О. Г. Трофименко, С. Ю. Манаков, Д. Г. Ларін

Основи програмної інженерії

Навчально-методичний посібник
для підготовки здобувачів вищої освіти
галузі знань 12 «Інформаційні технології»

Одеса
Фенікс
2022

УДК 004.4 (076)

Рекомендовано Навчально-методичною радою
Національного університету «Одеська юридична академія»,
Протокол № 3 від 17 травня 2022 р.

Рецензенти:

Малаксіано М. О. – доктор технічних наук, професор, завідувач кафедри «Технічна кібернетика й інформаційні технології ім. проф. Р.В. Меркта» Одеського національного морського університету;

Панченко Б. Є. – доктор фізико-математичних наук, с.н.с, в.о. завідувача кафедри «Інженерія програмного забезпечення» Державного університету інтелектуальних технологій і зв'язку

Трофименко О. Г.¸Манаков С. Ю., Ларін Д. Г.

Т 76 Основи програмної інженерії : навч.-метод. посібник [Електронне видання] / О. Г. Трофименко, С. Ю. Манаков, Д. Г. Ларін. – Одеса : Фенікс, 2022. – 194 с.

ISBN 978-966-928-808-0

Розглянуто основні засоби програмної інженерії щодо розробки моделей засобами UML, роботи в системі контролю версій Git, планування програмного проєкту, декомпозиції і тестування програм, вивченню методів і механізмів DevOps і DevSecOps. Містить завдання з індивідуальними варіантами до лабораторних занять, які виконуються у комп’ютерному класі, та самостійних робіт у межах навчальної дисципліни. Призначено для студентів з метою закріплення лекційного матеріалу і підготовки до лабораторних занять та самостійних робіт з дисципліни «Основи програмної інженерії».

Для підготовки здобувачів вищої освіти галузі знань 12 «Інформаційні технології».

УДК 004.4 (076)

ISBN 978-966-928-808-0

© О. Г. Трофименко, С. Ю. Манаков, Д. Г. Ларін, 2022

ЗМІСТ

Передмова 6

Структура дисципліни 7

Лабораторна робота 1 Система контролю версій Git 9
Теоретичні відомості 9
Контрольні запитання для самоконтролю 10
Лабораторне завдання 11

Самостійна робота 1 Контроль версій ПЗ засобами сервісу Replit 15
Теоретичні відомості 15
Контрольні запитання 19
Завдання 20

Лабораторна робота 2 Створення віддаленого репозиторія GIT 21
Теоретичні відомості 21
Контрольні запитання для самоконтролю 21
Лабораторне завдання 22

Лабораторна робота 3 Робота з віддаленим репозиторієм GIT 24
Теоретичні відомості 24
Контрольні запитання для самоконтролю 25
Лабораторне завдання 25

Самостійна робота 2 Проєктування програмного забезпечення 27
Теоретичні відомості 27
Контрольні запитання 47
Завдання 47

Самостійна робота 3 DevOps 48
Теоретичні відомості 48
Контрольні запитання 51
Завдання 51

Самостійна робота 4 DevSecOps 52
Теоретичні відомості 52
Контрольні запитання 57
Завдання 57

Лабораторна робота 4 Діаграми UML.
Ознайомлення з інтерфейсом StarUML 58
Теоретичні відомості 58
Контрольні запитання для самоконтролю 66
Лабораторне завдання 67

Лабораторна робота 5 Створення діаграми варіантів використання (Use Cases Diagrams) 80
Теоретичні відомості 80
Контрольні запитання для самоконтролю 90
Лабораторне завдання 90

Лабораторна робота 6 Створення діаграми послідовностей (Sequence Diagrams) 93
Теоретичні відомості 93
Контрольні запитання для самоконтролю 105
Лабораторне завдання 105

Лабораторна робота 7 Створення діаграм діяльності (Activity Diagrams) 107
Теоретичні відомості 107
Контрольні запитання для самоконтролю 116
Лабораторне завдання 116

Лабораторна робота 8 Створення діаграми класів (Class Diagrams) 117
Теоретичні відомості 117
Контрольні запитання для самоконтролю 125
Лабораторне завдання 125

Самостійна робота 5 Планування програмного проєкту 129
Теоретичні відомості 129
Контрольні запитання 143
Завдання 143

Лабораторна робота 9 Календарне планування робіт проєкту 144
Теоретичні відомості 144
Контрольні запитання для самоконтролю 145
Лабораторне завдання 145

Лабораторна робота 10 Планування ресурсів проєкту 150
Теоретичні відомості 150
Контрольні запитання для самоконтролю 150
Лабораторне завдання 151

Лабораторна робота 11 Декомпозиція програм 154
Теоретичні відомості 154
Контрольні запитання для самоконтролю 162
Лабораторне завдання 163

Самостійна робота 6 Класифікація видів та напрямків тестування 164
Теоретичні відомості 164
Контрольні запитання для самоконтролю 170
Завдання 171

Лабораторна робота 12 Розробка специфікації вимог до програмного продукту 172
Теоретичні відомості 172
Контрольні запитання для самоконтролю 181
Лабораторне завдання 181

Самостійна робота 7 Тестування вимог 182
Теоретичні відомості 182
Контрольні запитання 186
Завдання 186

Лабораторна робота 13 Добір видів тестування 187
Теоретичні відомості 187
Контрольні запитання для самоконтролю 189
Лабораторне завдання 189

Список літератури 192

Передмова

У навчально-методичному посібнику подано короткий теоретичний матеріал і завдання для лабораторних та самостійних робіт з дисципліни «Основи програмної інженерії». Запропоновано до виконання тринадцять лабораторних та сім самостійних робіт. Завдання містять індивідуальні варіанти завдань. У подальшому при оцінюванні знань викладач може враховувати рівень складності виконання роботи та оптимальність підходів до виконаної роботи.

Мета розробки навчально-методичного посібника полягає у підвищенні ефективності засвоєння теоретичних і практичних знань та організації самостійної роботи студентів по засвоєнню матеріалу навчальної дисципліни. Систематизований та логічно послідовний виклад практичного змісту курсу сприяєтиме поліпшенню якості підготовки студентів в цілому.

Перед виконанням лабораторного завдання студентові потрібно:

– уточнити у викладача індивідуальне завдання;

– вивчити відповідні розділи теоретичного курсу згідно з лекційними записами та навчальною літературою;

– підготувати протокол виконання лабораторної роботи і подати його викладачеві для перевірки.

До виконання лабораторної роботи допускається студент, який має попередньо підготовлений самостійно заповнений протокол лабораторної роботи.

Зміст протоколу лабораторної роботи:

– назва теми і мета лабораторної роботи;

– відповіді на контрольні запитання;

– результати роботи на комп’ютері.

Правильність роботи програми та здобутих результатів перевіряються і оцінюються викладачем.

Структура дисципліни

Навчальна дисципліна «Основи програмної інженерії» знайомить майбутніх програмістів з основними аспектами виробництва програмного забезпечення (ПЗ) від початкових стадій створення специфікації до підтримки системи після здачі в експлуатацію. Крім того, дисципліна знайомить студентів із соціальними та професіональними питаннями програмування, зокрема з професійною та етичною відповідальністю фахівця з програмної інженерії.

Метою дисципліни «Основи програмної інженерії» є ознайомлення студентів з основними поняттями, методами та засобами програмної інженерії, а також формування у здобувачів розуміння принципів розробки ефективного програмного забезпечення та набуття ними навичок використання основних принципів реалізації етапів життєвого циклу ПЗ.

Після успішного завершення цього модуля здобувач вищої освіти буде здатен:

– здійснювати професійну діяльність на основі знань основ програмної інженерії, ключових понять, методів і засобів програмної інженерії, як діяльності, спрямованої на створення програмних систем, що відповідають потребам замовників;

– знати професійні та етичні вимоги до фахівців з програмної інженерії, кодекс ділової етики, правила дисципліни зобов'язань як форми відносин і соціальних взаємин, що виникають при колективній розробці ПЗ;

– володіти питаннями життєвого циклу ПЗ, основами управління проєктами, якості програмного забезпечення, іншими поняттями та вимогами державних та міжнародних стандартів у галузі розробки ПЗ;

– застосовувати на практиці знання методів і засобів розробки ПЗ, стандартів, правил і методик управління розробкою ПЗ, принципів системного підходу до розробки ПЗ;

– володіти фундаментальними поняттями уніфікованої мови моделювання UML, яка є невід'ємною частиною процесу розробки ПЗ і може бути застосована на всіх етапах життєвого циклу аналізу бізнес-систем і розробки прикладних програм;

– моделювати різні аспекти системи, для якої створюється ПЗ.

Перелік тем лабораторних робіт

Лабораторна робота № 1. Система контролю версій Git.

Лабораторна робота № 2. Створення віддаленого репозиторія GIT.

Лабораторна робота № 3. Робота з віддаленим репозиторієм GIT.

Лабораторна робота № 4. Діаграми UML. Ознайомлення з інтерфейсом StarUML.

Лабораторна робота № 5. Створення діаграми варіантів використання (Use Cases Diagrams).

Лабораторна робота № 6. Створення діаграми послідовностей (Sequence Diagrams).

Лабораторна робота № 7. Створення діаграм діяльності (Activity Diagrams).

Лабораторна робота № 8. Створення діаграми класів (Class Diagrams).

Лабораторна робота № 9. Календарне планування робіт проєкту.

Лабораторна робота № 10. Планування ресурсів проєкту.

Лабораторна робота № 11. Декомпозиція програм.

Лабораторна робота № 12. Розробка специфікації вимог до програмного продукту.

Лабораторна робота № 13. Добір видів тестування.

Кожна із запропонованих до виконання робіт має теоретичні відомості з докладним описом порядку виконання та контрольні запитання для закріплення відповідного тематичного матеріалу.

Перелік тем самостійної роботи

Самостійна робота № 1. Контроль версій ПЗ засобами сервісу Replit.

Самостійна робота № 2. Проєктування програмного забезпечення.

Самостійна робота № 3. DevOps.

Самостійна робота № 4. DevSecOps.

Самостійна робота № 5. Планування програмного.

Самостійна робота № 6. Класифікація видів та напрямків тестування.

Самостійна робота № 7. Тестування вимог.

Правильність виконаних лабораторних і самостійних робіт перевіряє викладач.

Лабораторна робота 1
Система контролю версій Git

Мета роботи: вивчити основні засоби роботи з репозиторієм Git.

Теоретичні відомості

Коли над проєктом працює команда розробників, постає питання: "Як не заплутатися у версіях проєкту та поєднувати роботу різних розробників?" Як відповідь на це було придумано систему контролю версій Git. Вона дозволяє підтримувати актуальні версії файлів серед усіх розробників, "відкотитися" на останню робочу версію у разі помилки і працювати розробнику з різних комп'ютерів у своєму профілі.

Git — це безкоштовна розподілена система контролю версій з відкритим вихідним кодом, призначена для швидкої та ефективної роботи від малих до дуже великих проєктів.[1]

Три стани

Git має три основні стани, в яких можуть перебувати файли:

- **збережений у коміті (commited)** – означає, що дані безпечно збережено в локальній базі даних (репозиторії);
- **змінений (modified)** – означає, що у файл внесено редагування, які ще не збережено в базі даних;
- **індексований стан (staged)** – виникає тоді, коли ви позначаєте змінений файл у поточній версії, щоб ці зміни ввійшли до наступної фіксації коміту.

З цього випливають **три основні частини проєкту** під управлінням Git: директорія Git, робоче дерево та індекс (рис. 1.1) [2].

