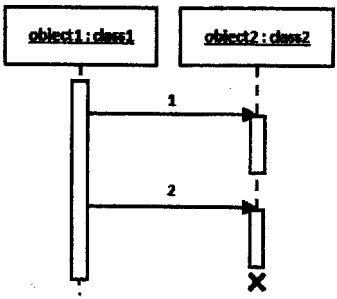


Рис.3.2. Діаграма послідовності в нотації UML:
1 – виведення даних на екран;
2 – видалення об'єкта.

Текст підпису до рисунка слід виконувати тільки 14-м кеглем шрифту без крапки в кінці. Текст експлікації до рисунка слід виконувати 12-м кеглем шрифту з одинарним інтервалом без крапки в кінці. Перенесення тексту експлікації або назви рисунка на наступну сторінку не допускається. Розташування тексту справа та зліва від рисунка не допускається.

Після назви рисунка перед подальшим текстом пояснювальної записки необхідно залишити один незаповнений рядок.

Рисунки слід нумерувати арабськими цифрами порядковою нумерацією в межах розділу. Номер рисунка складається з номера розділу і порядкового номера рисунка, відокремлених крапкою, наприклад, Рис. 3.2. – другий рисунок третього розділу.

Якість ілюстрацій повинна забезпечувати їх чітке відтворення. Для якісної подачі ілюстративного матеріалу можна використовувати папір форматів, більших за А4. У цьому випадку аркуші паперу складають у послідовності, що вказана в таблиці додатку И цифрами на лініях згинів.

Аркуші паперу всіх форматів слід складати спочатку вздовж ліній, перпендикулярних (поздовжніх) до основного напису, а потім вздовж ліній, паралельних (поперечних) до основного напису. Аркуші після складання повинні мати основний напис на лицьовій стороні.

6.1.8. Таблиці

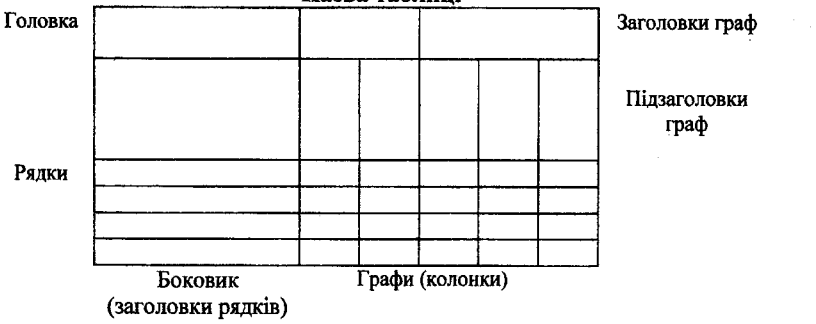
Таблицю слід розташовувати безпосередньо після тексту, у якому вона згадується вперше, або на наступній сторінці. На всі таблиці мають бути посилання в тексті пояснювальної записки.

Кожна таблиця повинна мати назву, яку розміщують над таблицею і друкують симетрично до тексту. Назву і слово «*Таблиця*» починають з великої літери. Назву не підкреслюють, її друкують жирним шрифтом. Назва має бути стислою і відбивати зміст таблиці.

Приклад побудови таблиці:

Таблиця (номер)

Назва таблиці



За логікою побудови таблиці її логічний суб'єкт, або підмет (позначення тих предметів, які в ній характеризуються), розміщують у боковику, головці чи в них обох, а не у графі; логічний предмет таблиці, або присудок (тобто дані, якими характеризується присудок), – у графі, а не в головці чи

боковику. Кожен заголовок над графою стосується всіх даних цієї графи, кожен заголовок рядка в боковику – всіх даних цього рядка.

Заголовок кожної графи в головці таблиці має бути за можливістю коротким. Слід уникати повторів тематичного заголовка в заголовках граф, одиниці виміру зазначати у тематичному заголовку, виносити до узагальнюючих заголовків слова, що повторюються.

Боковик, як і головка, вимагає лаконічності. Повторювані слова тут також виносять в об'єднувальні рубрики; загальні для всіх заголовків боковика слова розміщують у заголовку над ним.

У прографці повторювані елементи, які мають відношення до всієї таблиці, виносять в тематичний заголовок або в заголовок графи; однорідні числові дані розміщують так, щоб їх класи співпадали; неоднорідні – посередині графи; лапки використовують тільки замість однакових слів, які стоять одне під одним.

Заголовки граф повинні починатися з великих літер, підзаголовки – з маленьких, якщо вони складають одне речення із заголовком, і з великих, якщо вони є самостійними. У кінці заголовків і підзаголовків таблиць крапки не ставлять.

Якщо дані в якомусь рядку не приводяться, то у графі ставлять прочерк.

Таблиці слід нумерувати арабськими цифрами порядковою нумерацією в межах розділу. Номер таблиці складається з номера розділу і порядкового номера таблиці, відокремлених крапкою, наприклад, таблиця 2.1 – перша таблиця другого розділу.

Якщо рядки або графи таблиці виходять за межі формату сторінки, таблицю поділяють на частини, розміщуючи одну частину під одною, або поруч, або переносячи частину таблиці на наступну сторінку, повторюючи в кожній частині таблиці її головку і боковик.

Слово «*Таблиця*» з зазначенням номера вказують один раз справа над першою частиною таблиці без крапки в кінці, над іншими частинами пишуть: «*Продовж. табл.*» із зазначенням номера таблиці.

Між текстом пояснювальної записки та заголовком таблиці, а також після таблиці перед подальшим текстом необхідно залишити по одному порожньому рядку.

6.1.9. Формули

Формули слід розташовувати безпосередньо після тексту, в якому вони згадуються.

Найбільші, а також довгі і громіздкі формули, котрі мають у складі знаки суми, добутку, диференціювання, інтегрування, розміщують на окремих рядках з абзацного відступу. Для економії місця кілька коротких однотипних формул, відокремлених від тексту, можна подати в одному рядку, а не одну під одною. Невеликі і нескладні формули, що не мають самостійного значення, вписують всередині рядків тексту.

Переносити формули на наступний рядок допускається тільки на знаках виконуваних операцій, повторюючи знак операції на початку наступного рядка. При перенесенні на знакові операції множення застосовують знак «×».

Пояснення значень символів і числових коефіцієнтів треба подавати безпосередньо під формулою в тій послідовності, в якій вони дані у формулі. Рядок пояснення слід починати без абзацного відступу словом «де» без двокрапки.

Вимоги до шрифтів та їх стилю у формулах:

Style	Font	Character Format	
		Bold	Italic
Text	Times New Roman	☐	☐
Function	Times New Roman	☐	☐
Variable	Times New Roman	☐	☑
L.C. Greek	Symbol	☐	☐
U.C. Greek	Symbol	☐	☐
Symbol	Symbol	☐	☐
Matrix-Vector	Times New Roman	☑	☐
Number	Times New Roman	☐	☐

Language:	
Text style	Any
Other styles	Any

Вимоги до розміру символів у формулах:

Full	14 pt
Subscript/Superscript	9 pt
Sub- Subscript/Superscript	7 pt
Symbol	18 pt
Sub-symbol	12 pt

Нумерувати слід лише ті формули, на які є посилання у наступному тексті. Інші нумерувати не рекомендується.

Формули в пояснювальній записці (за винятком формул, наведених у додатках) слід нумерувати порядковою нумерацією в межах розділу. Номер формули складається з номера розділу і порядкового номера формули, відокремлених крапкою.

Номер формули зазначають на рівні формули в круглих дужках у крайньому правому положенні на рядку. Номер, який не вміщується у рядку з формулою, переносять у наступний нижче формули. Номер формули при її перенесенні вміщують на рівні останнього рядка. Номер формули-дробу

подають на рівні основної горизонтальної риски формули. Номер групи формул, розміщених на окремих рядках і об'єднаних фігурною дужкою, ставиться справа від вістря дужки, яке знаходиться в середині групи формул.

Загальне правило пунктуації в тексті з формулами таке: формула входить до речення як його рівноправний елемент. Тому в кінці формул і в тексті перед ними розділові знаки ставлять відповідно до правил пунктуації.

Двокрапку перед формулою ставлять лише у випадках, передбачених правилами пунктуації: у тексті перед формулою є узагальнююче слово або якщо цього вимагає побудова тексту, що передує формулі.

Розділовими знаками між формулами, котрі йдуть одна за одною і не відокремлені текстом, можуть бути кома або крапка з комою безпосередньо за формулою до її номера.

Формули повинні приводитися в тексті тільки один раз. В подальшому на них необхідно давати посилання.

6.1.10. Посилання

Посилання в тексті пояснювальної записки на джерела слід зазначати порядковим номером за переліком посилань, виділеним двома квадратними дужками, наприклад,«... у роботі [5] показано, що ...».

Посилання на джерела в мережі Інтернет включаються в загальний перелік посилань.

На джерела необхідно посилатися в порядку їх згадування в тексті пояснювальної записки. Нумерація використаних джерел виконується наскрізна по всім розділам арабськими цифрами.

За умови посилання на розділи, підрозділи, пункти, рисунки, таблиці, додатки зазначають їх номери, при цьому слід писати: «... у розділі 4 ...», «... дивись 2.1 ...», «... за 3.3.4 ...», «... на рис. 1.3 ...», «... у табл. 3.2 ...», «... у додатку А ...», «... за рівнянням (1.2) ...».

6.1.11. Цитування

Цитування варто звести до мінімуму. До нього слід вдаватися лише тоді, коли воно справді необхідне.

Під час оформлення речень з використанням цитат слід дотримуватися кількох формальних правил:

- текст цитати повинен подаватися у лапках і супроводжуватися посиланням на відповідне джерело;
- текст цитати повинен бути дослівним;
- текст цитати не повинен бути надто довгим;
- трикрапкою позначають вилучення певних фрагментів у тексті цитати, що дозволяє уникнути довгого цитування;
- пояснення, які розміщуються всередині тексту цитати, слід наводити у квадратних дужках;
- текст цитати, що є незавершеним реченням, граматично узгоджується з текстом.

6.1.12. Список використаних джерел

Список використаних джерел має відображати ступінь фундаментальності проведеної роботи та має охоплювати всі види опублікованих та неопублікованих документів, які були використані під час написання курсової роботи.

Бібліографічні описи джерел у списку наводять відповідно до чинних стандартів з бібліотечної та видавничої справи.

Структура бібліографічного опису може містити лише обов'язкові або обов'язкові та факультативні (необов'язкові) елементи.

У будь-якому бібліографічному описі мають бути наведені обов'язкові елементи, оскільки вони забезпечують ідентифікацію використаного джерела. Такий опис називається стислим і містить прізвище та ініціали автора (авторів), назву, відомості про видання, видавництво, рік видання, том (якщо є), кількість сторінок.

На відміну від стислого, розширений бібліографічний опис передбачає наявність ще й факультативних елементів. Такий опис додатково містить бібліографічні відомості щодо загального визначення матеріалу, паралельний заголовок, додаткові відомості про назву, відомості щодо наявності довідкового чи ілюстративного матеріалу тощо.

Джерелом інформації для складання бібліографічного опису є документ в цілому. Опис документів здійснюється за титульним аркушем, титульним екраном, етикеткою, наклейкою тощо.

Мова бібліографічного опису, як правило, відповідає мові вихідних відомостей документів.

Проміжки між знаками та елементами опису є обов'язковими, вони використовуються для розрізнення знаків граматичної пунктуації та приписаної (умовних розділових знаків). Бібліографічний опис складається за сучасною орфографією.