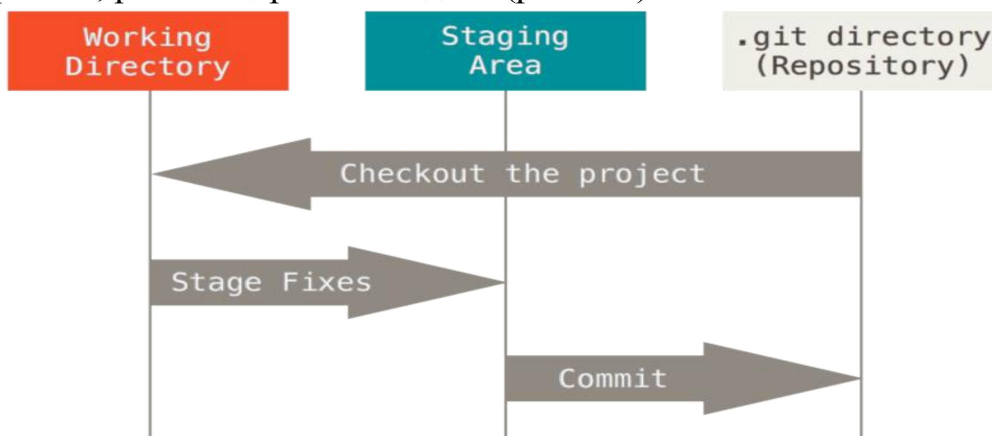


Рисунок 1.1. Робоча директорія, індекс та директорія Git

[1] Git. URL: https://git-scm.com/

[2] Вступ - Про систему контролю версій. URL: https://git-scm.com/book/uk/Вступ-Три-стани

У **директорії** Git система зберігає метадані та базу даних об’єктів вашого проєкту. Це найважливіша частина Git, саме вона копіюється при *клонуванні* сховища з іншого комп’ютера.

Робоче дерево — це одна окрема версія проєкту, взята зі сховища. Ці файли видобуваються з бази даних у папці Git і розміщуються на диску для подальшого використання та редагування.

Індекс — це файл, який зазвичай розміщено в директорії Git і містить інформацію про те, що буде збережено у наступному коміті. Також цей файл називають “областю додавання” (staging area), проте переважно користуються технічним терміном Git “індекс”.

Найпростіший процес взаємодії з Git виглядає приблизно так:

1. ви редагуєте файли у своїй робочій директорії;
2. вибірково надсилаєте до індексу лише ті зміни, які бажаєте зберегти в наступному коміті, і лише ці зміни буде збережено в індексі;
3. створюєте коміт: знімок з індексу остаточно зберігається в директорії Git.

Якщо окрема версія файлу вже є в директорії Git, цей файл вважається збереженим у коміті. Якщо він зазнав змін і перебуває в індексі, то він індексований. Якщо ж його стан відрізняється від того, який був у коміті, і файлу немає в індексі, то він називається зміненим.

Послідовність створення Git-репозиторія розглянуто на офіційному сайті https://git-scm.com/book/uk/v2/Основи-Git-Створення-Git-репозиторія#ch02-git-basics-chapter.

Посібник з повним списком усіх команд Git можна подивитись на сайті https://git-scm.com/docs.

Контрольні запитання для самоконтролю

1. Як створити Git-репозиторій?
2. Як долучити файл до репозиторія для контролю версій?
3. У скількох станах можуть перебувати файли в репозиторії Git?
4. Яка команда використовується для визначення стану файлу в репозиторії?
5. Для чого використовується файл .gitignore?
6. Якою командою переглянути непроіндексовані зміни, зроблені після останнього коміту?
7. Якою командою виконується фіксація змін?
8. Як видалити файл з Git-репозиторія?
10. Як переглянути історію комітів?
11. Як змінити останній коміт?
12. Яка команда створить нову гілку в Git-репозиторії?
13. Як об‘єднати гілки А та Б?

Лабораторне завдання

1 Перші кроки у роботі з Git

1) Спочатку треба встановити Git на ваш комп’ютер.

Якщо ви користувач Windows, потрібно перейти на сайт **Помилка! Неприпустимий об'єкт гіперпосилання.** завантажити відповідну версію Git.

Для Linux потрібно виконати у терміналі команду:

```
apt-get install git
```

Подальші кроки однакові для обох ОС.

2) Відкрити термінал (консоль) та пройти реєстрацію за допомогою команд:

```
git config --global user.name "Name"
git config --global user.email ″email@fcit.ua″
```

Name та email тут вказувати власні.

3) У вигляді перших кроків пропонуємо створити папку з назвою “FCIT”, перейти до неї та створити файл “student.html”:

```
mkdir FCIT
cd FCIT
touch student.html
```

Перейти до іншої папки можна командою:

```
cd <шлях до іншої папки>
```

Щоб перейти у папку, вищу у ієрархії, слід виконати команду:

```
cd ..
```

Щоб перейти у папку іншого диску, слід виконати команду `cd` з параметром `/d`, наприклад:

```
cd /d d:\Student\FCIT\
```

Така команда перейде до щойно створеної папки FCIT.

Переглянути вміст поточної папки можна командою:

```
ls
```

Для Windows ця команда зветься `dir`.

До речі, для створення папки “FCIT” та виконання інших файлових операцій можна використовувати звичайну програму *Провідник* Windows (Explorer). Зайшовши у створену папку можна натиснути ПКМ та вибрати пункт контекстного меню “Git Bash Here”. Після цього консоль Git відкриється саме у поточній папці *Провідника*.

4) Відкрити цей файл за допомогою будь-якого текстового редактора і вписати туди свої ім’я та прізвище.

5) Тепер потрібно ініціалізувати репозиторій, тобто дати Git знати, що ви бажаєте надалі контролювати версії цього проєкту, і, можливо, завантажити його до будь-якого сервісу Git. Щоб ініціалізувати репозиторій, треба виконати команду:

```
git init
```

6) Долучити файл до репозиторія:

```
git add student.html
git commit -m "First Commit"
```

У відповідь ви побачите таке:

```
$ git add student.html
$ git commit -m "First Commit"
[master (root-commit) 911e8c9] First Commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 student.html
```

7) Перевірити стан репозиторія за допомогою команди:

```
git status
```

У відповідь побачите:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Команда перевірки стану повідомляє, що всі файли, які є у директорії, були долучені до репозиторія.

8) Знов відкрити файл “student.html”, відокремити ваше прізвище за допомогою HTML тегу та зберегти файл.

```
Taras <b>Tarasov</b>
```

9) Перевірити стан репозиторія знову. Після змінення файлу “student.html”, ви повинні отримати:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   student.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Тобто Git одразу зрозумів, що файл “student.html” було змінено, але ці зміни ще не зафіксовані у репозиторії. Якщо ви бажаєте зберегти ці зміни, то варто знову додати цей файл за допомогою команди “add”, якщо ні – зробіть команду:

```
git checkout
```

10) Після збереження змін, створити ще 2 файли з назвами “junior.html” та “senior.html”, наповнити їх довільним HTML-кодом.

11) Ви вже вмієте долучати файли та фіксувати (робити коміт), проте що робити, якщо треба завантажити декілька файлів з різними комітами, щоб відрізняти їх? Тоді варто зробити таке:

```
git add junior.html
git add senior.html
git commit -m "People who work"

git add student.html
git commit -m "People who study"
```

12) Якщо потрібно додати до коміту всю директорію — цей спосіб є найбільш зручним та поширеним:

```
git add .
```

2 Робота з гілками

При роботі над великим проєктом його розбивають на модулі та дають доступ декільком розробникам. Кожен з них робить свою частину, проте їм потрібно буде зібрати ці частини докупи. У такому разі стане у нагоді використання гілок.

1. Спочатку треба створити файл, з якого починатиметься робота. Як приклад пропонуємо створити проєкт «Калькулятор»:

```
touch calc.cpp
```

2. У ньому потрібно написати основу майбутнього калькулятора:

```
#include <iostream>
using namespace std;
int main() {
  int a, b, op;   double result;
  cout << "Введіть перше число\n";   cin >> a;
  cout << "Введіть оператор\n1\t+\n2\t-\n3\t*\n4\t/\n\n";  cin
>> op;
  cout << "Введіть друге число\n";     cin >> b;
  switch(op) {
    case 1:     // code
                break;
    case 2:    // code
                break;
    case 3:     // code
                break;
    case 4:     // code
                break;
  }
  cout << "Результат = " << result;
}
```

3. Далі потрібно створити гілку для операції додавання:

```
git checkout -b addition
git status
```

4. Наступним кроком розробляється модуль для операції додавання. Після розробки код “case 1” виглядатиме так:

```
switch(op) {
  case 1:    result = a + b;  break;
  ...
}
```

5. Потрібно зафіксувати («закомітити») результати:

```
git add calc.cpp
git commit -m "addition"
git status
```

6. Для створення інших гілок потрібно перейти до головної гілки та зробити нові гілки:

```
git checkout master
git status
git checkout -b subtraction
git status
```

7. Далі повторити дії для решти операцій.
8. Після того, як буде зроблено кожен з модулів калькулятора, потрібно з'єднати їх з головною гілкою master:

```
git checkout addition
git merge master
git status
```

9. Після з'єднання кожної з гілок до головної, ви повинні отримати такий результат:

```
#include <iostream>
using namespace std;
int main() {
  int a, b, op;
  double result;
  cout << "Введіть перше число\n";
  cin >> a;
  cout << "Введіть оператор\n1\t+\n2\t-\n3\t*\n4\t/\n\n";
  cin >> op;
  cout << "Введіть друге число\n";
  cin >> b;
  switch(op) {
    case 1:
           result = a + b;
           break;
    case 2:
           result = a - b;
           break;
    case 3:
           result = a * b;
           break;
    case 4:
           result = (double) a / b;
           break;
  }
  cout << "Результат = " << result;
}
```

10. Виконати аналіз отриманих результатів. Оформити протокол лабораторної роботи та відповісти на контрольні питання.

Самостійна робота 1
Контроль версій ПЗ засобами сервісу Replit

Теоретичні відомості

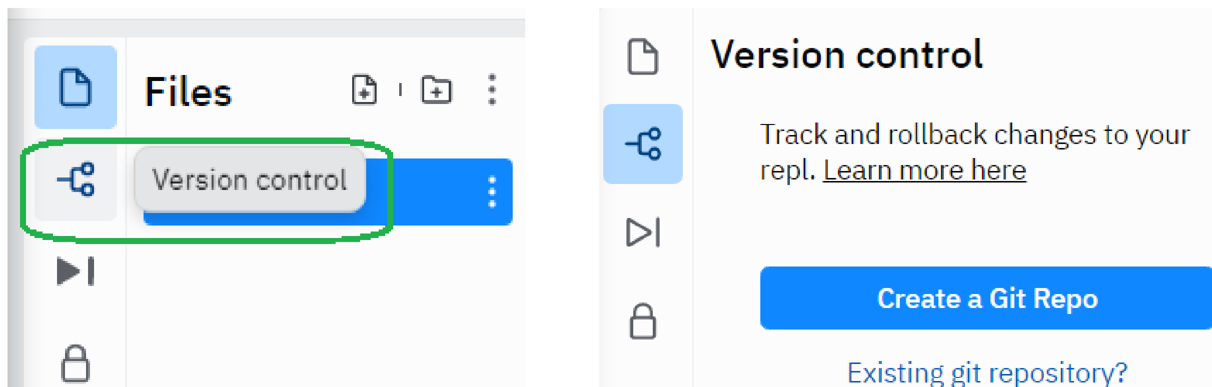
Replit (https://replit.com/, раніше repl.it) – онлайнове інтегроване середовище розробки (IDE), створене 2016 року. Його назва походить від абревіатури REPL (Read – eval – print loop), що означає «цикл читання – оцінки – друку»[3]. Сервіс Replit дозволяє користувачам писати код та створювати програми та вебсайти безпосередньо у браузері в режимі реального часу, і навіть кількома користувачами водночас.

Replit є інтерактивним середовищем програмування, що підтримує понад 60 мов популярних мов програмування (C++, Java, Python, HTML тощо), і дозволяє користувачам створювати програми та вебсайти. Користувачу надається контейнер на віртуальній машині, де може виконуватися його програмний код.

Replit інтегровано з GitHub, що надає можливість імпортувати та запускати проєкти з GitHub.

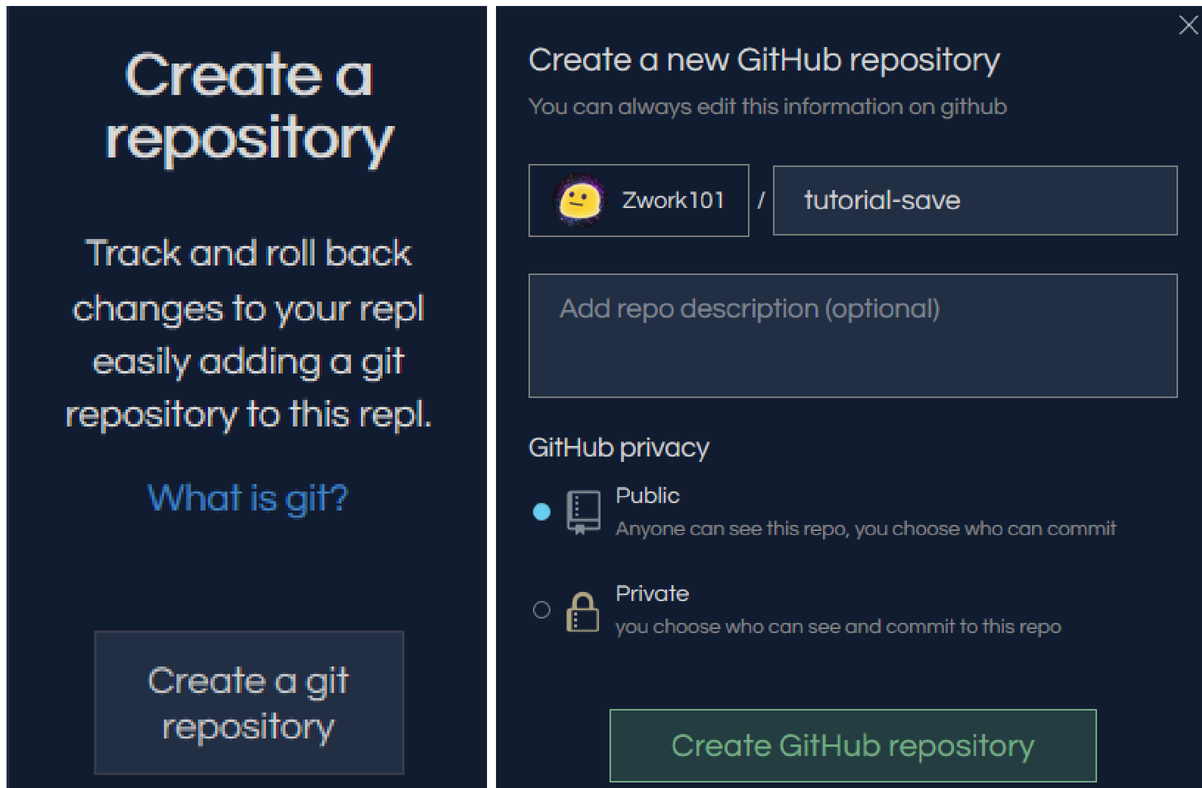
Як розпочати?

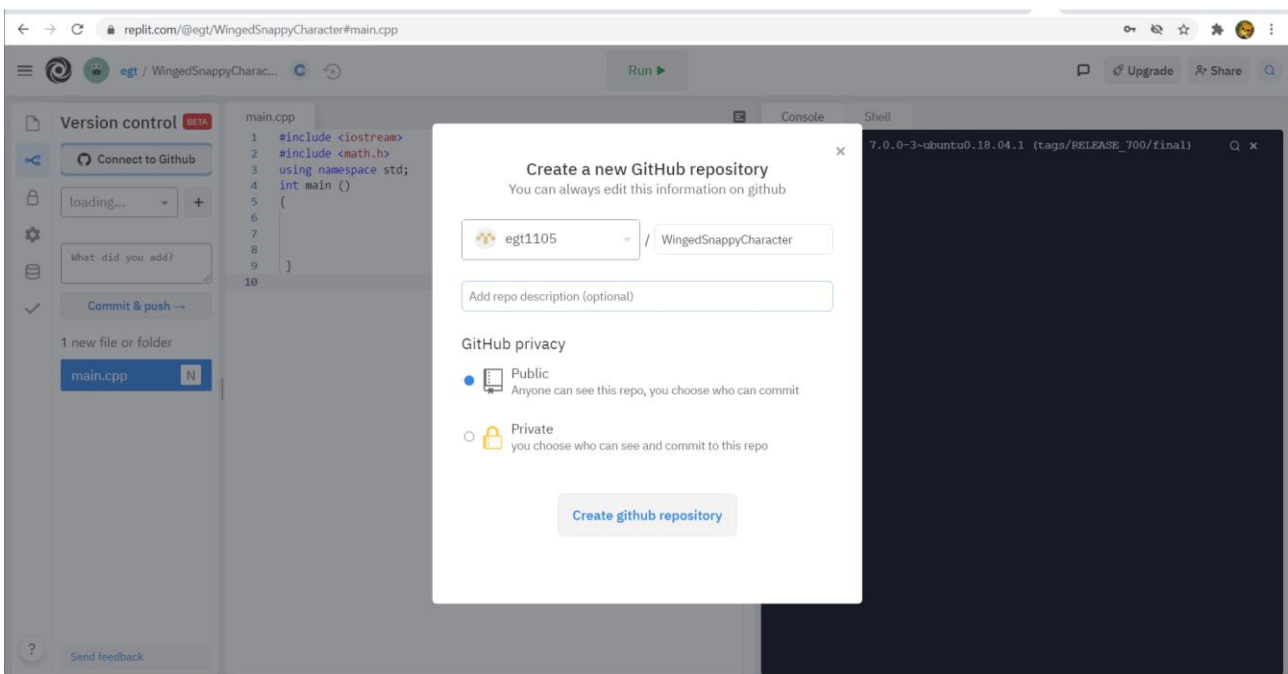
На бічній панелі відкритого проєкту (repl) є значок git, схожий на гілку.



Натискання кнопки «Create a git repository» (Створити git репозиторій) виконає інтеграцію Replit з профілем користувача у Github. Якщо у користувача ще немає облікового запису, можна створити його безпосередньо тут. Після цього потрібно підключити свій проєкт Replit до сховища. Сховищем тут є папки з програмним кодом. Натискання кнопки "Connect to Github" (Підключитися до Github) призведе до появи діалогового вікна, в якому варто заповнити поля або погодитись зі значеннями, які там пропонує система.

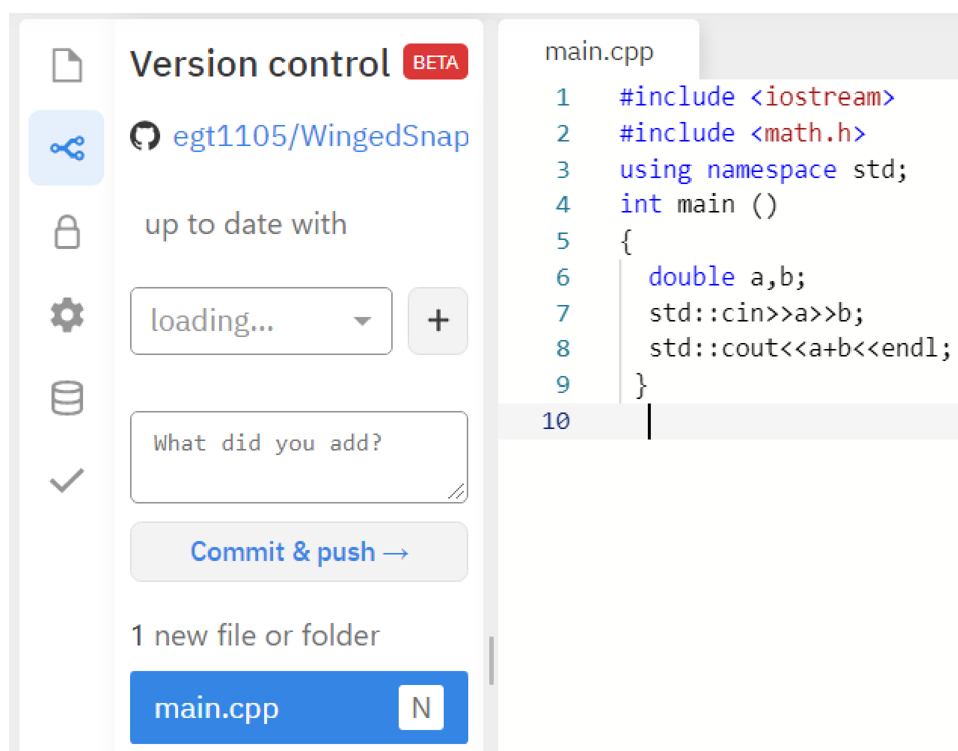
[3] Replit. URL: https://en.wikipedia.org/wiki/Replit

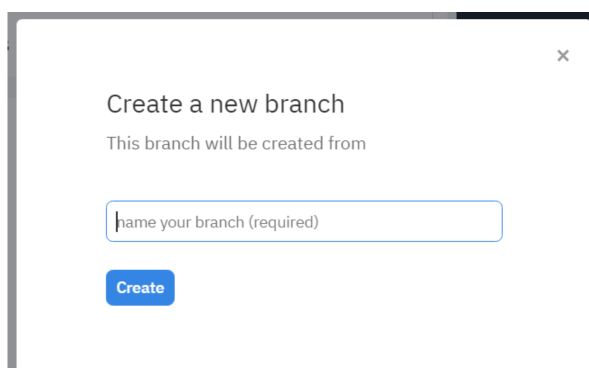


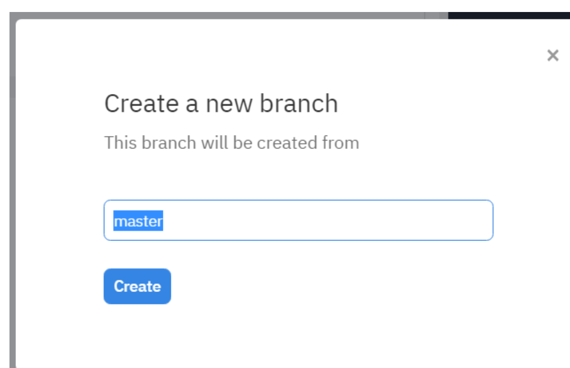


Також у цьому вікні можна задати опис репозиторія.

Далі можна зробити свій repl-проєкт публічним або приватним. Якщо проєкт є загальнодоступним, будь-хто зможе побачити цю роботу, додати або змінити частину програмного коду. Звичайно спочатку таку зміну розробник має схвалити, проте він може й уберегти свій код від змін. – Вибір за розробником, але це не впливає на те, як Replit взаємодіє з його сховищем.