Числівники в описі, як правило, наводять так, як вони подані у джерелі інформації. Однак римські цифри і числівники у словесній формі замінюють арабськими цифрами при позначенні порядкових номерів видання; дат виходу документа; номерів випусків багаточастинного документа, кількості класів чи курсів навчальних закладів.

Відомості про джерела нумеруються арабськими цифрами. Номер ставиться перед бібліографічним записом і відокремлюється від нього крапкою.

Приклади оформлення бібліографічного опису в списку використаних джерел наведені у додатку К.

6.1.13. Додатки

Додатки слід оформлювати як продовження пояснювальної записки на наступних сторінках, розташовуючи додатки в порядку появи посилань на них у тексті.

Кожний новий додаток повинен починатися з нової сторінки. Додаток повинен мати заголовок, надрукований угорі малими літерами з першої великої

симетрично відносно тексту сторінки. Посередині рядка над заголовком малими літерами з першої великої друкується слово «Додаток», а далі – велика літера, що позначає додаток.

Додатки слід позначати послідовно великими літерами української абетки, за винятком літер Ґ, Є, І, Ї, Й, О, Ч, Ь, наприклад, додаток А, додаток Б і т.д.

Якщо в роботі є тільки один додаток, то він позначається як додаток А.

Текст кожного додатка, у разі необхідності, може бути поділений на розділи й підрозділи, які нумерують у межах кожного додатка. У цьому разі перед кожним номером ставлять позначення додатка (літеру) і крапку, наприклад, А.2 – другий розділ додатка А; В.3.1 – підрозділ 3.1 додатка В.

Ілюстрації, таблиці та формули, що є у тексті додатка, слід нумерувати в межах одного додатка. Наприклад: «Рисунок Г.3» – третій рисунок додатку Г, «Таблиця А.2» – друга таблиця додатка А, «формула (А.1)» – перша формула додатка А.

Додатки повинні мати спільну з рештою пояснювальної записки наскрізну нумерацію сторінок.

6.2. Перевірка оформлення пояснювальної записки

Після завершення написання тексту пояснювальної записки необхідно виконати перевірку її оформлення відповідно до вимог, які були наведені в цьому розділі. Особливо слід звернути увагу та перевірити наступне:

- правильність нумерації сторінок в змісті (за умови додержання вимог до оформлення пояснювальної записки, зміст має починатися з 6-ї сторінки);
- правильність нумерації та коректність використаних символів для позначення додатків;
- правильність нумерації рисунків, таблиць та формул (нумерація підряд у межах кожного розділу);
- наявність в тексті посилань на всі пронумеровані в роботі рисунки, таблиці та формули;
- наявність в тексті посилань на всі джерела, використані в роботі;
- наявність в тексті посилань на всі наявні додатки;
- розташування всіх заголовків має бути разом з текстом, що йде далі, а не відриватися від нього в кінці попередньої сторінки;
- розташування всіх підписів до рисунків має бути разом з рисунком, що йде попереду, а не відриватися від нього на початку наступної сторінки (те саме має виконуватися щодо таблиць);
- відсутність великих вільних місць на сторінках (це можна виправити, змінивши розміри якогось рисунка чи дописавши якесь речення);
- правильність даних про об'єм роботи, кількість у ній таблиць, рисунків та цитованих джерел у всіх трьох анотаціях (українською, російською, англійською мовами).

Після друку пояснювальної записки необхідно виконати перевірку:

- якості друку всіх сторінок;
- якості друку інформації на рисунках (за потреби рисунки мають бути роздруковані на папері форматів, більших за А4);
- завдання на курсову роботу та календарний план мають бути роздруковані на одному аркуші паперу з різних сторін;
- наявність підпису студента на сторінці з календарним планом виконання курсової роботи;
- коректність нумерації сторінок в змісті (автоматичні посилання можуть бути хибними або взагалі бути відсутніми після друку, що неприпустимо);
- правильність слідування сторінок;
- послідовність складування аркушів паперу за наявності матеріалу, роздрукованого на папері форматів, більших за А4.

6.3. Вимоги до оформлення електронного звіту

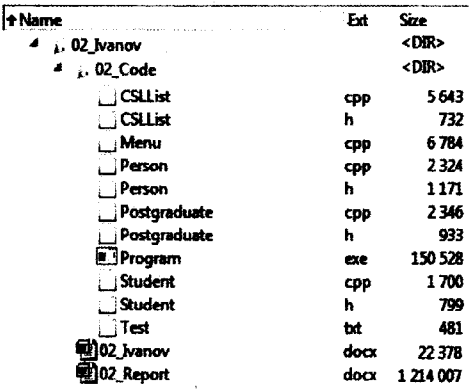
Електронний звіт студента з курсової роботи повинен містити наступні файли:

- файл у форматі *.doc або *.docx з повним текстом пояснювальної записки;
- файл у форматі *.doc або *.docx, в якому окремо дублюється лише та частина пояснювальної записки до курсової роботи, що відповідає тексту анотації, написаному трьома мовами (українською, російською, англійською);
- початкові файли програми (надати потрібно лише ті файли, які необхідні для компіляції і компонування програмного продукту в середовищі Microsoft Visual Studio);
- виконуваний exe-файл розробленої в курсовій роботі програми (файл має виконуватися незалежно від його розташування на комп'ютері користувача);
- файл з тестовими даними, який містить не менше 10 об'єктів, що є елементами контейнера, і які було використано під час проведення тестування розробленої в курсовій роботі програми.

Електронний звіт може знаходитися на будь-якому носієві інформації та має бути переданий викладачу після закінчення виконання курсової роботи, але не пізніше встановленого викладачем терміну.

Структура каталогів та файлів має бути строго визначена відповідно до виконуваного варіанта.

Приклад структури каталогів та файлів для електронного звіту з курсової роботи, що були виконані для 2-го варіанта студентом Івановим І.І., наступний:



Name	Ext	Size
02_Ivanov		<DIR>
02_Code		<DIR>
CSLList	cpp	5 643
CSLList	h	732
Menu	cpp	6 784
Person	cpp	2 324
Person	h	1 171
Postgraduate	cpp	2 346
Postgraduate	h	933
Program	exe	150 528
Student	cpp	1 700
Student	h	799
Test	txt	481
02_Ivanov	docx	22 378
02_Report	docx	1 214 007

6.4. Перевірка оформлення електронного звіту

Після завершення підготовки електронного звіту перед його здачею необхідно виконати перевірку, звернувши увагу на наступне:

- всі файли електронного звіту містять потрібну інформацію, їх версії сумісні між собою (тексти коду програми і коду розроблених для тестування роботи контейнера модульних тестів, що наведені у додатках пояснювальної записки до курсової роботи, відповідають вмісту відповідних файлів з кодом; файл з тестовими даними містить дані, які були використані під час проведення тестування програмного продукту);
- папка з файлами програми має назву, яка складається з номера варіанта завдання, що було виконано в курсовій роботі, і через символ нижнього підкреслювання «_» англійського слова «Code»;
- в папці з файлами програми наявні лише файли з кодом форматів *.h та *.cpp, виконуваний exe-файл програми і файл з тестовими даними;
- файл з повним текстом пояснювальної записки має назву, яка складається з номера варіанта завдання, що було виконано в курсовій роботі, і через символ нижнього підкреслювання «_» англійського слова «Report»;
- файл з анотаціями має назву, яка складається з номера варіанта завдання, що було виконано в курсовій роботі, і через символ нижнього підкреслювання «_» прізвища студента англійською мовою;
- батьківська папка електронного звіту має назву, яка складається з номера варіанта завдання, що було виконано в курсовій роботі, і через символ нижнього підкреслювання «_» прізвища студента англійською мовою;
- вміст електронного звіту заархівований для передачі викладачу (ім'я архіву відповідає назві батьківської папки електронного звіту.

1. Лафоре, Р. Объектно-ориентированное программирование в C++ / Р. Лафоре; пер. с англ. А. Кузнецов, М. Назаров, В. Шрага. – СПб. : Питер, 2018. – 928 с.
2. Грицюк, Ю. І. Об'єктно-орієнтоване програмування мовою C++ : навч. посіб. / Ю. І. Грицюк, Т. Є. Рак – Львів : Вид-во Львів. ДУ БЖД, 2011. – 404 с.
3. Кравець, П. О. Об'єктно-орієнтоване програмування : навч. посібн. / П. О. Кравець. – Львів : Вид-во Львів. політехніки, 2012. – 624 с.
4. Пелешко, Д. Д. Об'єктні технології C++11 : навч. посібн. / Д. Д. Пелешко, В. М. Теслюк. – Львів : Вид-во Львів. політехніки, 2013. – 360 с.
5. Гома, Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений / Х. Гома; пер. с англ. А. Слинкин. – М. : ДМК Пресс, 2016. – 700 с.
6. Дудзяний, І. М. Об'єктно-орієнтоване моделювання програмних систем : навч. посібн. / І. М. Дудзяний. – Львів : Вид. Центр ЛНУ імені Івана Франка, 2007. – 108 с.
7. Дейтел, Х. Как программировать на C++ / Х. Дейтел, П. Дейтел; пер. с англ. В. Тимофеев. – М. : Бином, 2010. – 1456 с.
8. Шилдт, Г. C++. Полное руководство / Г. Шилдт; пер. с англ. – М. : Вильямс, 2017 – 800 с.
9. Шилдт, Г. C++. Базовый курс / Г. Шилдт; пер. с англ. Н. Ручко. – М. : Вильямс, 2015 – 624 с.
10. Березин, Б. И. Начальный курс C и C++ / Б. И. Березин, С. Б. Березин – М. : Диалог-Мифы, 2017. – 288 с.
11. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп; пер. с англ. Н. Мартынов. – М. : Бином, 2012. – 1136 с.
12. Прата, С. Язык программирования C++ / С. Прата. – М. : Вильямс, 2007. – 1184 с.
13. Страуструп, Б. Программирование. Принципы и практика с использованием C++ / Б. Страуструп; пер. с англ. И. Красиков. – М. : Вильямс, 2016. – 1328 с.

14. Солтер, Н. А. C++ для профессионалов / Н. А. Солтер, М. Л. Клепер; пер. с англ. Н. Ручко. – М. : Вильямс, 2006. – 912 с.

15. Седжвик, Р. Алгоритмы на C++ / Р. Седжвик; пер. с англ. А. Моргунов. – М. : Вильямс, 2017. – 1056 с.

16. Кормен, Т. Х. Алгоритмы: построение и анализ / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн; пер. с англ. И. Красиков. – М. : Вильямс, 2016. – 1328 с.

17. Кнут, Д. Э. Искусство программирования. Сортировка и поиск / Д. Э. Кнут; пер. с англ. В. Тертышный, И. Красиков. / Т. 3. – М. : Вильямс, 2017. – 824 с.

18. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт; пер. с англ. Ф. Ткачев. – М. : ДМК Пресс, 2016, – 272 с.

19. Грицюк, Ю. І. Аналіз вимог до програмного забезпечення / Ю. І. Грицюк. – Львів : Вид-во Львів. політехніки, 2018. – 456 с.

20. Лєвус Є. В. – Життєвий цикл програмного забезпечення : навчальний посібник / Є. В. Лєвус. – Львів : Вид-во Львів. політехніки, 2017. – 208 с.

ДОДАТОК А
Варіанти завдань

Надалі наведено варіанти завдань для розробки ієрархії класів.