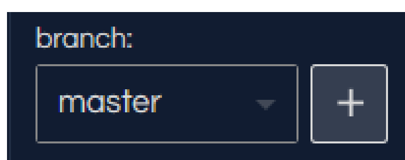


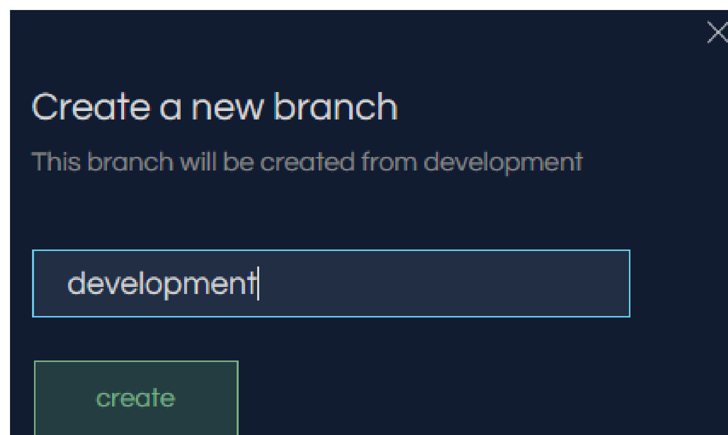




Призначення кнопок

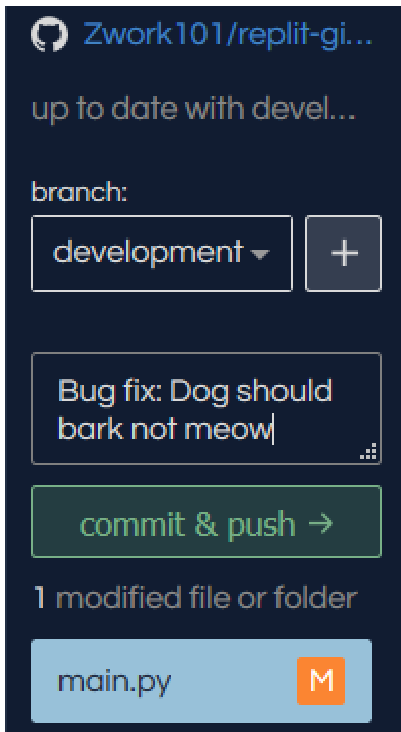
Вгорі вікна є розкривне меню для гілок:





Гілки (branch) використовуються для того, щоб розробники могли організувати свої зміни. *Master* (головна гілка) – це поточний робочий код. Щоб створити нову гілку, треба натиснути "+", і назвати її, наприклад, "*development*" (*розробка*).

Попрацювавши деякий час над своїм проєктом, можна натиснути "commit & push".



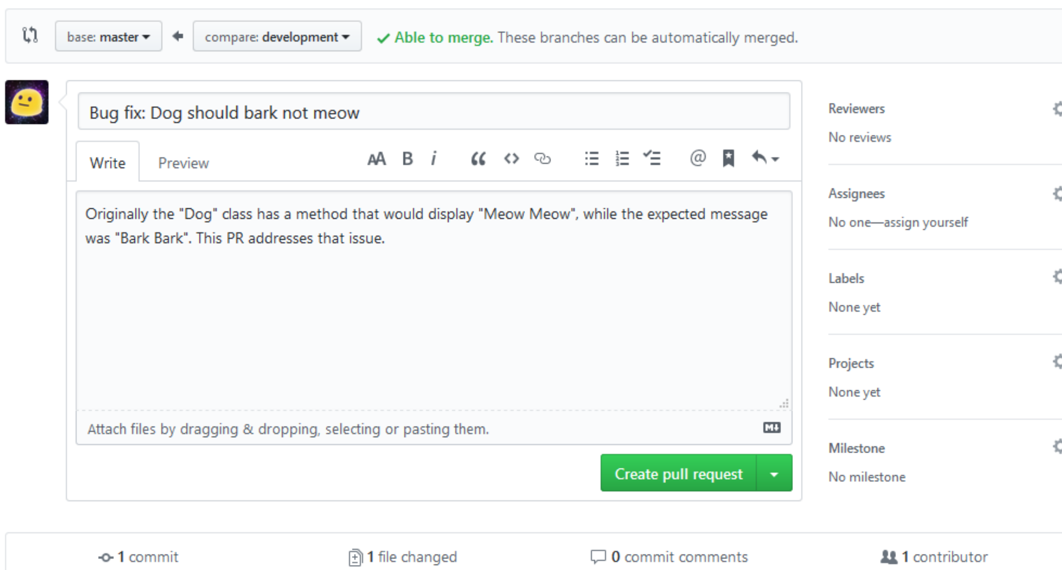
https://storage.googleapis.com/replit/images/1575231987038_e3886c46d52f40a699ac9ccd4a61bd94.png

Якщо після цього переключити гілки, то буде видно, що *master* не має такого коду, який є в гілці *development*. Це не помилка. Щоб "злити" код у *master*, потрібно створити "Pull request". Перейшовши до свого сховища Github, побачите:

Your recently pushed branches:

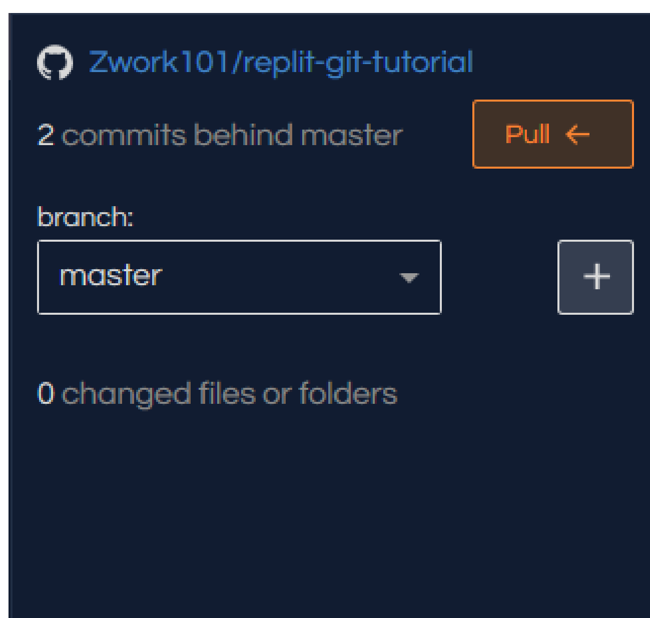
development (33 minutes ago) Compare & pull request

Натиснувши зелену кнопку "Compare & pull request" (Порівняти та запитати витягування), можна конкретизувати опис змін. Після цього варто натиснути кнопку "Create pull request" (Створити запит на витягування).



Тепер можна коментувати та переглядати зміни. Github повідомить, чи сумісні зміни з *головною* гілкою. Інколи це не так, і тоді можна об’єднати код власноруч. На щастя, це трапляється не часто, і в Github у будь-якому випадку це легко зробити. Будь-які подальші зміни, внесені в гілку *розробки,* будуть відображені в цьому запиті на витягування, дозволяючи вносити зміни, які запропоновані іншими розробниками. Після розгляду запиту на витягування можна натиснути зелену кнопку "merge pull request" (поєднати запит на витягування). Це поєднає гілку *розробки* з *головною* гілкою. Потім Github запитає, чи хоче розробник видалити гілку *розробки*, яку при бажанні можна відтворити пізніше.

Якщо після виконаних дій повернутися до repl і перейти на *головну* гілку, стане доступною нова кнопка Pull.



Replit виявe, що у сховище внесено зміни і поточна програма застаріла. Натискання "Pull <–" (Витягнути) призведе до оновлення коду. Replit оновить код новим вмістом репозиторія, і розробник зможе продовжувати працювати.

Контрольні запитання

1. Назвіть основні етапи життєвого циклу ПЗ.
2. Назвіть призначення та можливості системи контролю версій Git.
3. Назвіть призначення та можливості ресурсу Github.com.
4. Які операції з локальним репозиторієм Git вам відомі?
5. Які операції з віддаленим репозиторієм Git вам відомі?
6. Які можливості надає ресурс Replit для організації контролю версій?

Завдання

1. Використовуючи методичний матеріал, ознайомитись з особливостями організації контролю версій засобами Git у сервісі Replit.
2. Створити новий програмний проєкт у Replit.
3. Долучити його до свого публічного репозиторія Github.
4. Вмістом програмного проєкту зробити програмний код до лабораторної роботи на тему «Одновимірні масиви» з курсу «Алгоритмізація та програмування».
5. Створити принаймні дві гілки та три коміти у цьому проєкті.
6. Відпрацювати злиття гілок та відкочування до попереднього коміту (revert).
7. Дати відповіді на контрольні запитання.

Лабораторна робота 2
Створення віддаленого репозиторія GIT

Мета роботи: вивчити основні засоби створення та наповнення віддаленого репозиторія Git у консолі.

Теоретичні відомості

У лабораторній роботі 1 виконувалася робота у локальному репозиторії Git, розміщеному в якійсь папці на комп'ютері користувача. Це є прийнятним для невеликих проєктів, але більш функціональною є робота з *віддаленим репозиторієм*, доступ до якого може отримати будь-який розробник команди, підключений до мережі Інтернет.

Основні операції з віддаленими сховищами, які підтримує Git:

- підключення до віддаленого репозиторія (remote add);
- відправлення змін до сервера (push);
- запит змін із сервера (pull);
- клонування репозиторія (clone);
- отримання нових даних із сервера (fetch).

З повним переліком команд Git для роботи з віддаленим репозиторієм можна ознайомитись на офіційному сайті:
https://git-scm.com/book/uk/v2/-Основи-Git-Взаємодія-з-віддаленими-сховищами

Контрольні запитання для самоконтролю

1. Для чого використовується команда add при роботі з Git?
2. Для чого використовується команда init?
3. Для чого використовується команда status?
4. Для чого використовується команда commit?
5. Що необхідно писати в коментарях комітів?
6. Для чого використовується команда checkout?
7. Для чого використовується команда reset?
8. Для чого використовується команда log?
9. Як відновити кілька файлів з попередньо створеного коміта?
10. Як перейти до попереднього коміта по імені?
11. Як перейти до попереднього коміта по коментарю?
12. Як створити нову гілку?
13. Як перейти на нову гілку?
14. Як перейти на основну гілку?
15. Як виконати повне злиття гілок?
16. Як виконати часткове злиття гілок?

Лабораторне завдання

1 Створення віддаленого репозиторія

Щоб завантажити що-небудь у віддалений репозиторій, спочатку потрібно до нього підключитися. Для цього, його потрібно попередньо створити.

1.1. Зареєструватися на сайті github.com і створити свій репозиторій.

Звертаємо увагу на те, що для опції public/private при створенні сховища для цієї роботи слід вибрати public (публічний доступ).

1.2. Ініціалізувати свій наявний локальний репозиторій з попередньої лабораторної роботи:

```
$ cd <шлях до папки>
$ git init
```

1.3. Приєднати локальний репозиторій до створеного віддаленого репозиторія на github командою:

```
$ git remote add origin https://github.com/FCIT/Name_lab2.git
```

Тут треба вказати свій URI репозиторія.

Важливо знати, що один проєкт може мати декілька віддалених репозиторіїв водночас. Щоб їх розрізняти, вони мають мати різні імена. Звичайно головний репозиторій називається `origin`.

2 Створення нового функціоналу у новій гілці

2.1. Створити нову гілку.

Основна гілка в кожному репозиторії називається `master`. Щоб створити ще одну гілку, можемо використати команду `git branch <name>` (у попередній роботі для цього була використана команда `git checkout -b <name>`, яка, до того ж, автоматично виконувала перехід до новоствореної гілки).

```
$ git branch new_feature
```

Це створить нову гілку – поки що точну копію гілки `master`.

2.2. Перейти до створеної гілки.

Якщо виконати команду `branch`, ви побачите дві доступні опції:

```
$ git branch
new_feature
* master
```

`master` – це активна гілка, вона позначена зірочкою. Щоб працювати з нашою новою “фічею”, треба перейти на цю гілку.

```
$ git checkout new_feature
```

2.3. Створити у цій новій гілці свій код для файлу `calc.cpp`, який реалізуватиме *нову операцію* калькулятора з попередньої роботи – піднесення числа a до степеню b.

2.4. Зафіксувати (закомітити) новий функціонал, впевнитися у його працездатності, після чого об’єднати гілки.

Спочатку, потрібно проіндексовати змінений файл. Для цього виконати:

```
$ git add calc.cpp
```

Виконати коміт нового функціоналу можна командою:

```
$ git commit -m "New feature added"
```

Перейти назад на гілку master дозволить команда:

```
$ git checkout master
```

Об'єднати гілки (застосувати зміни з new_feature до основної версії проєкту) можна командою:

```
$ git merge new_feature
```

Тепер гілка master є актуальною. Гілка new_feature більш не потрібна і її можна вилучити командою:

```
$ git branch -d new_feature
```

3 Надсилання змін до сервера

3.1. Надіслати локальний коміт на сервер. Цей процес відбувається щоразу, коли потрібно оновити дані у віддаленому репозиторії.

Команда, що призначена для цього: push. Вона має два аргументи: *ім'я віддаленого репозиторія* (у вас це origin) і *гілка*, в яку варто внести зміни (усталено для всіх репозиторіїв це гілка master):

```
$ git push origin master
```

3.2. Проаналізувати здобуті результати, відповісти на контрольні запитання та оформити протокол лабораторної роботи.

Лабораторна робота 3
Робота з віддаленим репозиторієм GIT

Мета роботи: вивчити основні засоби роботи з віддаленим репозиторієм Git у консолі.

Теоретичні відомості

$ git fetch – це основна команда, яка використовується для завантаження вмісту віддаленого репозиторія.

$ git pull – оновлення локального репозиторія та робочої папки з віддаленого репозиторія. Ця команда складається з двох етапів:

- синхронізує віддалений репозиторій з вашим локальним ($ git fetch);
- мерджить (об'єднує) нові зміни, які відбулися щойно в локальному репозиторії, з вашою робочою папкою ($ git merge).

$ git clone – створення копії віддаленого репозиторія у локальній папці. Працює як обгортка над деякими іншими командами. Ця команда створює новий каталог, переходить всередину і виконує git init для створення порожнього репозиторія. Потім вона додає новий віддалений репозиторій (git remote add) для зазначеного URL (усталено він отримає ім'я origin), виконує git fetch для цього репозиторія і, нарешті, витягує останній коміт у ваш робочий каталог, використовуючи git checkout.

$ git revert – це операція для безпечного скасування змін. Для відкочування змін команда не видаляє з історії коміти або батьківські елементи, а створює новий коміт зі скасуванням потрібних дій. Використовувати git revert безпечніше, тому що вона не створює загрози втрати коду, на відміну від команди git reset.

Часто виникає ситуація, коли у локальному сховищі є фали, які не потрібно включати до комітів. Для ігнорування таких файлів передбачене створення спеціального файлу .gitignore у папці .git. Зазвичай приховують конфігураційні файли (особливо з паролями), тимчасові файли та папки. gitignore використовує формат glob для вибірки файлів, наприклад:

```
# Ігнорувати файл readme.txt
readme.txt
# Ігнорувати html-файли
*.html
# Але, саме index.html не ігнорувати
!index.html
```

З повним переліком команд Git для роботи з віддаленим репозиторієм можна ознайомитись на офіційному сайті:
https://git-scm.com/book/uk/v2/-Основи-Git-Взаємодія-з-віддаленими-сховищами

Контрольні запитання для самоконтролю

1. Для чого використовується команда clone при роботі з Git?
2. Для чого використовується команда remote?
3. До чого відноситься ім’я origin?
4. Для чого використовується команда pull?
5. Для чого використовується команда push?
6. Для чого використовується команда revert?
7. Для чого використовується команда fetch?
8. Для чого використовується файл .gitignore?

Лабораторне завдання

1 Клонування віддаленого репозиторія

1.1. Підготувати папку для локального репозиторія у консолі Git Bash:

```
$ cd <шлях до папки>
$ git init
```

1.2.Клонувати вміст віддаленого репозиторія https://github.com/FCIT-SoftwareEngineeringBasics/Lab3.io :

```
$ git clone https://github.com/FCIT-SoftwareEngineeringBasics/Lab3.io
$ cd Lab3.io
```

2 Змінення копії репозиторія

2.1. На базі завантаженого шаблону у локальному сховищі створити вебсторінку-фотоальбом.

2.2. У файлі “git/.gitignore” локального репозиторія додати рядок, який виключатиме з майбутніх комітів всі файли з розширенням “.txt”:

```
*.txt
```

2.3. Додати та закомітити свої зміни:

```
$ git add images
$ git commit -m "My album"
```

3 Відкочування «невдалого» коміту

3.1. Додати «невдалий» функціонал, для чого записати у локальне сховище файл “bad_feature.js” (з довільним вмістом) та закомітити його:

```
$ git add bad_feature.js
$ git commit -m "Bad feature"
```

3.2. Подивитися хеш минулих комітів та скопіювати хеш коміта “Bad feature”:

```
$ git log
```

Для завершення перегляду логу натисніть “q”.

3.3. Відкотити коміт “Bad feature”:

```
$ git revert <хеш коміта "Bad feature">
```

4 Створення та наповнення своєї версії віддаленого репозиторія

4.1. Зареєструватися на сайті github.com та створити власний віддалений репозиторій з ім’ям формату

```
Lab3_<ваші П.І.Б.>.io
```

4.2. Видалити поточний origin:

```
git remote rm origin
```

4.3. Призначити новий origin:

```
$ git remote add origin https://github.com/<Ім'я_акаунта>/Lab3_<ваші П.І.Б.>.io
```

4.4. Відправити останній коміт до віддаленого репозиторія:

```
$ git push origin master
```

4.4 Проаналізувати здобуті результати, відповісти на контрольні запитання та оформити протокол лабораторної роботи.

Самостійна робота 2
Проєктування програмного забезпечення

Теоретичні відомості

Проєктування ПЗ – це процес визначення архітектури, набору компонентів, їх інтерфейсів, інших характеристик системи і кінцевого складу програмного продукту.

Область знань «Проєктування ПЗ (Software Design)» складається з таких розділів:

– базові концепції проєктування ПЗ (Software Design Basic Concepts),

– ключові питання проєктування ПЗ (Key Issue in Software Design),

– структура й архітектура ПЗ (Software Structure and Architecture),

– аналіз і оцінка якості проєктування ПЗ (Software Design Quality Analysis and Evaluation),

– нотації проєктування ПЗ (Software Design Notations),

– стратегія і методи проєктування ПЗ (Software Design Strategies and Methods).

Базова концепція проєктування ПЗ – це методологія проєктування архітектури за допомогою різних методів (об'єктного, компонентного й ін.), процеси ЖЦ (стандарт ISO/IEC 12207) і техніки – декомпозиція, абстракція, інкапсуляція й ін. На початкових стадіях проєктування предметна область декомпозується на окремі об'єкти (при об'єктно-орієнтованому проєктуванні) або на компоненти (при компонентному проєктуванні). Для подання архітектури програмного забезпечення вибираються відповідні артефакти (нотації, діаграми, блок-схеми і методи).

Ключові питання проєктування – це декомпозиція програм на функціональні компоненти для незалежного і одночасного їхнього виконання, розподіл компонентів у середовищі функціонування і їх взаємодія між собою, забезпечення якості і живучості системи й ін.

Проєктування архітектури ПЗ проводиться архітектурним стилем, заснованим на визначенні основних елементів структури – підсистем, компонентів, об'єктів і зв'язків між ними.

Архітектура проєкту – високорівневе подання структури системи і специфікація її компонентів. Архітектура визначає логіку системи через окремі компоненти системи настільки детально, наскільки це необхідно для написання коду, а також визначає зв'язки між компонентами. Існують і інші види подання структур, засновані на проєктуванні зразків, шаблонів, сімейств програм і каркасів програм.

Один з інструментів проєктування архітектури – *патерн (шаблон).* Це типовий конструктивний елемент ПЗ, що задає взаємодію об'єктів (компонентів) проєктованої системи, а також ролі і відповідальності виконавців. Основна мова опису – UML. Патерн може бути *структурним*, що містить у собі структуру типової композиції з об'єктів і класів, об'єктів, зв'язків і ін.; *поведінковим*, що визначає схеми взаємодії класів об'єктів і їх поведінку, задається діаграмами діяльності, взаємодії, потоків керування й ін.; *погоджувальним*, що відображає типові схеми розподілу ролей екземплярів об'єктів і способи динамічної генерації структур об'єктів і класів.

Аналіз і оцінка якості проєктування ПЗ – це заходи щодо аналізу сформульованих у вимогах атрибутів якості, функцій, структури ПЗ, з перевірки якості результатів проєктування за допомогою метрик (функціональних, структурних і ін.) і методів моделювання і прототипування.

Нотації проєктування дозволяють представити опис об'єкта (елемента) ПЗ і його структуру, а також поведінку системи за цим об’єктом. Існує два типи нотацій: структурна, поведінкова, та множина їх різних представлень.

Структурні нотації – це структурне, блок-схемне або текстове подання аспектів проєктування структури ПЗ з об'єктів, компонентів, їх інтерфейсів і взаємозв'язків. До нотацій відносять формальні мови специфікацій і проєктування: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity–Relation Diagrams), IDL (Interface Description Language) тощо. Нотації містять у собі мовний опис архітектури й інтерфейсу, діаграм класів і об'єктів, діаграм сутність–зв'язок, конфігурації компонентів, схем розгортання, а також структурні діаграми, що задають у наочному вигляді оператори циклу, розгалуження, вибору і послідовності.

Поведінкові нотації відбивають динамічний аспект роботи системи та її компонентів. Ними можуть бути діаграми потоків даних (Data Flow), діяльності (Activity), кооперації (Colloboration), послідовності (Sequence), таблиці прийняття рішень (Decision Tables), передумови і постумови (Pre-Post Conditions), формальні мови специфікації (Z, VDM, RAISE) і проєктування.

Стратегія і методи проєктування ПЗ. До стратегій відносять: проєктування вгору, вниз, абстрагування, використання каркасів і ін. Методи є функціонально-орієнтовані, структурні, які базуються на структурному аналізі, структурних картах, діаграмах потоків даних й ін. Вони орієнтовані на ідентифікацію функцій і їх уточнення знизу-вгору, після цього уточнюються діаграми потоків даних і проводиться опис процесів.

В об’єктно-орієнтованому проєктуванні ключову роль відіграє спадкування, поліморфізм й інкапсуляція, а також абстрактні структури даних і відображення об'єктів. Підходи, орієнтовані на структури даних, базуються на методі Джексона і використовуються для подання вхідних і вихідних даних структурними діаграмами. Метод UML призначений для опису сценаріїв роботи проєкту у наочному діаграмному вигляді. Компонентне проєктування ґрунтується на використанні готових компонентів (reuse) з визначеними інтерфейсами

і їх інтеграції в конфігурацію, як основи розгортання компонентної системи для її функціонування в операційному середовищі.

Формальні методи опису програм ґрунтуються на специфікаціях, аксіомах, описах деяких попередніх умов, твердженнях і постумовах, що визначають заключну умову одержання програмою правильного результату. Специфікація функцій і даних, якими ці функції оперують, а також умови і твердження – основа доведення правильності програми.

1. Конструювання програмного забезпечення

Конструювання ПЗ – створення ПЗ з конструкцій (блоків, операторів, функцій) і його перевірка методами верифікації і тестування. До інструментів конструювання ПЗ віднесені мови конструювання, програмні методи й інструментальні системи (компілятори, СКБД, генератори звітів, системи керування версіями, конфігурацією, тестуванням й ін.). До формальних засобів опису процесу конструювання ПЗ, взаємозв'язків між людиною і комп'ютером з урахуванням середовища оточення віднесені структурні діаграми Джексона.[4]

Область знань «Конструювання ПЗ (Software Construction)» містить у собі такі розділи:

– зниження складності (Reduction in Complexity),

– попередження відхилень від стилю (Anticipation of Diversity), – структуризація перевірок (Structuring for Validation), – використання стандартів (Use of External Standards).