Номер варіанта	Перелік класів	Абстрактний клас
1	Республіка, монархія	Держава
2	Студент, аспірант	Особа
3	Область, місто	Місце
4	Двигун внутрішнього згорання, турбореактивний двигун	Двигун
5	Завод з утилізації автомобілів, суднобудівельний завод	Підприємство
6	Художня книга, підручник	Друкарське видання
7	Гарантійний ремонт техніки, регламентне обслуговування техніки	Задача сервісного центру
8	Квитанція (касовий чек), накладна	Документ
9	Ковбасний виріб, молочний продукт	Товар
10	Пасажирський поїзд, вантажний поїзд	Поїзд
11	Інженер, директор	Робітник організації
12	Театр, кінотеатр	Установа культури
13	Тест, випускний іспит	Випробування
14	Деталь, вузол	Виріб
15	Цех пошиття одягу, меблевий цех	Цех
16	Теплохід, вітрильник	Судно
17	Стаціонар, поліклініка	Медична установа
18	Дефект, задача для розробки	Завдання в системі управління проектом з розробки програмного забезпечення
19	Страхова компанія, банк	Комерційна організація
20	Лялька, конструктор	Іграшка
21	Олівець, фарба	Канцелярське приладдя
22	Викладач, завідувач кафедрою	Співробітник кафедри

Надалі наведено варіанти завдань для розробки контейнера.

Номер варіанта	Тип контейнера
1, 12	Однозв'язний лінійний список
2, 13	Однозв'язний кільцевий список
3, 14	Двозв'язний лінійний список
4, 15	Двозв'язний кільцевий список
5, 16	Стек
6, 17	Черга
7, 18	Черга з пріоритетом
8, 19	Дек з обмеженим входом (з кінця дека можна тільки видаляти елементи)

9, 20	Дек з обмеженим виходом (в кінець дека можна тільки додавати елементи)
10, 21	Динамічний масив (розмір масиву задається як аргументи конструктора)
11, 22	Двійкове дерево пошуку

Надалі наведено варіанти завдань для розробки запиту.

Номер варіанта	Завдання
1	Вивести назву республіки, що має найбільшу площу
2	Вивести прізвища всіх студентів, які навчаються на заданому курсі
3	Вивести назви всіх міст, що входять в задану область
4	Обчислити сумарну потужність всіх двигунів заданого типу
5	Підрахувати загальну кількість робочих на підприємствах в заданому місті
6	Підрахувати загальну кількість підручників із заданого предмету
7	Підрахувати загальну кількість гарантійних ремонтів техніки заданої фірми
8	Вивести номери документів, що були виписані після заданої дати
9	Вивести найменування продуктів з жирністю не вище заданої
10	Вивести характеристики поїзда за заданим номером
11	Вивести інформацію про робітника організації за його ідентифікаційним номером
12	Вивести назву театру, що має найстарішу дату заснування
13	Вивести теми всіх тестів, які відносяться до заданого предмету
14	Вивести найменування всіх деталей, що входять в заданий вузол
15	Обчислити сумарну потужність всіх верстатів в цехах заданого типу
16	Підрахувати загальну кількість суден, мінімальний склад екіпажу яких перевищує задане значення
17	Підрахувати загальну кількість лікарів, що працюють у медичних установах заданого міста
18	Підрахувати загальну кількість дефектів, знайдених в заданому компоненті програмного забезпечення
19	Вивести найменування комерційних організацій зі статутним капіталом нижче заданого
20	Вивести назви іграшок, вартість яких вище заданої
21	Вивести характеристики канцелярського приладдя як товару за заданим артикулом
22	Вивести інформацію про співробітника кафедри за номером його посвідчення

ДОДАТОК Б
Відлагодження програми в Microsoft Visual Studio

Відлагодження – це процес пошуку і виправлення помилок у програмному забезпеченні, що перешкоджають коректній роботі. Відлагодження є одним з найбільш важливих і трудомістких етапів розробки програмного забезпечення.

Відлагодження в сучасних умовах зазвичай виконується одним з таких методів:

- printf-like відлагодження (виведення інформації під час виконання програми в зовнішній потік: консоль, файл тощо);
- використання інтегрованого відлагоджувача (покрокове виконання програми під наглядом програміста з інструментами встановлення контрольних точок і переглядом значень у змінних, регістрах тощо).

Надалі мова піде про відлагодження програмного забезпечення з використанням інтегрованого відлагоджувача в Microsoft Visual Studio.

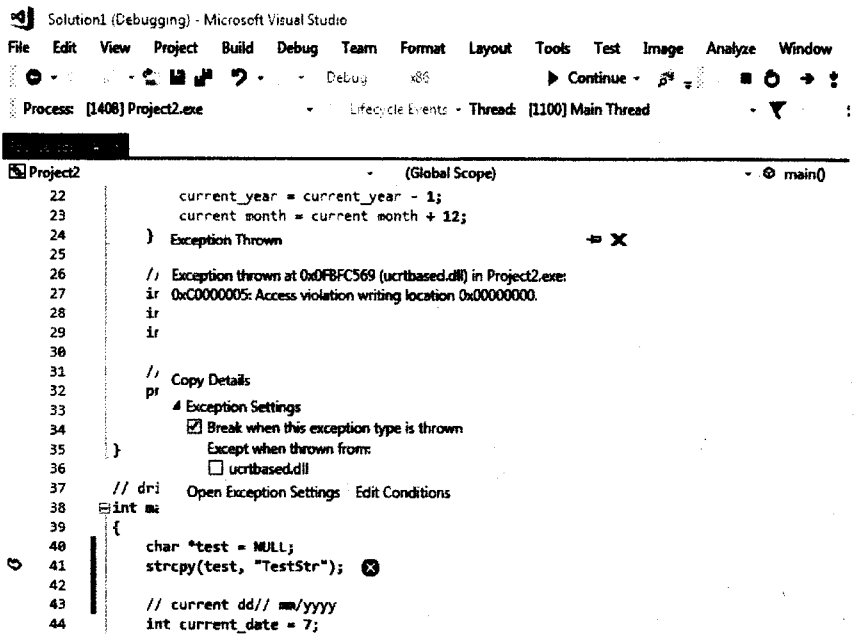
Для полегшення відлагодження програми з використанням інтегрованого відлагоджувача не слід розташовувати на одному рядку програми більше одного оператора, оскільки в процесі виконання програми відлагодження відбувається рядок за рядком. Вимога розташування на одному рядку лише одного оператора буде забезпечувати виконання не більше одного оператора щоразу в процесі відлагодження.

Існує три основних типи помилок: помилки етапу компіляції, помилки етапу виконання та логічні помилки.

Помилки етапу компіляції (синтаксичні помилки) відбуваються, коли початковий код порушує правила синтаксису мови C++. Компілятор не може скомпілювати програму, поки вона не буде містити припустимі оператори і мати правильну структуру. Коли компілятор зустрічає оператор, який він не може розпізнати, то у вікно Output середовища розробки заносяться номер рядка, в якому була знайдена помилка, і опис помилки. Двічі натиснувши на запис про помилку, текстовий курсор потрапляє з редакторі коду на рядок, в якому вона відбулася. Найбільш загальною причиною помилок етапу компіляції є помилки набору (друкарські помилки), пропущені крапки з комою, посилання на невизначені змінні, передача невірного числа (або типу) параметрів до функції і присвоювання змінним значень невірного типу. Після усунення в програмі всіх синтаксичних помилок і її успішної компіляції програма буде готова до виконання і пошуку помилок етапу виконання і логічних помилок.

Помилки етапу виконання (семантичні помилки) відбуваються, коли після компіляції певної програми під час її виконання робиться щось неприпустиме. Тобто програма містить допустимі оператори, але при

виконанні операторів щось відбувається неправильно. Наприклад, програма може намагатися виконати ділення на нуль чи намагатися відкрити для введення неіснуючий файл. Коли Microsoft Visual Studio виявляє таку помилку, то виконання програми завершується і виводиться вікно з повідомленням про тип помилки і адресою інструкції, в якій вона відбулася, наприклад:

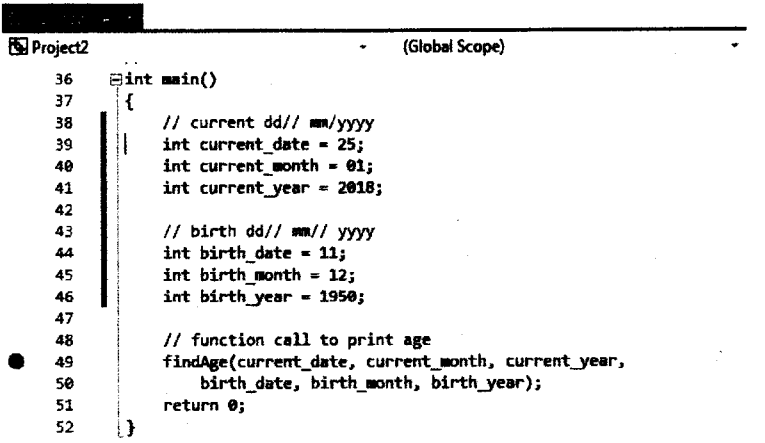


Логічні помилки – це помилки проектування та реалізації програми. Тобто всі оператори припустимі і щось роблять, але не те, що передбачалося. Ці помилки часто важко відстежити, оскільки Microsoft Visual Studio не може знайти їх автоматично, як синтаксичні та семантичні помилки. Але Microsoft Visual Studio включає в себе засоби налагодження, що допомагають знайти логічні помилки. Логічні помилки призводять до некоректного або непередбаченого значення змінних, неправильного виду графічних зображень або невиконання коду, коли це очікується.

Іноді, коли програма робить щось непередбачене, причина досить очевидна і можна швидко виправити код програми. Але інші помилки більш важко знайти, оскільки їх може викликати взаємодія різних частин програми. У цих випадках краще всього зупинити програму в заданій точці, пройти її крок за кроком і переглянути стан змінних і виразів. Таке кероване виконання – ключовий елемент відлагодження.

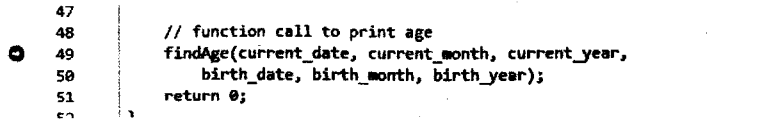
Встановлення точок переривання дозволяє зупинити виконання програми перед виконанням будь-якого оператора. Після цього можна продовжувати виконання програми або в покроковому режимі, або в безперервному режимі до наступної точки переривання.

Щоб задати точку переривання перед деяким оператором, необхідно встановити перед ним текстовий курсор і натиснути клавішу F9 або натиснути мишею на лівому бічному полі вікна редагування навпроти оператора. Точка переривання позначається у вигляді червоного круга на лівому полі вікна редагування:



Повторна дія (натискання на зазначеному крузі точки переривання або натискання F9) знімає точку переривання. У програмі може бути кілька точок переривання. Для перегляду всіх точок переривання необхідно вибрати пункт *Debug→Windows→Breakpoints* або натиснути комбінацію клавіш Ctrl+Alt+B.

Програма запускається в режимі відлагодження за допомогою команди *Debug→Start Debugging* (або натисканням клавіші F5). У результаті код програми виконується до рядка, на якому встановлена точка переривання. Потім програма зупиняється і відображає у вікні *Editor* ту частину коду, де знаходиться точка переривання, причому жовта стрілка на лівому полі вказує на рядок, який буде виконуватися на наступному кроці відлагодження.