Зниження складності – це мінімізація, зменшення і локалізація складності конструювання.

Мінімізація складності – це обмеження на обробку складних структур і великих обсягів інформації протягом тривалого періоду часу. Вона досягається, зокрема, використанням у процесі конструювання простих елементів, а також рекомендацій стандартів.

Зменшення складності в конструюванні ПЗ досягається шляхом створення простого коду, що легко читається і спрощує тестування, підвищує продуктивність і впливає на досягнення інших характеристик і обмежень проєкту. Зменшення складності спрощує процеси верифікації і тестування результатів конструювання елементів ПС.

Локалізація складності – це спосіб конструювання з застосуванням об’єктно-орієнтованого підходу, що лімітує інтерфейс об'єктів, спрощує їхню взаємодію, перевірку правильності самих об’єктів і зв'язків між ними. Локалізація призначена для внесення змін, пов'язаних з виявленими помилками в коді, або коли джерелом помилок є середовище, у якому виконується код.

[4] Методичні вказівки до виконання лабораторних робіт з дисципліни «Конструювання програмного забезпечення» / Укладачі: Гусєва-Божаткіна В.А., Пономоренко Т.В. Миколаїв, 2013.

Попередження відхилень від стилю. Для розв'язання різних задач конструювання застосовуються різні стилі конструювання (лінгвістичний, формальний, візуальний).

Лінгвістичний стиль заснований на використанні словесних інструкцій і виразів для подання окремих елементів (конструкцій) програм. Він призначений для конструювання нескладних конструкцій і приводиться до вигляду традиційних функцій і процедур або реалізується методами логічного і функціонального програмування й ін.

Формальний стиль використовується для точного й однозначного визначення компонентів системи, мінімальної кількості помилок, що можуть виникнути в зв'язку з неоднозначністю визначень або невдалих узагальнень об'єктів конструювання ПЗ.

Візуальний стиль – найбільш універсальний для конструювання прикладного ПЗ. Він дозволяє представляти елемент конструювання у наочному вигляді. Візуальна мова проєктування UML надає розробнику набір діаграм для подання статичної і динамічної структур ПЗ. При його застосуванні створюється текстовий і діаграмний опис конструктивних елементів ПЗ, який виводиться на екран дисплея для перегляду і коригування.

Структуризація перевірок припускає, що побудова ПС структурована таким чином, що спрощується пошук помилок, дефектів і різних збоїв у процесі перевірок як на стадії незалежного тестування, так і в процесі експлуатації. Структуризації перевірок сприяють огляд, інспектування, спільний перегляд, модульне тестування із застосуванням автоматизованих засобів тестування й ін.

Використання зовнішніх стандартів. Конструювання ПЗ залежить від застосовних зовнішніх стандартів, пов'язаних з мовами програмування, інструментальними засобами й інтерфейсами. При конструюванні має бути визначений достатній набір стандартів для керування і забезпечення координації між визначеними видами діяльності і групами операцій, мінімізації складності, внесення змін, аналізу ризиків тощо.

До таких стандартів відносять: мови програмування (Java, Ада 95, C++ і ін.), інтерфейси мов програмування (МП) і прикладні інтерфейси платформ Windows (COM, DCOM), CORBA і ін. При конструюванні використовують стандарти мов опису даних (XML, SQL і ін.), засобів комунікації (COM, CORBA і ін.), інтерфейсних мов (POSIX, IDL, APL), UML і ін.

Перелічені вище розділи області знань «Конструювання ПЗ» у ядрі знань SWEBOK об'єднуються в групу «Основи конструювання». Крім того, розглядаються групи розділів «Керування конструюванням» та «Практичні міркування». Опишемо першу з них детальніше.

Керування конструюванням – це керування процесом конструювання ПЗ, планування, оцінка виконання плану і розроблення заходів щодо внесення змін.

Моделі конструювання містять у собі набір операцій, послідовність дій і результатів. Види моделей визначаються стандартом ЖЦ, методологіями і практиками. Деякі стандарти ЖЦ за своєю природою орієнтовані на конструю-

вання засобами екстремального програмування і раціонального уніфікованого процесу – RUP (Rational Unified Process) [13].

Планування – це визначення порядку операцій, термінів і рівня виконання заданих умов у процесі конструкторської діяльності за моделлю ЖЦ, що містить у собі задачі і дії зі створення, перевірки й оцінки показників якості. Виконавці розподіляються за процесами і виконують відповідні задачі з реалізації проміжного і кінцевого продукту. Остаточний результат виміряється за обсягом коду, ступенем повторного використання, кількістю помилок і дефектів, а також оцінюванням показників якості ПЗ.

Внесення змін пов'язане з помилками, виявленими при перевірці і тестуванні, проводиться з метою збереження функціональної цілісності системи. У випадку виявлення помилок на процесі супроводження приймається рішення про внесення змін або заміну коду у цілому.

2. Тестування програмного забезпечення

Тестування ПЗ – це процес перевірки готової програми в статиці (перегляди, інспекції, налагодження вихідного коду) і в динаміці (прогін на наборі тестових даних) з метою перевірки різних шляхів виконання програми і порівняння отриманих результатів із заздалегідь заданими.

Існує дві форми перевірки коду – модульна й інтеграційна. Спочатку використовують стандарти (IEEE 829:1996 і IEEE 1008:1987) з перевірки і тестування модулів. Потім проводиться інтеграційне тестування модулів системи і їх інтерфейсів у динаміці виконання. Під час різних видів перевірок збираються дані про помилки, дефекти, відмови тощо і оформляється відповідна документація (таблиці типів помилок, частоти і часу виявлення відмов і ін.). Зібрані дані використовуються при оцінюванні характеристик якості готового ПЗ, наприклад, надійності.

Область знань «Тестування ПЗ (Software Testing)» містить у собі такі розділи:

– основні концепції і визначення тестування (Testing Basic Concepts and definitions),

– рівні тестування (Test Levels),

– техніки тестування (Test Techniques),

– метрики тестування (Test Related Measures),

– керування процесом тестування (Managing the Test Process).

Дана область знань SWEBOK визначає методи перевірки правильності ПЗ:

верифікація, валідація, тестування. Наводяться типи, рівні і техніки тестування ПЗ, методи планування процесу тестування, розроблення тестових наборів даних для прогону ПЗ в режимі випробування модулів або системи в цілому і наступною оцінкою результатів тестування.

Основна концепція тестування – це базові терміни, ключові проблеми і їхній зв'язок з іншими областями знань. Тестування визначається як процес перевірки правильності програми в динаміці її виконання за тестовими даними.

При тестуванні виявляються недоліки: відмови (faults) і дефекти (defects) як причини порушення роботи програми, збої (failures) як небажані ситуації, помилки (errors) як наслідки збоїв і ін. Базове поняття тестування – тест, що виконується в заданих умовах і за наборами даних. Тестування вважається успішним, якщо знайдено дефект або помилка, і вони відразу усуваються. Ступінь тестованості визначається критерієм покриття системи тестами, перевірки всіх можливих шляхів виконання програм і імовірності припущення стосовно того, що може з'явитися збій або помилкова ситуація в системі.

Рівні тестування:

– *тестування окремих елементів* – це перевірка окремих, ізольованих і незалежних одна від одної частин ПЗ;

– *інтеграційне тестування* орієнтоване на перевірку зв'язків і взаємодії компонентів (інтерфейсів), що можуть розміщуватися на різних архітектурних платформах розподіленого середовища;

– *тестування системи* – це перевірка правильності функціонування системи, пошук і виявлення відмов і дефектів у системі і їхнє усунення. При цьому контролюється виконання сформульованих не функціональних вимог (безпека, надійність і ін.) у системі, правильність подання і здійснення зовнішніх інтерфейсів системи з зовнішнім середовищем.

На всіх рівнях тестування застосовуються методи:

– *функціонального тестування*, які забезпечують перевірку реалізації функцій, що визначені у вимогах, а також правильності їх виконання;

– *регресійного тестування*, що орієнтоване на повторне вибіркове тестування системи або її компонентів після внесення в них змін на тих самих тестах, що і до модифікації;

– *тестування ефективності* – це перевірка продуктивності, пропускної здатності, максимального обсягу даних і системних обмежень відповідно до вимог;

– *стрес-тестування* – це перевірка поведінки системи при максимально припустимому навантаженні або в разі його перевищення;

– *альфа- і бета-тестування* – це тестування системи (альфа) групою тестувальників організації-розробника і тестування системи «зовнішніми» користувачами (бета);

– *конфігураційного тестування* – перевірка структури й ідентифікації системи, а також роботи системи на різних конфігураціях апаратури й устаткування.

Тестуванню підлягають також перевірка реалізації вимог і забезпечення параметрів настроювання і розміщення компонентів ПЗ на заданій кількості і типах комп'ютерів і середовища.

Техніки тестування базуються на певних теоретичних і практичних положеннях щодо проєктування (компонентного, об'єктно-орієнтованого, сервісного і т.п.), а також на таких даних:

– інформація про структуру ПЗ або системи в документації («біла скринька»*)*;

– підбір тестових наборів даних для перевірки правильності роботи компонентів і системи в цілому без знання їх структури («чорна скринька»);

– аналіз граничних значень, таблиць прийняття рішень, потоків даних, статистики відмов і ін.;

– блок-схеми побудови програм і складання наборів тестів для покриття системи цими тестами;

– виявлені і зафіксовані в таблицях системи дефекти, перед- і постумови виконання, структурні характеристики системи (кількість модулів, обсяг даних тощо).

Метрики тестування. Для вимірювання результатів тестування ПЗ й оцінки якості використовуються метрики. Вимір як частина планування і розробки тестів базується на розмірі програм, їх структурі і кількості виявлених помилок і дефектів. Метрики тестування – це вимірювання процесу планування, проєктування і тестування, а також результатів тестування на основі таксономії відмов і дефектів, покриття границь тестування, перевірки потоків даних і ін.

Процес тестування документується і, відповідно до стандарту IEEE 829:1995, містить у собі опис тестових документів, їх зв'язку між собою і з задачами тестування. Без документації на процес тестування неможливо провести сертифікацію продукту за моделями зрілості, зокрема, моделлю CMM [11]. Після завершення тестування оцінюється вартість і ризики ПЗ, викликані збоями або недостатньо надійною роботою системи. Вартість тестування – одне з обмежень, на основі якого приймається рішення про його припинення або продовження.

Керування тестуванням:

– планування процесу тестування (складання планів, тестів, наборів даних) і оцінювання показників якості готового продукту;

– проведення тестування компонентів повторного використання і патернів як основних об'єктів складання ПЗ;

– генерація необхідних тестових сценаріїв, що відповідають середовищу виконання ПЗ;

– верифікація правильності реалізації системи і валідація реалізації вимог до ПЗ;

– збирання даних про відмови, помилки і виявлені непередбачені ситуації при виконанні програмного продукту;

– підготовка звітів за результатами тестування й оцінка характеристик системи.

Відповідно до стандарту ISO/IEC 12207 тестування ПЗ розглядається як невід'ємна частина ЖЦ.

3. Супровід програмного забезпечення

Супровід ПЗ – сукупність дій із забезпечення його роботи, внесення змін при виявленні помилок, адаптації ПЗ до нового середовища функціонування, а також підвищення продуктивності або поліпшення деяких характеристик ПЗ. У

зв'язку з вирішенням так званої проблеми 2000 року (пов’язаної з кодуванням дат у новому тисячолітті, зокрема, у двохсимвольному форматі) супроводження почав розглядатися, як більш важливий процес, що здійснюють розробники. Після змін система має вирішувати ті самі задачі, а також мати план перенесення інформації в інші БД. Супровід відповідно до стандартів ISO/IEC 12207 і ISO/IEC 14764 проводиться з метою виконання і модифікації програмного продукту в процесі експлуатації за умови збереження його цілісності.