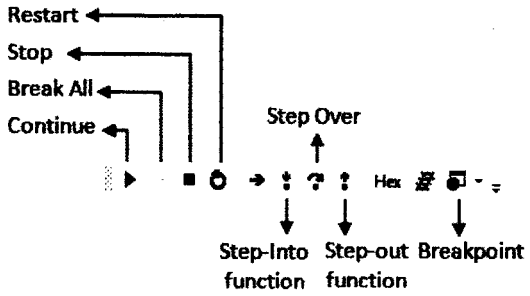


У процесі відлагодження іноді бажано почати все спочатку. Вибір команди *Debug→Stop Debugging* або натискання клавіші Shift+F5 приводять до того, що виконання по кроках або трасування програми припиняться.

Команди виконання по кроках *Step Over* (клавіша F10) і трасування *Trace Into* (клавіша F11) меню *Debug* дають можливість виконувати програму рядок за рядком. Єдина відмінність виконання по кроках і трасування полягає в тому, як вони працюють з викликами функцій.

Виконання по кроках виклик функції інтерпретує як простий оператор і після завершення підпрограми повертає управління на наступний рядок. Трасування підпрограми завантажує код цієї підпрограми і продовжує її порядкове виконання. Якщо при покроковому виконанні програми необхідно пройти через код функції, то треба натиснути клавішу F11 на операторі виклику функції в коді, а якщо внутрішня робота функції не цікавить, а цікавлять тільки результати її виконання, то треба натиснути клавішу F10. Якщо під час трасування програми в середині коду функції необхідно вийти з функції у точку її виклику, тобто продовжити відлагодження після повернення з функції, у цьому випадку треба натиснути клавіші Shift+F11.

Всі описані команди відлагодження також доступні з меню:



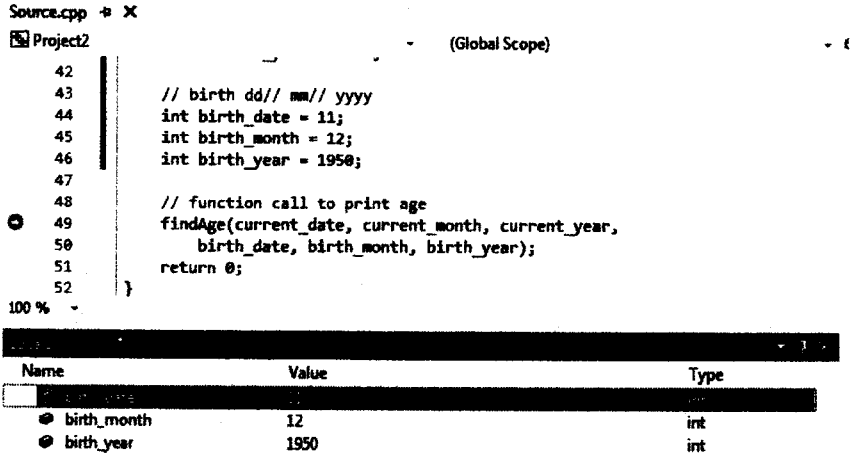
Іноді небажано виконувати по кроках всю програму тільки для того, щоб дістатися до того місця, де виникає проблема. Відлагоджувач дозволяє виконати відразу великий фрагмент програми до тієї точки, де необхідно почати виконання по кроках. Для цього необхідно встановити текстовий курсор в потрібне місце програми і натиснути клавіші Ctrl+F10. У такий чин програма буде виконана до курсору і зупиниться в даній точці, очікуючи подальших дій користувача. Ці дії можна зробити як на початку сеансу відлагодження, так і коли вже частина програми виконана по кроках. Щоб продовжити подальше виконання програми без покрокового режиму, можна натиснути F5.

Виконання програми по кроках або її трасування можуть допомогти знайти помилки в алгоритмі програми, але зазвичай бажано також знати, що відбувається на кожному кроці зі значеннями окремих змінних. Для цього

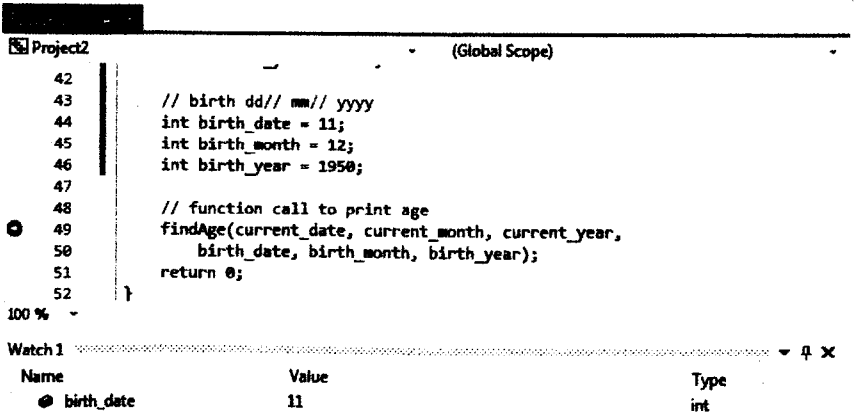
відлагоджувач має засоби перегляду значень змінних, виразів і структур даних.

Щоб дізнатися значення змінної в процесі відлагодження, можна затримати над цією змінною курсор миші. Поруч з ім'ям змінної на екрані з'явиться підказка зі значенням цієї змінної.

Крім екранної підказки, змінні зі своїми останніми значеннями відображаються у вікні *Locals*:



Крім цього, у вікні *Watch* можна задати ім'я будь-якої змінної, за значеннями якої необхідно спостерігати:



Також можна додати змінну в список перегляду, натиснувши праву кнопку миші над ім'ям змінної і вибравши пункт *Add Watch*. Крім цього, у вікні перегляду *Watch* можна додавати або видаляти елементи, можна перевіряти змінні та вирази і змінювати значення будь-яких змінних, включаючи рядки, покажчики, елементи масиву і поля записів, що дозволяє перевірити реакцію програми на різні умови.

Середовище розробки підтримує створення додаткових умов для точок переривання. Задавши точку переривання, можна вказати додаткову умову її спрацьовування, натиснувши правою кнопкою миші над рядком з точкою переривання:

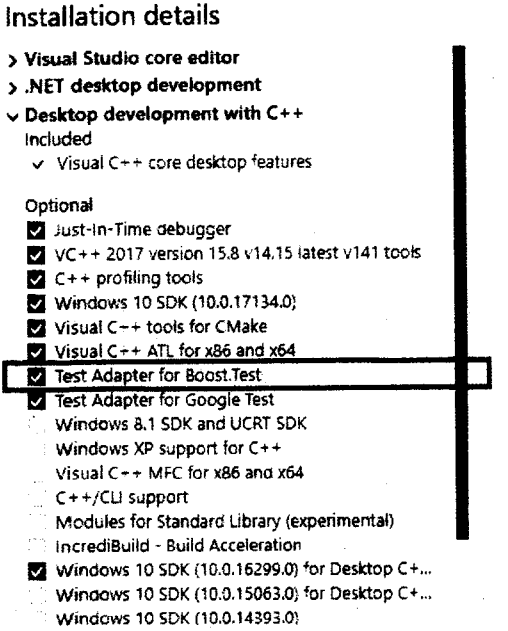
З контекстного меню можна виставити додаткові параметри для точок переривання:

- *Location* – місце зупинки (рядок і модуль для зупинки);
- Condition – перевірка умови зміни (на рівність якомусь значенню чи на зміну значення);
- *Hit Count* – перевірка на лічильник проходів;
- *Filter* – комбінація декількох умов для значень змінної;
- *When Hit* – задає дії при виникненні зупинки.

ДОДАТОК В
Модульне тестування програмного забезпечення за допомогою тестової платформи Boost::Test

Boost – одна з найпопулярніших і самих великих бібліотек для C++, а Boost::Test – це платформа для тестування, що входить в її склад, яка побудована на макросах, що надають можливість виконувати складні перевірки в модульних тестах.

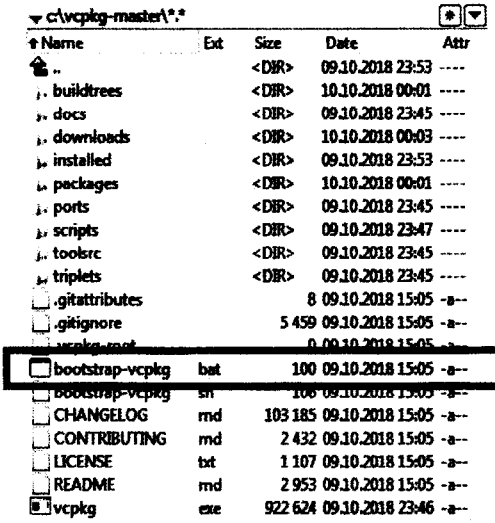
У Visual Studio 2017 версії 15.5 і більш пізніших адаптер тесту Boost::Test наявний в інтегрованому середовищі розробки як компонент, який встановлюється за замовчуванням під час інсталяції набору компонентів з розділу Desktop development with C++:



Якщо в інтегрованому середовищі розробки Visual Studio не встановлено компоненти Desktop development with C++, необхідно відкрити програму установки Visual Studio Installer і обрати Modify, потім обрати компоненти Desktop development with C++ і продовжити їх установку.

Для роботи з Boost::Test необхідно, щоб була встановлена бібліотека Boost. Для її встановлення зручно скористатися диспетчером пакетів vcpkg.

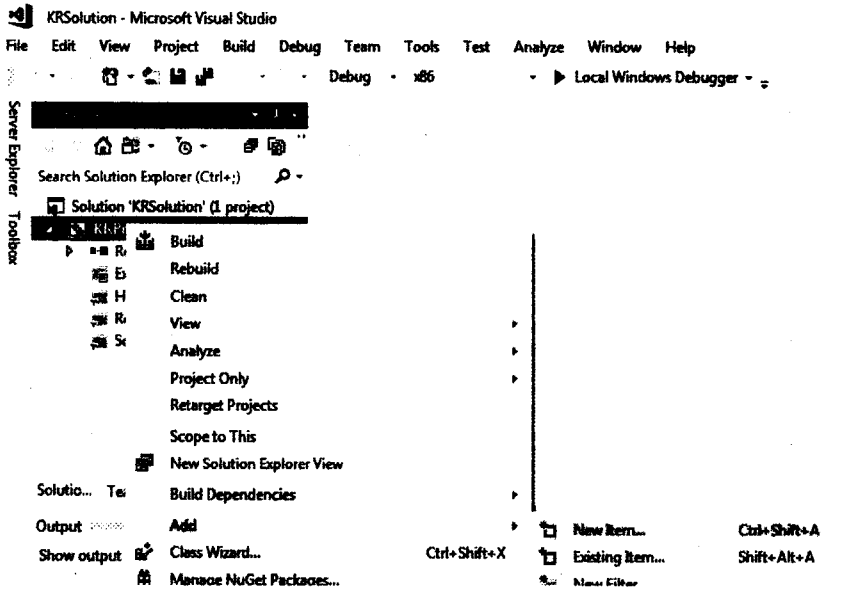
Якщо vcpkg не встановлено, скачати його можна з GitHub за посиланням https://github.com/Microsoft/vcpkg. Виконання програми завантажувача vcpkg виконується в кореневій папці через запуск bootstrap-vcpkg.bat.



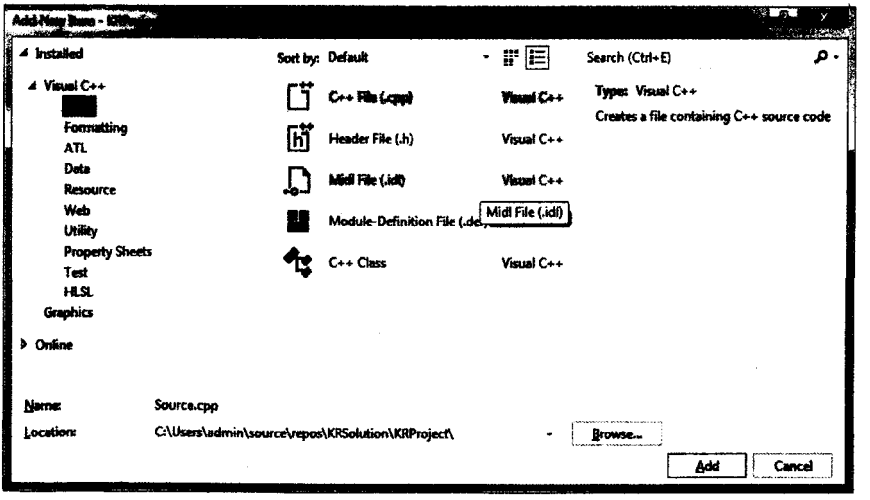
Виконання в командному рядку команди vcpkg search дозволяє дізнатися, які пакети вже встановлено:

Запуск на виконання встановлення Boost::Test виконується через команду vcpkg install boost-test. Після завершення процедури встановлення необхідно виконати команду vcpkg integrate install, щоб налаштувати в Visual Studio встановлені бібліотеки і шляхи підключення заголовних і бінарних файлів.

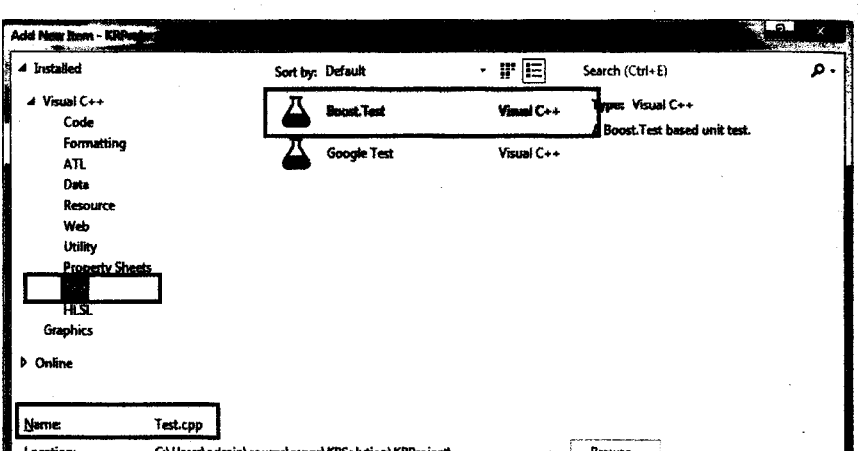
Щоб створити файл для написання коду тестів, необхідно нажати правою кнопкою миші на елементі проекту в Solution Explorer і обрати пункт Add, з якого обрати New Item:



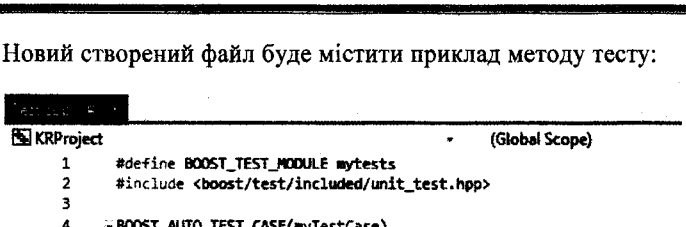
Після цього відкриється вікно додавання нового елемента:



У вікні необхідно обрати елемент Boost::Test з пункту Test і задати бажане ім'я для файлу, в якому буде створюватися код майбутнього тесту:



Новий створений файл буде містити приклад методу тесту:



Шаблон елемента використовує варіант Boost::Test з одним заголовком:

```
#include <boost/test/included/unit_test.hpp>
```

Оскільки вище було описано встановлення динамічної версії Boost::Test, для уникнення проблем з підключенням заголовних файлів у коді файлів тестів буде використовуватися підключення автономного варіанта бібліотеки:

```
#include <boost/test/unit_test.hpp>
```

Таким чином, кожен файл, який буде використовувати Boost::Test, повинен підключати зазначений заголовний файл.

Для модульних тестів також необхідно визначити точку входу, тобто функцію, з якої почнеться їх виконання. Залежно від того, як організований проект, можна або написати цю функцію самостійно, або довірити всю роботу Boost. Функція main буде додана автоматично, якщо перед підключенням unit_test.hpp оголосити константи *BOOST_TEST_MAIN* і *BOOST_TEST_DYN_LINK*:

```
#define BOOST_TEST_MAIN
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

У такому випадку виклики всіх модульних тестів проекту будуть автоматично розміщені в згенерованій функції.

Якщо код тестів розташований в файлах того самого проекту, що й основний код програми, для попередження виникнення помилок зібрання проекту необхідно забезпечити наявність лише однієї функції main. Якщо ж код тестів розташований в файлах окремого проекту, ця проблема не виникне.

Після виконання зазначеного вище в даному розділі все готово для написання і виконання модульних тестів, використовуючи платформу для тестування Boost::Test.

Набори тестів можуть бути згруповані в пакети за допомогою макросів *BOOST_AUTO_TEST_SUITE* і *BOOST_AUTO_TEST_SUITE_END*, що визначають початок і кінець тестового набору відповідно. Макрос *BOOST_AUTO_TEST_SUITE* очікує рядок, який визначає ім'я тестового пакету. Якщо тест визначено поза межами тестового пакету, то він автоматично буде доданий до головного тестового пакету, якому за замовчуванням присвоєно ім'я «Master Test Suite».

Після запуску тестування програма буде послідовно заходити в тестові пакети і виконувати вкладені тести.

Для опису тесту використовується макрос *BOOST_AUTO_TEST_CASE*, що містить його ім'я і сам код тесту.

У коді тесту використовуються спеціальні макроси, які дозволяють перевіряти відповідність отримуваних результатів очікуваним. Ці макроси умовно поділяються на три основні категорії: *BOOST_WARN*, *BOOST_CHECK* і *BOOST_REQUIRE*. Перевірки категорії *BOOST_WARN* не призводять до збільшення значення лічильника помилок і переривання виконання тесту, а лише до виводу попереджень в протокол виконання тестів. Різниця між перевірками категорій *BOOST_CHECK* і *BOOST_REQUIRE* полягає в тому, що в першому випадку виконання тесту

триває навіть у випадку порушення умови, тоді як у другому випадку помилка вважається критичною і виконання тесту переривається (в обох випадках значення лічильника помилок збільшується).

Найчастіше вживані макроси перевірки відповідності отримуваних результатів очікуваним:

- *BOOST_WARN_MESSAGE(умова, повідомлення)* – виводить текст *повідомлення*, якщо *умова* помилкова;
- *BOOST_CHECK(умова)* – повідомляє про помилку, якщо *умова* помилкова;
- *BOOST_CHECK_NO_THROW(вираз)* – повідомляє про помилку, якщо обчислення *виразу* призвело до генерації винятку;
- *BOOST_CHECK_THROW (вираз, виняток)* – повідомляє про помилку, якщо обчислення *виразу* не призвело до генерації винятку необхідного типу;
- *BOOST_CHECK_CLOSE_FRACTION(аргумент1, аргумент2, похибка)* – тест не проходить у випадку, якщо *аргумент1* не дорівнює *аргумент2* із заданою *похибкою*;
- *BOOST_REQUIRE_EQUAL(аргумент1, аргумент2)* – повідомляє про критичну помилку, якщо *аргумент1* не дорівнює *аргумент2*.

Більш повний список доступних макросів можна знайти в офіційній документації за посиланням: https://www.boost.org/doc/libs/, обравши розділ, що відноситься до платформи Boost::Test.

Як приклад створення модульних тестів, далі розглядається невелика програма, в коді якої реалізована функція порівняння двох чисел з допустимою похибкою:

```
#include <cmath>

inline bool is_close(const float a
                   , const float b
                   , const float epsilon = 0.0001)
{
    return std::fabs(a - b) < epsilon;
}
```

Для перевірки роботи зазначеної функції створюється набір тестів з назвою TestCompare.

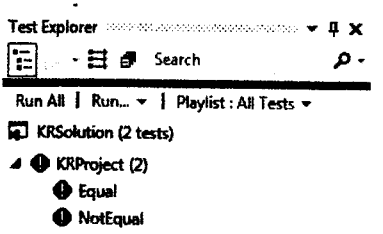
У *.cpp файлах модульних тестів необхідно додавати через директиву *#include* файли програми, щоб типи і функції були доступні коду тесту. Як правило, файли з кодом програми знаходяться в ієрархії папок на один рівень вище, тому при додаванні їх через директиву *#include* в шляху необхідно буде скористатися «../».

У *.cpp файлі модульного тесту для перевірки функції порівняння двох чисел з допустимою похибкою підключено заголовний файл Header.h, що містить код зазначеної функції, і код набору тестів TestCompare, який містить два тести Equal та NotEqual:

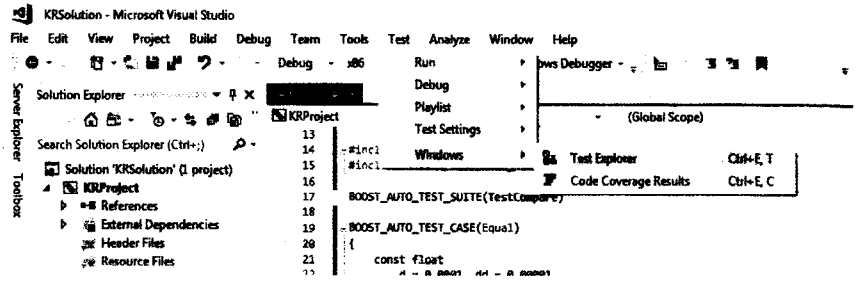
```
#include "Header.h"
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE(TestCompare)


BOOST_AUTO_TEST_CASE(Equal)
{
      const float
          d = 0.0001, dd = 0.00001,
          a = 3, b = a + d - dd;
      BOOST_REQUIRE( is_close(a, b, d) );
}

BOOST_AUTO_TEST_CASE(NotEqual)
{
      const float
          d = 0.0001, dd = 0.00001,
          a = 3, b = a + d + dd;
      BOOST_REQUIRE_EQUAL( is_close(a, b, d), false );
}