Область знань «Супровід ПЗ (Software Maintenance)» складається з таких розділів:

– основні концепції (Basic Concepts),

– процес супроводження (Process Maintenance),

– ключові питання супроводу ПЗ (key Issue in Software Maintenance) , – техніки супроводу (Techniques for Maintenance).

Супровід розглядається з точки зору задоволення вимог споживача у готовому ПЗ, коректності його виконання, процесів навчання й оперативного обліку його процесу.

Основні концепції – це базові визначення і термінологія, підходи до еволюції і супроводу ПЗ, до оцінки вартості супроводу тощо. До основних концепцій можна віднести ЖЦ ПЗ (стандарт ISO/IEC 12207) і складання документації. Головне призначення цієї області знань полягає у виконанні готової програмної системи, фіксації помилок, що виникають при виконанні, дослідженні їх причин, аналізі необхідності модифікації системи з метою усунення помилок, оцінці вартості робіт із проведення змін функцій і системи в цілому. Розглядаються проблеми, пов'язані з ускладненістю продукту при великій кількості змін, і методи її подолання.

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

– коригування, тобто зміни продукту для усунення виявлених помилок або нереалізованих задач;

– адаптації, тобто настроювання продукту в умовах експлуатації, що змінилися, або в новому середовищі виконання;

– поліпшення, тобто еволюційної зміни продукту для підвищення

– продуктивності або рівня супроводу;

– перевірки ПЗ, пошуку і виправлення помилок при експлуатації системи.

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

Техніка супроводу (цей розділ називають також еволюцією ПЗ). Відомий фахівець в області ПЗ Дж. Леман (1970 р.) запропонував розглядати супровід як еволюційну розробку програмних систем, оскільки здана в експлуатацію система не завжди цілком завершена, її треба змінювати протягом терміну експлуатації. Внаслідок змін система стає більш складною і погано керованою. У зв'язку з цим виникає проблема зменшення її складності. До технологій еволюції ПЗ відносять реінженерію, реверсну інженерію і рефакторинг.

Реінженерія – це удосконалення застарілого ПЗ шляхом його реорганізації або реструктуризації, а також перепрограмування окремих елементів або настроювання параметрів на іншу платформу, середовище виконання зі збереженням зручності його супроводу.

Реверсна інженерія полягає у відновленні специфікації (графів викликів, потоків даних і ін.) за отриманим кодом системи для її аналізу на більш високому рівні. Відновлюється ідентифікація компонентів і зв'язків між ними для забезпечення перепрограмування системи на нову платформу. Найчастіше реверсна інженерія застосовується після того, як у код ПЗ було внесено багато змін і воно стало некерованим або змінилася платформа комп'ютера.

Рефакторинг – це реорганізація коду для поліпшення характеристик і показників якості об’єктно-орієнтованих і компонентних програм без зміни їх поведінки. Цей процес реалізується шляхом поступової зміни окремих операцій над текстами, інтерфейсами, середовищем програмування і виконання ПЗ, а також настроювання або внесення змін в інструментальні засоби підтримки ПЗ. Якщо при зміні зберігається формат існуючої системи, то рефакторинг – один з варіантів реверсної інженерії.

4. Керування конфігурацією

Керування конфігурацією – це ідентифікація компонентів системи, визначення функціональних, фізичних характеристик системи, апаратного і програмного забезпечення для контролю виконання, внесення змін і трасування конфігурації. Процес керування визначено як один з допоміжних процесів ЖЦ (ISO/IEC 12207), виконуваний технічним і адміністративним менеджментом проєкту. При цьому складаються звіти про зміни, внесені у конфігурацію, і ступінь їхньої реалізації, а також проводиться перевірка відповідності внесених змін заданим вимогам.

Конфігурація системи – це склад функцій, програмного і технічного забезпечення системи, можливі їх комбінації залежно від наявності устаткування, загальносистемних засобів і вимог до продукту.

Конфігурація ПЗ складається з набору функціональних і технічних характеристик ПЗ, заданих у технічній документації і реалізованих у готовому продукті. Це сполучення різних елементів продукту з заданими процедурами збирання компонентів і настроювання на середовище. Вхідними елементами конфігурації є графік розробки, проєктна документація, вихідний виконуваний код, бібліотека компонентів, інструкції з установки і розгортання системи.

Область знань «Керування конфігурацією ПЗ (Software Configuration Management – SCM)» складається з таких розділів:

– керування процесом конфігурації (Management of SMC Process),

– ідентифікація конфігурації ПЗ (Software Configuration Identification),

– контроль конфігурації ПЗ (Software Configuration Control),

– облік статусу (поведінка або стани) конфігурації ПЗ (Software Configuration Status Accounting),

– аудит конфігурації ПЗ (Software Configuration Auditing),

– керування версіями ПЗ і доставкою (Software Release Management and Delivery).

Керування процесом конфігурації. Це діяльність з контролю еволюції і цілісності продукту при ідентифікації, змінах і забезпеченні звітною інформацією, що стосується конфігурації. Вона містить у собі:

– систематичне відстеження внесених змін в окремі складові частини конфігурації, виконання аудита змін і автоматизованого контролю за внесенням змін у конфігурацію системи або в ПЗ;

– підтримку цілісності конфігурації, її аудит і забезпечення внесення змін в елементи конфігурації;

– ревізію конфігурації з метою перевірки наявності розроблених програмних або апаратних засобів і узгодження версії конфігурації з заданими вимогами;

– трасування змін у конфігурації на процесах супроводу й експлуатації ПЗ.

Ідентифікація конфігурації ПЗ полягає в документуванні функціональних і фізичних характеристик елементів конфігурації, а також в оформленні технічної документація на елементи конфігурації.

Контроль конфігурації ПЗ – це роботи з координації, затвердження або відкидання реалізованих змін в елементах конфігурації після ідентифікації, а також з аналізу вхідних компонентів конфігурації.

Облік статусу або стану конфігурації ПЗ – комплекс заходів для визначення ступеня зміни конфігурації, а також правильності внесених змін у систему при супроводі. Інформація і кількісні показники накопичуються у відповідній БД і використовуються при складанні звітності, оцінюванні якості і виконанні процесів ЖЦ.

Аудит конфігурації – це діяльність, що виконується для оцінки відповідності продукту і процесів стандартам, інструкціям, планам і процедурам. Аудит визначає ступінь задоволення конфігурації функціональним і фізичним (апаратним) характеристикам системи.

Керування версіями ПЗ – це відстеження наявної версії компонентів конфігурації; складання компонентів; створення нових версій системи на основі існуючих шляхом внесення змін у конфігурацію; узгодження версії продукту з вимогами і проведеними змінами на процесах ЖЦ; забезпечення оперативного доступу до інформації про елементи конфігурації і системи, до яких вони належать. Дане керування містить у собі такі основні поняття.

Базис (baseline) – формально позначений набір елементів ПЗ, зафіксований на процесах ЖЦ.

Бібліотека ПЗ – колекція об'єктів ПЗ і документації, призначена для полегшення процесу розроблення, використання і супроводження.

Складання ПЗ – об'єднання коректних елементів і конфігураційних даних у єдину виконувану програму.

5. Керування інженерією програмного забезпечення

Керування інженерією ПЗ (Software Engineering Management) – керування роботами команди розробників ПЗ у процесі виконання плану проєкту, визначення критеріїв ефективності роботи цієї команди й оцінка процесів і продуктів проєкту з використанням загальних методів планування і контролю робіт.

Як будь-яке керування, воно полягає у плануванні, координації, контролі, вимірі й обліку виконаних робіт у процесі розроблення програмного проєкту. Координацію людських, фінансових і технічних ресурсів виконує менеджер проєкту аналогічно до того, як це робиться в технічних проєктах. У його обов'язки входить дотримання запланованих бюджетних і часових характеристик і обмежень, стандартів і сформульованих вимог.

Загальні питання керування проєктом містяться в ядрі знань PMBOK [12], а також у стандарті ISO/IEC 12207 – Software life cycle processes, де керування проєктом розглядається як організаційний процес ЖЦ.

Область знань «Керування інженерією ПЗ (Software Engineering Management)» складається з таких розділів:

– організаційне керування (Organizational Management),
– керування процесом/проєктом (Process/Project Management),
– інженерія вимірювання ПЗ (Software Engineering Measurement).

Організаційне керування – це планування і складання графіка робіт, підбір і керування персоналом, контроль виконання й оцінка вартості робіт згідно з прийнятими стандартами і договорами з замовником. Головним об'єктом організаційного керування проєктом є персонал (навчання, мотивація й ін.), комунікації між співробітниками (зустрічі, презентації й ін.), а також попередження й усунення ризику невиконання проєкту. Для керування проєктом створюється спеціальна структура колективу. Фахівці розподіляються за видами робіт і розв'язують задачі проєкту під керуванням менеджера з урахуванням заданої вартості і термінів розробки. Для реалізації задач проєкту підбираються необхідні програмні, інструментальні й апаратні засоби.

Керування проєктом/процесом – це складання плану проєкту, побудова графіків робіт (мережних або часових діаграм) з урахуванням наявних ресурсів, розподіл персоналу за видами робіт у проєкті, виходячи з заданих термінів і вартості їх виконання. Для ефективного керування проєктом проводиться аналіз фінансової, технічної, операційної і соціальної політики організації-розробника для вибору правильної стратегії виконання робіт і контролю плану, а також проміжних продуктів (проєктних рішень, діаграм UML, алгоритмів і ін.).

У задачі керування проєктом входять також уточнення вимог, перевірка їх відповідності заданим специфікаціям характеристик якості, а також верифікація функцій проєкту. Процес керування базується на планових термінах, що відображені мережними діаграмами PERT (Program Evaluation and Review Technique), CPM (Critical Path Method). У них указуються роботи, зв'язки між ними і час виконання.

На сьогоднішній день найбільш поширена мережна діаграма PERT – граф, у вершинах якого розміщуються роботи, а дуги задають взаємні зв'язки між цими роботами. Інший тип мережної діаграми – CPM – є становим. У його вершинах указують події, а роботи задають лініями між двома вузлами-подіями. Очікуваний час виконання робіт за допомогою мережних діаграм оцінюється середнім ваговим значенням трьох оцінок: оптимістичної, песимістичної й очікуваної, тобто імовірнісної. Ці оцінки надають експерти, що враховують обсяги виконаної роботи і відведений на неї час.

Коректно складений план забезпечує виконання вимог і цілей проєкту. Контроль здійснюється при виконанні і внесенні змін у проєкт з урахуванням ризиків і прийнятих рішень щодо їх мінімізації.

Під *ризиком* розуміють імовірність виникнення несприятливих обставин, що можуть негативно вплинути на керування розробкою (наприклад, звільнення співробітника і відсутність заміни для продовження робіт і ін.). При складанні плану проєкту проводиться ідентифікація й аналіз ризику, планування непередбачених ситуацій щодо ризиків. Запобігання ризику полягає у виконанні дій, що знімають ризик (наприклад, збільшення часу розробки й ін.). Причиною появи ризику може бути реорганізація проєкту, БД або транзакцій, а також помилки при виконанні ПЗ.

Інженерія вимірювання ПЗ проводиться з метою визначення окремих характеристик продуктів і процесів (наприклад, кількість рядків у продукті, помилок у специфікаціях і т.п.). Попередньо проводяться роботи з вибору метрик процесів і продуктів з урахуванням обставин, що впливають на вимірювання характеристик програмного продукту.