BOOST_AUTO_TEST_SUITE_END()
```

У першому тесті параметри функції еквівалентні один одному із заданою похибкою, тому в якості результату очікується true і використовується макрос *BOOST_REQUIRE*. У другому тесті параметри функції не рівні, тому використовується макрос *BOOST_REQUIRE_EQUAL(вираз, false)*.

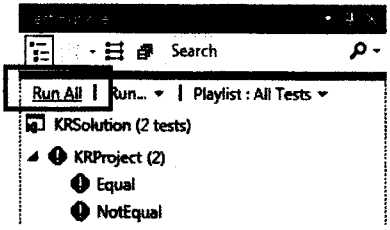
Після зборки коду проекту, який містить модульні тести, останні з'являються у вікні Test Explorer:



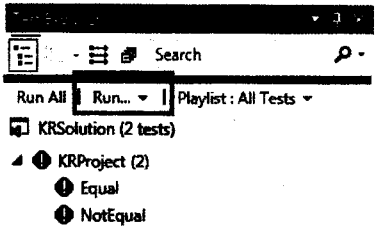
Якщо вікно Test Explorer закрито, його можна відкрити через обрання пункту Test головного меню інтегрованого середовища розробки Visual Studio, обравши пункт Windows, а потім пункт Test Explorer:



Для запуску всіх наявних модульних тестів на виконання необхідно обрати Run All у вікні Test Explorer:



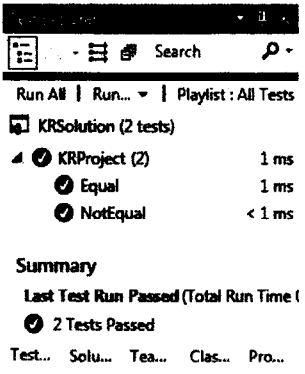
Обравши пункт Run... у вікні Test Explorer, можна задати не всі, а окремі тести на виконання:



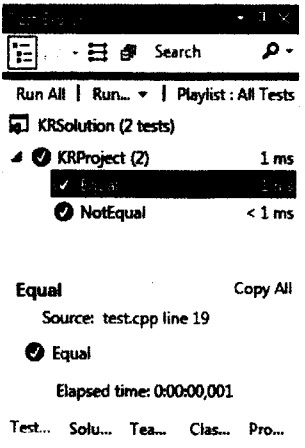
Результати виконання тестів відображаються у вікні Test Explorer за групами:

- невдалі тести (Failed Tests);
- пропущені тести (Skipped Tests);
- пройдені тести (Passed Tests);
- тести, які не запускалися (Not Run Tests).

У нижній частині вікна Test Explorer після виконання тестів буде відображатися зведений звіт тестового запуску:



Щоб переглянути детальну інформацію для окремого тесту, необхідно обрати цей тест:



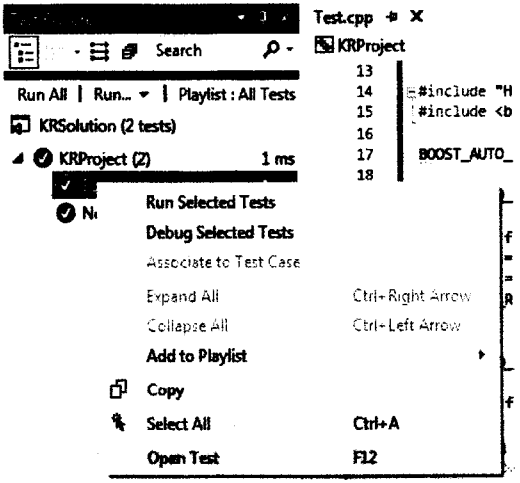
У результаті таких дій у нижній частині вікна Test Explorer буде відображуватися інформація щодо:

- імені файлу, в якому розміщено тест і номер рядка методу тесту;
- стану тесту;
- часу, витраченого на виконання методу тесту.

У випадку, якщо тест буде мати статус невдалого після виконання, в області відомостей також відображатиметься наступне:

- повідомлення, що повертає платформа модульного тестування для тесту;
- трасування стека під час збою тесту.

Щоб вивести код методу модульного тесту для перегляду в редакторі Visual Studio, необхідно обрати тест, а потім пункт Open Test із контекстного меню, яке відкриється (або нажати клавішу F12):



ДОДАТОК Г
Приклад оформлення титульної сторінки

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ БІОМЕДИЧНОЇ ІНЖЕНЕРІЇ
КАФЕДРА БІОМЕДИЧНОЇ КІБЕРНЕТИКИ

КУРСОВА РОБОТА

з дисципліни «Об'єктно-орієнтоване програмування»

Варіант №2

Керівник :
ст. викладач каф. БМК,
к.т.н. Петров П.П.

Виконав:
студент гр. БС-61, ФБМІ
Іванов І.І.
залікова книжка № БС-6114

Допущено до захисту
"___" _____ 2018 _____
підпис

Захищено з оцінкою

_____ _____
оцінка підпис
"___" _____ 2018

Київ-2018

ДОДАТОК Д
Приклад оформлення завдання на курсову роботу та календарного плану

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет *біомедичної інженерії*
(назва факультету, інституту)
Кафедра *біомедичної кібернетики*
(назва кафедри)
Дисципліна *«Об'єктно-орієнтоване програмування»*
(назва)
Курс *2* Група *БС-61* Семестр *4*

ЗАВДАННЯ
на курсову роботу студента

Іванова Іван Івановича
(прізвище, ім'я, по батькові)

1. Тема роботи: *Розробка програмного забезпечення з використанням об'єктно-орієнтованого підходу.*

2. Строк здачі студентом закінченого проекту (роботи) *18.05.2018 р.*

3. Вихідні дані до проекту (роботи): *Варіант №2*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці): *1. Визначення класів та програмування меню користувача (ієрархія класів програми: абстрактний клас – особа, класи-нащадки – студент, аспірант). 2. Створення об'єктів та використання контейнерів (тип контейнера – одно-зв'язний кільцевий список). 3. Організація роботи з даними через файл. 4. Пошук даних у контейнері (запит для пошуку – прізвища всіх студентів, які навчаються на заданому курсі).*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): *діаграми класів, послідовності та об'єктів в нотації UML.*

6. Дата видачі завдання: *12.02.2018 р.*

КАЛЕНДАРНИЙ ПЛАН

п/п	Назва етапів курсового проекту (роботи) та питань, які мають бути розроблені відповідно до завдання	Термін виконання етапу	Позначки керівника про виконання завдань
1.	Отримання завдання на курсову роботу	12.02.2018	
2.	Огляд технічної літератури за темою роботи	19.02.2018	
3.	Розробка першої частини курсової роботи (визначення класів та програмування меню користувача)	05.03.2018	
4.	Перший контроль за процесом виконання курсової роботи, консультація у викладача	05.03.2018	
5.	Розробка другої частини курсової роботи (створення об'єктів та використання контейнерів)	19.03.2018	
6.	Розробка третьої частини курсової роботи (робота з даними через файл)	09.04.2018	
7.	Другий контроль за процесом виконання курсової роботи, консультація у викладача	09.04.2018	
8.	Розробка четвертої частини курсової роботи (пошук даних у контейнері)	30.04.2018	
9.	Оформлення пояснювальної записки	18.05.2018	
10.	Захист курсової роботи	28.05.2018	

Студент _____
(підпис)

Керівник _____ Петров Петро Петрович
(підпис) (прізвище,ім'я по батькові)

«____» _____ 20___ р.

ДОДАТОК Е
Приклади оформлення анотацій різними мовами

Приклад оформлення анотації українською мовою:

Анотація

Іванов І.І. Розробка програмного забезпечення з використанням об'єктно-орієнтованого підходу.

Курсова робота з дисципліни «Об'єктно-орієнтоване програмування» присвячена питанню створення ієрархії класів, застосування спадкування та поліморфізму, розробки контейнера для зберігання даних множини об'єктів, які створюються користувачем. У курсовій роботі було виконано визначення класів (ієрархія класів програми: базовий клас – особа, класи-нащадки – студент, аспірант), розроблено меню користувача, створено множину об'єктів та розроблено контейнер для її зберігання (тип контейнера – однозв'язний кільцевий список), виконано серіалізацію даних елементів контейнера у файл та створення вмісту контейнера через десеріалізацію даних файлу, реалізовано пошук даних у контейнері (запит для пошуку – прізвища всіх студентів, які навчаються на заданому курсі).

Структура і обсяг роботи. Курсова робота складається із вступу, трьох розділів, висновків, списку використаної літератури з 3 джерел і 2 додатків. Загальний обсяг курсової роботи становить 45 сторінок, основного тексту (без додатків) – 25 сторінок, ілюстрацій – 34, таблиць – 38.

Приклад оформлення анотації англійською мовою:

Annotation

I. Ivanov. Software development using object oriented approach.

Coursework on the Object Oriented Programming course is devoted to the issue of creating a hierarchy of classes, applying inheritance and polymorphism, developing storage container for the collection of objects created by the user. In the coursework class definitions were performed (hierarchy of program classes: base class – person, derived classes – student, postgraduate), user menu was designed, set of objects was created and its storage container was developed (type of container – circular singly linked list), serialization of container data was done and container elements were created via deserialization of file data, data search in the container was performed (search query – surnames of students who are studying at a given course).

The structure and the amount of work. Coursework consists of an introduction, three partitions, conclusions, list of used literature with 3 references, and 2 applications. The total volume of coursework is 45 pages, main text (without applications) – 25 pages, illustrations – 34, tables – 38.

Приклад оформлення анотації російською мовою:

Аннотация

Иванов И.И. Разработка программного обеспечения с использованием объектно-ориентированного подхода.

Курсовая работа по дисциплине «Объектно-ориентированное программирование» посвящена вопросу создания иерархии классов, применения наследования и полиморфизма, разработки контейнера для хранения данных коллекции объектов, которые создаются пользователем. В курсовой работе было выполнено определение классов (иерархия классов программы: базовый класс – лицо, классы-потомки – студент, аспирант), разработано меню пользователя, создано множество объектов и разработан контейнер для их хранения (тип контейнера – односвязный кольцевой список), выполнены сериализация данных элементов контейнера в файл и создание содержимого контейнера путем десериализации данных файла, реализован поиск данных в контейнере (запрос поиска – фамилии всех студентов, обучающихся на заданном курсе).

Структура и объем работы. Курсовая работа состоит из введения, трех разделов, выводов, списка использованной литературы из 3 источников и 1 приложения. Общий объем курсовой работы составляет 45 страниц, основного текста (без приложений) – 25 страниц, иллюстраций – 34, таблиц – 38.

ДОДАТОК Ж
Приклад оформлення змісту

ДОДАТОК З
Оформлення коду

Стандарт оформлення коду (стандарт написання коду) зазвичай приймається та використовується групою розробників програмного забезпечення для єдиного стилю оформлення коду, над яким йде спільна праця.

Зазвичай метою стандарту є досягнення такого стану, коли програміст достатньої кваліфікації міг би дати висновки про функцію, яку виконує конкретна ділянка коду, а в ідеалі – також визначити його коректність, вивчивши тільки цю ділянку коду, або мінімально вивчивши інші частини програми. Іншими словами, сенс коду повинен буди зрозумілим з самого коду без необхідності вивчати контекст. Тому стандарти написання коду будуються так, щоб шляхом визначеного візуального оформлення елементів програми збільшити інформативність коду для людини. Тобто прийняття та використання стандарту має спрощувати сприйняття програмного коду людиною, мінімізувати навантаження на пам'ять та зір людини при читанні програми.

Зразком для стандарту написання коду може стати набір угод, прийнятих в якій-небудь поширеній печатній праці з мови програмування (наприклад, стандарт написання коду мовою С, що отримав коротку назву K&R, виходить з класичного описання С його творцями – Керніганом та Рітчі), поширена бібліотека або API (так, на поширення угорської нотації вплинув той факт, що її використовували в MS-DOS та Windows API, а більшість стандартів для Delphi використовують манеру написання коду бібліотеки VCL). Рідше розробник мови програмування випускає детальні рекомендації з написання коду на ній; випущені, наприклад, стандарти для мови C# від Microsoft та для мови Java від Sun. Запропонований розробником та прийнятий в загально відомих джерелах стандарт написання коду може бути доповнений та уточнений у корпоративних стандартах.

Стандарт дуже залежить від використовуваної мови програмування. Наприклад, стандарт оформлення коду для С++ буде сильно відрізнятися від стандарту для мови BASIC.

Зазвичай стандарт оформлення коду визначає:

- засоби вибору значень та використовуваний регістр символів для імен змінних та інших ідентифікаторів;
- запис типу змінної в її ідентифікаторі;
- регістр символів (нижній, верхній, «верблюжий», «верблюжий» з малої букви), використання знаку підкреслення для розділу слів;
- стиль відступів при оформленні логічних блоків (використання табуляції, ширину відступів);
- спосіб розстановки дужок, що обмежують логічні блоки;
- використання пробілів у логічних та арифметичних виразах;
- стиль коментарів.

Поза стандартом існують правила, які, наприклад, встановлюють умову відсутності в коді магічних чисел або обмеження розміру коду по горизонталі та вертикалі.

Під час виконання курсової роботи слід дотримуватися наступних вимог щодо оформлення коду.

- Код програм повинен містити коментарі, але вони не повинні підтверджувати очевидне.
  Приклад непотрібних коментарів:

```
x++; // збільшення x
```

- На початку файлу необхідно вставляти коментар, який вказує автора, а також, що робиться у файлі і якісь особливості використання даного файлу.

- Коментар краще розташовувати в багаторядкових блоках, вирівнюючи початок і кінець коментаря за вертикаллю:

```
/* Перший рядок
 * Другий рядок
 * Третій рядок */
```

або текст коментаря вирівнювати за стовпцями:

```
int a;              // поточний покажчик символу в рядку
char str [50];      // результуючий рядок
float b;            // опис, що робить змінна b
```

- Не треба розміщувати текст коментаря між ім'ям функції і відкриттям дужок тіла функції:

```
void function(int a)
// не пишіть коментар тут
{
      ...
}
```

Якщо необхідно розмістити коментар до функції, можна це зробити наступним чином:

```
// можна помістити коментар тут
void function(int a)
{
      /* Або тут,
       * з таким же відступом, що і для коду
       * /

      ...
}
```

- Текст програми повинен поміщатися на екрані, оскільки це зручно для редагування. Коли рядок стає довшим, ніж 100 символів, треба розділити його на два, зробивши переніс на новий рядок, і продовжити писати:

```
int result = reallyLongFunctionOne() + reallyLongFunctionTwo()
           + reallyLongFunctionThree() + reallyLongFunctionFour();

int result2 = reallyLongFunction(parameterOne, parameterTwo
                                 , parameterThree, parameterFour
                                 , parameterFive, parameterSix);
```

- Не слід розміщувати в одному рядку більше одного оператора:

```
// поганий підхід до написання коду
int x = 3, y = 7;  double z = 4.25;  x++;
if (a == b) { foo(); bar(); }


// добрий підхід до написання коду
int x = 3;
int y = 7;
double z = 4.25;

x++;

if (a == b)
{
      foo();
      bar();
}
```

- Слід використовувати пунктирну лінію для візуального розділення функцій перед кожним визначенням функції:

```
// ------------------------------------------------------------------
// коментар до функції
void function(int a)
{
      ...
}
```

- Необхідно залишати порожні лінії між функціями і між групами виразів:

```
// ------------------------------------------------------------------
void foo()
{
    ...
}

// ------------------------------------------------------------------
void bar()
{
    ...
}
```

- Позначати кінець довгого складеного оператора треба за допомогою коментаря, наприкінці такого блоку краще повністю описувати керуючий оператор:

```
for ( int i = 0; i < 10; i++ )
{
      ...

      while ( a > b )
      {
            ...
      } // end of "while ( a > b )"

      ...
} // end of "for ( int i = 0; i < 10; i++ )"
```

- Необхідно робити відступи, щоб блоки в коді були видимі. Для відступів треба використовувати табуляцію, а не пробіли. Якщо блок потребує використання фігурних дужок, то їх слід вирівнювати за лівим краєм:

```
int DelElement(int pos)
{
      if( pos<0 )
      {
            Errors(3);
            return 1;
      }
      if( pos==index[active] )
            Data[active][--index[active]] = 0;
      else
            while( pos<index[active] )
            {
                  Data[active][pos++] = Data[active][pos+1];
                  pos++;
            }
      return 0;
}
```

- Імена ідентифікаторів повинні бути простими, які описують дії функцій, аргументів чи змінних. Краще, якщо це будуть англійські прості слова. Не треба використовувати один символ для назв, за винятком імен ітераторів.

- Якщо змінна використовується лише всередині певного блоку, то необхідно зробити її локальною, оголошуючи в тому ж блоці коду.

- Для змінних необхідно вибирати відповідні типи даних. Якщо змінна містить лише цілі числа, то її необхідно визначити як int, а не double.

- Якщо певна константа часто використовується в коді, то її треба позначити як const і завжди посилатися на дану константу, а не на її значення:

```
const int VOTING_AGE = 18;
```

- Слід намагатися уникати використання виразів break або continue. Використання цих операторів можливе за умови абсолютної необхідності.

- Якщо один і той же код використовується двічі або більше, то необхідно знайти спосіб видалити зайвий код, щоб він не повторювався. Наприклад, його можна помістити в допоміжну функцію. Якщо повторюваний код схожий, але не зовсім, то слід зробити допоміжну функцію, яка буде приймати параметри:

```
// поганий підхід до написання коду
foo();
x = 10;
y++;

...

foo();
x = 15;
y++;


// добрий підхід до написання коду
helper(10);
helper(15);

...

// ---------------------------------------------------------------------
void helper(int newX)
{
    foo();
    x = newX;
    y++;
}
```

- Визначення класів слід розмістити у файлах *.h, а визначення методів класу слід поміщати у файлах *.cpp.

- Для визначення користувацького типу завжди необхідно використовувати class. Винятком може бути лише дуже маленький і простий тип даних, для якого потрібні лише кілька змінних.

- Не треба створювати непотрібні поля в класах: поля використовуються для того, щоб зберігати важливі дані про об'єкти, а не тимчасові значення, які використовуються в програмі один раз.

- Необхідно використовувати інкапсуляцію, зробивши будь-які поля даних у класі недоступними для змін ззовні (private або protected).

- Якщо в класі необхідно мати метод, який буде викликатися тільки з методів цього самого класу, то такий метод необхідно зробити недоступним для викликів ззовні (private або protected).

- Необхідно використовувати списки ініціалізації, а не операції привласнення для ініціалізації змінних-членів класу в конструкторах:

```
// Values.h
class Values
{
private:
      int m_value1;
      double m_value2;
      char m_value3;
public:
      Values(int value1, double value2, char value3);
      ...
};

// Values.cpp
Value::Values(int value1, double value2, char value3)
        : m_value1(value1), m_value2(value2), m_value3(value3)
{
      ...
}
```

- Якщо певний метод класу не змінює стан об'єкта, для якого він викликається, то такий метод треба оголосити як const-метод:

```
class Student
{
private:
      int ID;
      double GPA;
public:
      int getID() const;
      double getGPA(int year) const;
      void ShowToScreen() const;
      ...
};
```

- Якщо об'єкт передається до функції, код якої не змінить стану об'єкта, то його необхідно передавати як const-посилання:

```
// поганий підхід до написання коду
void display(BankAccount account)
{
    ...
}

// добрий підхід до написання коду
void display(const BankAccount &account)
{
    ...
}
```

- Файли оголошення класів .h необхідно розміщувати в блоках препроцесорів #ifndef / #define / #endif, щоб уникнути множинних оголошень одного класу:

```
// Point.h
#ifndef _point_h
#define _point_h

class Point
{
public:
    Point(int x, int y);
    int getX() const;
    int getY() const;
    void translate(int dx, int dy);
private:
    int m_x;
    int m_y;
};

#endif


// Point.cpp
#include "Point.h"

// ------------------------------------------------------------------------
Point::Point(int x, int y)
{
    m_x = x;
    m_y = y;
}

// ------------------------------------------------------------------------
void Point::translate(int dx, int dy)
{
    m_x += dx;
    m_y += dy;
}

...
```

- C ++ має функцію виходу exit(), яка призводить до негайного виходу з всієї програми. Використовувати цю функцію не можна, оскільки програма завжди повинна закінчуватися відповідно до логіки, досягнувши кінця головної функції main().

Не слід надмірно піклуватися про підвищення ефективності коду шляхом зниження його читабельності. Розробка окремих функцій не є проблемою, оскільки сучасні компілятори здатні перетворити на *inline* все, що потрібно, так само як і об'єднувати різні змінні в один регістр. Якщо ж в якомусь місці заради низкорівневої оптимізації дійсно потрібно погіршити читабельність, необхідно забезпечити цей фрагмент коду достатніми коментарями з приводу того, що ж відбувається в коді, і найголовніше – чому такий трюк знадобився.

Не слід забувати, що зробити код таким, щоб він був зрозумілий для компілятора, простіше, ніж зробити його зрозумілим для людини. Зазвичай добре написаний код живе довго, а отже, такий код буде прочитаний, підправлений, зрозумілий і пояснений багато разів.

Розуміння чужого коду набагато складніше, ніж написання нового з нуля, тому не виникає сумнівів у важливості інвестування у зрозумілість коду (якщо, звичайно, передбачається, що код буде в подальшому використовуватися).

ДОДАТОК Н
Схеми складання аркушів різних форматів

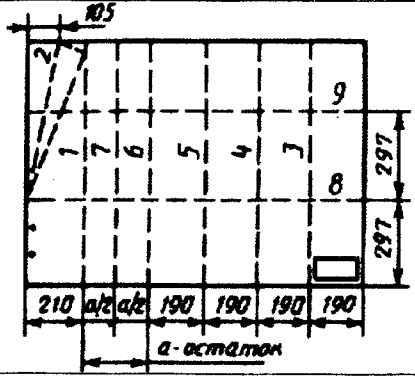
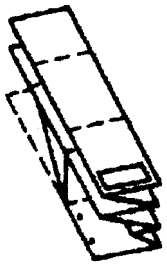
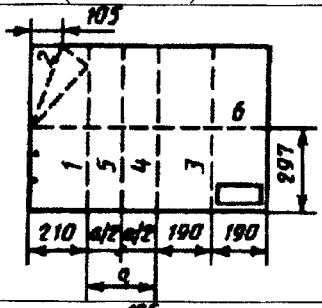
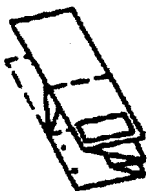
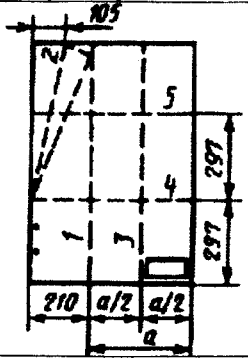
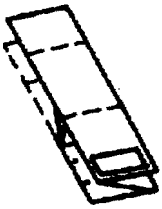
Схема складання	Поздовжнє складання	Поперечне складання
Формат А0 (841х1189 мм)		
		
Формат А1 (594х841мм)		
		
		

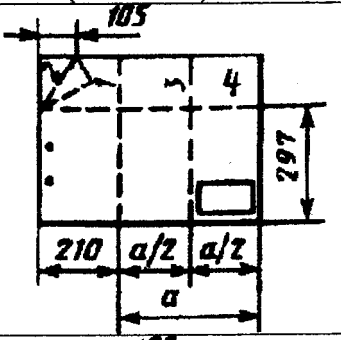
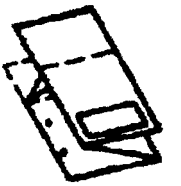
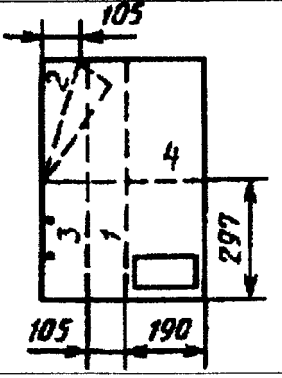
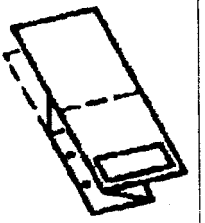
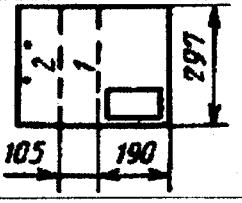
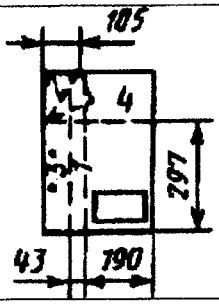
Схема складання	Поздовжнє складання	Поперечне складання
Формат А2 (420х594 мм)		
		
		
Формат А3 (297х420 мм)		
		

Схема складання	Поздовжнє складання	Поперечне складання
		

ДОДАТОК К
Оформлення списку використаних джерел

Надалі наведені приклади оформлення бібліографічних описів джерел згідно з національним стандартом ДСТУ ГОСТ 7.1:2006 «Система стандартів з інформації, бібліотечної та видавничої справи. Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання», запровадженим у дію в Україні з 01.07.2007, та національним стандартом ДСТУ 3582:2013 «Інформація та документація. Бібліографічний опис. Скорочення слів і словосполучень українською мовою. Загальні вимоги та правила (ISO 4:1984, NEQ; ISO 832:1994, NEQ)», запровадженим у дію в Україні з 22.08.2013.

Характеристика джерела	**Приклади оформлення**
Книга з одним автором	Іванов І. І. Основи програмування: навч. посіб. для студ. вищ. навч. закл. / І. І. Іванов. – К. : Каравела, 2015. – 345 с.
Книга з двома авторами	Петров П. П. Об'єктно-орієнтоване програмування мовою C++ / П. П. Петров, І. І. Іванов. – К. : Каравела, 2016. – 234 с.
Книга з трьома авторами	Вайт В. Об'єктно-орієнтований підхід до розробки сучасного програмного забезпечення / В. Вайт, Г. Дж. Грін, Р. Ред ; пер. з англ. І. І. Іванов. – Тернопіль : Сайнс Букс, 2017. – 265 с.
Книга з чотирма авторами	Архітектура складних систем : навч. посіб. для студ. вищ. навч. закл. / [І. І. Іванов, П. П. Петров, С. С. Сидоров, В. В. Васильєв]. – Луцьк : Наука, 2018. – 250 с.
Книга з п'ятьма або більше авторами	Об'єктно-орієнтований аналіз та програмування / [І. І. Іванов, П. П. Петров, С. С. Сидоров та ін.]. – Луцьк : Наука, 2010. – 500 с.
Книга за редакцією	Об'єктно-орієнтоване програмування : підручник для студ. вищ. навч. закл. / за ред. П. П. Петрова. – Тернопіль : Центр навчальної літератури, 2015. – 700 с.
Книги за укладачами	Розробка сучасного програмного забезпечення / уклад.: П. П. Петров, І. І. Іванов. – К. : Каравела, 2015. – 502 с.
Перевидані книги	Іванов І. І. Об'єктно-орієнтоване програмування : навч. посіб. для студ. вищ. навч. закл. / І. І. Іванов. – [2-е вид.]. – К. : Каравела, 2018. – 350 с. – (Серія «Науково-технічна освіта»).
	Петров П. П. Архітектура складних систем / П. П. Петров. – 2-е вид., перероб. і доп. – К. : Каравела, 2015. – 320 с.

Характеристика джерела	Приклади оформлення
Багатотомний документ	Васильєв В. В. Теорія ймовірностей і математична статистика. Ч. 1 / В. В. Васильєв, І. В. Васильєв. – К. : КПІ ім. Ігоря Сікорського, Вид-во «Політехніка», 2006. – 225 с.
Іноземні видання	Bigs B. Modern approaches for software architecture / B. Bigs, J. J. Jonn. – Cambridge : Cambridge University Press, 2014. – 254 p.
Стаття з одним автором	Іванов І. І. Проектування системи електронних бібліотек наукових і навчальних закладів / І. І. Іванов // Інформаційні технології. – 2015. – № 6. – С. 40-47.
Стаття з двома авторами	Іванов І. І. Сучасний ринок праці / І. І. Іванов, П. П. Петров // Оцінка плагіату програмного коду на основі використання систем контролю версій. – 2010. – № 12. – С. 122-128.
Матеріали конференцій, з'їздів	Викладання понять спадкування і поліморфізму в об'єктно-орієнтованому програмуванні : зб. текстів виступів на міжвуз. наук.-практ. конф. «Інформаційні технології та комп'ютерна інженерія 2015» / Ін-т статистики, обліку та аудиту. – К. : ІСОА, 2015. – 147 с.
Стандарти	Графічні символи, що їх використовують на устаткуванні. Покажчик та огляд (ISO 7000:2004, IDT) : ДСТУ ISO 7000:2004. – [Чинний від 2006-01-01]. – К. : Держспоживстандарт України, 2006. – IV, 231 с. – (Національний стандарт України).
	Інформація та документація. Бібліографічний опис. Скорочення слів і словосполучень українською мовою. Загальні вимоги та правила : (ISO 4:1984, NEQ ; ISO 832:1994, NEQ). ДСТУ 3582:2013. – [Чин. від 2014-01-01]. – К. : Мінекономрозвитку України, 2014. – 14 с. – (Національний стандарт України).
Законодавчі та нормативні документи	Про зайнятість населення : Закон України від 1 березня 1991 р. № 803 – XII // Законодавство України про працю : зб. нормат.-прав. актів. – К. : Парлам. вид-во, 2006. – С. 15-27.
Словники	Інформатика: словник-довідник/ [авт.-уклад. Іванов І. І.]. – Херсон : Наука, 2016. – 75 с.
Дисертації	Петров П. П. Проектування ефективних розподілених систем з використанням об'єктно-орієнтованого підходу : дис. ... доктора фіз.-мат. наук : 01.05.03 / Петров Петро Петрович ; Херсон. нац. ун-т. ; наук. кер. Іванов І. І. – Херсон, 2015. – 146 с.

Характеристика джерела	Приклади оформлення
Автореферати дисертацій	Петров П. П. Проектування ефективних розподілених систем з використанням об'єктно-орієнтованого підходу : автореф. дис. на здобуття наук. ступеня канд. фіз.-мат. наук : спец. 01.05.03. «Математичне та програмне забезпечення обчислювальних машин і систем» / П. П. Петров. – Херсон, 2015. – 23 с.
Авторські свідоцтва	А. с. 1747944 СССР, МКИ[4]G01K 5/56, 7/32. Устройство для измерения температуры / В. А. Воронин, Е. П. Красноженов, Р. И. Байцар, А. В. Родионов, А. Н. Жирков, Н. Л. Маковский. – №478566/10 ; заявл. 23.01.90 ; опубл. 15.07.92, Бюл. №26.
Патенти	Пат. 75207 Україна, МПК H 02 J 3/14, H 02 J 3/28. Спосіб регулювання навантаження трансформатора в мережевих трансформаторних підстанціях / В. А. Маляренко, І. Д. Колотило, І. Є. Щербак (Україна) ; Харків. нац. акад. міськ. госп-ва. – No 201205527 ; заявл. 07.05.12 ; опубл. 26.11.12, Бюл. No 22.
Віддалений електронний ресурс	Про вищу освіту : Закон України від 01.07.2014 р. № 1556–VII [Електронний ресурс] // Верховна Рада України. – Офіц. веб-сайт. – Режим доступу: http://zakon.rada.gov.ua/laws/show/2984–14, вільний. – Дата звернення : 31.08.2018. – Назва з екрана.
	Загальна характеристика програмних продуктів, поняття і класифікація [Електронний ресурс] // Всеукраїнський портал учнівських і студентських рефератів. – Режим доступу : http://referat.at.ua, вільний. – Дата звернення : 31.08.2018. – Назва з екрана.
Локальний електронний ресурс	Архітектура складних систем [Електронний ресурс] / за ред. О. О. Ольженкова. – К. : CD-вид-во «Інфодиск», 2014. – 1 електрон, опт. диск (CD-ROM). – Систем. вимоги: Pentium; 32 Mb RAM; Windows 95, 98, 2000, XP, 7; MS Word 97 — 2010. – Назва з контейнера.
	Освітні інновації у вищих навчальних закладах України [Електронний ресурс] / МОН України, Науково-методичний центр вищої освіти. – Електронні дані і програми. – [Київ] : [б. в.], [2015]. – 1 електрон. опт. диск (CD-ROM). – Систем. вимоги: наявність інтернет-браузера, що підтримує CSS і кодову сторінку 1251. – Назва з контейнера.

Примітки:

– усі умовні розділові знаки, котрі відділяють окремі зони чи елементи у межах зон бібліографічного опису (за винятком граматичної

пунктуації і у випадках назв видань) відділяються проміжками з двох сторін;

- якщо видання має лише одного автора, його прізвище все одно повторюється в області відповідальності після скісної лінії;
- дані, котрі взяті не з титульного аркуша книжкового видання, беруться у квадратні дужки (таким чином, у квадратних дужках потрібно писати відомості про упорядників, авторів, вид видання, котрі наведені на звороті титульного аркуша), у квадратні дужки також береться вся інформація, яка взята не безпосередньо з видання, а встановлена самостійно на основі аналізу видання;
- усі частини бібліографічного опису, крім перших слів нових зон бібліографічного опису та власних назв, пишуться з малої літери (таким чином, додаткові відомості про назву, такі як підручник або посібник, інформацію про відповідальність, таку як автор-упорядник або редактор, потрібно писати з малої літери);
- міжнародний стандартний книжковий номер (ISBN), ціна, відомості про тираж при складанні бібліографічних описів не вказуються.

ДЛЯ НОТАТОК

Навчальне видання

Алхімова Світлана Миколаївна

Об'єктно-орієнтоване програмування

Частина 2

Об'єктно-орієнований підхід до розробки програмного забезпечення

Підручник

В авторській редакції
Комп'ютерна верстка авторська

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Свідоцтво про державну реєстрацію: серія ДК № 5354 від 25.05.2017 р.
просп. Перемоги, 37,
м. Київ, 03056

Темплан 2018 р., поз. 1-1-015

Підп. до друку 18.01.2019. Формат 60×84$^{1}/_{16}$. Папір офс. Гарнітура Times.
Спосіб друку – ризографічний. Ум. друк. арк. 11,16. Обл.-вид. арк. 18,56. Наклад 150 пр.
Зам. № 19-001.

Видавництво «Політехніка» КПІ ім. Ігоря Сікорського
вул. Політехнічна, 14, корп. 15
м. Київ, 03056
тел. (044) 204-81-78

КПІ ім. Ігоря Сікорського
Видавництво «Політехніка»