Інженерії вимірювання – удосконалювання процесів керування проєктом; оцінювання часових витрат і вартості ПЗ, їх регулювання; визначення категорій ризиків і відстеження чинників для регулярного розрахунку ймовірностей їх виникнення; перевірка заданих у вимогах показників якості окремих продуктів і проєкту в цілому [9].

Проведення різного роду вимірювань – важливий принцип будь-якої інженерної діяльності. У програмному проєкті результати вимірювань необхідні замовнику і споживачу для встановлення правильності реалізації проєкта. Без вимірювань в інженерії ПЗ процес керування стає неефективним і перетворюється в самоціль.

6. Базовий процес програмної інженерії

Процес інженерії – є метарівнем, що визначає основні поняття, способи реалізації, оцінювання, вимірювання, дії з керування змінами й удосконалення самого процесу. Як уже згадувалася раніше, для оцінювання й удосконалення процесу програмної інженерії застосовується модель зрілості CMM [11], яку розроблено Інститутом програмної інженерії SEI (Software Engineering Institute) США. Ця модель встановлює рівні готовності організації-розробника ПЗ створювати задовільно, середньо, добре і дуже добре програмну продукцію. Поняття рівня готовності визначається наявністю в організації необхідних ресурсів (людських, програмних, технічних і фінансових), стандартів і методик, а також здатністю колективу створювати програмні продукти. Модель CMM має п'ять рівнів. Перший і другий рівні фіксують недостатню готовність виконувати розробку продукту. Третій – п'ятий рівні характеризують певний ступінь готовності, зрілості і здатності фахівців (а, значить, і організації) виготовляти, відповідно, середній, гарний і відмінний продукт. Чим вище рівень зрілості, тим більше вимог ставиться до процесу програмної інженерії, придатного для виконання цілей і задач утворення продукту, що задовольняє користувача.

Існують різновиди цієї моделі: CMM – SW (Software) для оцінки зрілості ПЗ, CMMI (CMM Integrated) – для обліку потреб великих державних структур в ПЗ (США), а також інші моделі, наприклад, Bootstrap – для оцінки зрілості малих і середніх комерційних компаній, стандарт ISO 15504 (Software Process Improvement and Capability) – для удосконалення процесу (наприклад, удосконалювати процес на другому рівні, щоб одержати сертифікат на третій рівень зрілості).

Концепція зрілості процесу програмної інженерії ґрунтується на процесі ПЗ (software process), широті його можливостей (software process capability), результативності (software process performance) і зрілості (software process maturity). Процес ПЗ у моделі CMM – це множина діяльностей (activities), методів (methods), практичних прийомів (practices), що використовують при розробки ПЗ шляхом планування робіт і оцінювання проміжних результатів, які приводять до кінцевого продукту високої якості.

Область знань «Процес програмної інженерії (Software Engineering Process)» складається з таких розділів:

– концепції процесу інженерії ПЗ (Software Engineering Process Concepts),
– інфраструктура процесу (Process Infrastructure),
– визначення процесу (Process Definition),
– оцінки процесу (Process Assessments),
– якісний аналіз процесу (Qualitative Process Analysis),
– виконання і змінювання процесу (Process Implementation and Change).

Концепції процесу інженерії ПЗ – задачі і дії, що зв'язані з керуванням, реалізацією, оцінкою, змінами й удосконаленням процесу і/або ПЗ. Ціль керування процесом – це створення інфраструктури процесу, виділення необхідних ресурсів, планування реалізації і зміни процесу з метою впровадження його у

практику і, нарешті, оцінка переваг від його впровадження у практику проєктування ПЗ.

Інфраструктура процесу містить у собі ресурси (людські, технічні, інформаційні і програмні), стандарти, методики керування якістю, проєктом і структуру колективу розробників ПЗ типу: команда, бригада, експериментальна фабрика (Experimental Factory), каркас виробництва на лінії продуктів (Framework for Product Line Practice) і ін. До основних задач інфраструктури належать керування і комунікації в колективі, інженерні методи виробництва програмного продукту й удосконалення процесу з накопиченим досвідом розробки ПЗ.

Визначення процесу ґрунтується на: типах процесів і моделей (каскадна, спіральна, ітераційна й ін.); моделях ЖЦ процесів і засобів, стандартах ЖЦ ПЗ ISO/IEC 12207 і ISO/IEC 15504, IEEE std. 1074 і IEEE std. 1219; методах і нотаціях подання процесів і автоматизованих засобів їх підтримки. Основною метою процесу є підвищення якості одержуваного продукту, поліпшення різних аспектів програмної інженерії, автоматизація і удосконалення процесів.

Оцінка процесу проводиться з використанням відповідних моделей і методів оцінки. Наприклад, оцінка потенційної здатності фахівця до розроблення і виконання відповідного процесу, а також оцінювання зрілості процесу, згідно за яким проводиться розроблення ПЗ.

Оцінки стосуються також технічних робіт у сфері програмної інженерії, керування персоналом і якості ПЗ. Для цього проводяться експериментальні дослідження середовища, збирання інформації, моделювання, класифікація отриманих помилок і дефектів, а також статичний аналіз недоліків процесу порівняно з існуючими стандартами (наприклад, ISO/IEC 12207) і потенційних аспектів необхідності вдосконалювати процес.

Якісний аналіз процесу полягає в ідентифікації і пошуку «слабких місць» у процесі створення ПЗ на початку його функціонування і після експлуатації. Розглядається такі техніки аналізу: огляд даних і порівняння процесу з основними положеннями стандарту ISO/IEC 12207, збирання даних про якість процесів; аналіз головних причин відмов у функціонуванні ПЗ, відкіт назад від точки виникнення відхилення до точки правильної роботи системи для з'ясування причин зміни процесу. На якість результатів проєкту і процесу впливають застосовувані інструменти і досвід фахівців.

Виконання і зміна процесу. Існує ряд фундаментальних аспектів вимірювань в програмній інженерії, що покладені в основу детальних вимірювань процесу. Оцінка вдосконалення процесу проводиться шляхом встановлення кількісних характеристик процесу і продуктів. Після процесу розгортання ПЗ виконуються обчислення функцій і аналіз отриманих результатів, які можуть застосовуватися при оцінюванні якості, продуктивності, трудовитрат та ін. Якщо результати не задовольняють користувача ПЗ, проводять обговорення і приймають рішення щодо необхідності виправлення ситуації шляхом або внесення зміни у процес, або вдосконалення процесу, а також організаційну структуру і деякі інструменти керування змінами.

7. Методи і інструменти програмної інженерії

Методи забезпечують проєктування, реалізацію і виконання ПЗ. Вони накладають деякі обмеження на інженерію ПЗ у зв'язку з особливостями застосування їхніх нотацій і процедур, а також забезпечують оцінку і перевірку процесів і продуктів. Інструменти забезпечують програмну підтримку окремих методів інженерії ПЗ для автоматизованого виконання задач процесів ЖЦ.

Область знань «Методи та інструменти інженерії ПЗ (Software Engineering Tools and Methods)» складається з розділів:

– інструменти інженерії ПЗ (Software Engineering Tools),

– методи інженерії ПЗ (Software Engineering Methods).

Методи інженерії ПЗ – це евристичні методи (heuristic methods), формальні методи (formal methods) і методи прототипування (prototyping methods).

Евристичні методи містять у собі: структурні методи, засновані на функціональній парадигмі; методи, орієнтовані на структури даних, якими маніпулює ПЗ; об’єктно-орієнтовані методи, що розглядають предметну область як колекцію об'єктів; методи, орієнтовані на конкретну область застосування, наприклад, на системи реального часу, безпеки та ін.

Формальні методи засновані на формальних специфікаціях, аналізі, доведенні і верифікації програм. Специфікація записується мовою, синтаксис і семантика якої визначені формально і засновані на математичних концепціях (алгебрі, теорії множин, логіці). Розрізняються наступні категорії формальних методів:

– *мови і нотації специфікації* (specification languages and notations), орієнтовані на модель, властивості і поведінку;

– *уточнення специфікації* (refinement specification) шляхом трансформації в кінцевий результат, близький до кінцевого програмного продукту, що виконується;

– *методи верифікації/доведення* (verification/proving properties), що використовують твердження (теореми), перед- і постумови, формально описуються і застосовуються для встановлення правильності специфікації програм.

Методи доведення застосовувалися в основному в теоретичних експериментах. Понад 25 років їх застосування було обмежено через трудомісткість і економічну невигідність. У 2005 р. проблема верифікації знову набула актуальності у запропонованому новому міжнародному проєкті «Цілісний автоматизований набір інструментів для перевірки коректності ПС» (Т. Хоар, «Открытые системы», 2006, № 6), який поставив наступні перспективні задачі:

– розробка єдиної теорії побудови й аналізу програм;

– побудова багатостороннього інтегрованого набору інструментів верифікації на усіх виробничих процесах – розроблення формальних специфікацій, їх доведення і перевірка правильності, генерація програм і тестових прикладів, уточнення, аналіз і оцінка;

– створення репозиторія формальних специфікацій, верифікованих програмних об'єктів різних типів і видів.

Формальні методи верифікації будуть охоплювати всі аспекти створення і перевірки правильності програм. Це приведе до створення потужної верифікованої виробничої основи і сприятиме значному зменшенню помилок у ПЗ.

Методи прототипування (Prototyping Methods) засновані на використанні прототипу ПЗ для моделювання на ньому завдань нової системи і базуються на:

– стилях прототипування, що уособлюють тривалість використання прототипів, наприклад, стиль створення тимчасово використовуваних прототипів (throw away),

– моделях еволюційного прототипування – перетворення прототипу в кінцевий продукт і розроблення специфікацій, відповідно до якої він виконується;
– техніках оцінки/дослідження (evaluation) результатів прототипування.

Інструменти інженерії ПЗ забезпечують автоматизовану підтримку процесів розроблення ПЗ і містять у собі множину інструментів, що охоплюють усі процеси ЖЦ.

Інструменти роботи з вимогами (Software Requirements Tools) – це:

– інструменти розробки (Requirement Development) і керування вимогами (Requirement Management), орієнтовані на аналіз, збирання, специфікування і перевірку вимог;

– інструменти трасування вимог (Requirement traceability tools) є невід'ємною частиною роботи з вимогами, їх функціональний зміст залежить від складності проєктів і рівня зрілості процесів.

Інструменти проєктування (Software Design Tools) – це інструменти для створення ПЗ із застосуванням базових нотацій (структурної SADT/IDEF, моделювання UML тощо).

Інструменти конструювання ПЗ (Software Construction Tools) – це інструменти для трансляції і об'єднання програм. До них належать:

– редактори програм (program editors) і програми редагування загального призначення;

– компілятори і генератори коду (compilers and code generators) як самостійні засоби об'єднання програмних компонентів в інтегрованому середовищі для одержання вихідного продукту з використанням препроцесорів, складальників, завантажників тощо;

– інтерпретатори (interpreters), які забезпечують контрольоване виконання програм за їх описом. Намітилася тенденція злиття інтерпретаторів і компіляторів (наприклад, Java, в .NET);

– відлагоджувачі (debuggers), призначені для перевірки правильності опису вихідних програм і усунення помилок;

– інтегроване середовище розробки (IDE – integrated development environment) та бібліотеки компонентів (libraries components), що є утворюють середовище виконання процесу розроблення ПС;

– програмні платформи (Java, J2EE і Microsoft .NET) і платформи для розподілених обчислень (CORBA і WebServices тощо).

Інструменти тестування (Software Testing Tools) – це:

– генератори тестів (test generators), що допомагають у розробці сценаріїв тестування;

– засоби виконання тестів (test execution frameworks), які забезпечують виконання тестових сценаріїв і відслідковують поведінку об'єктів тестування;

– інструменти оцінки тестів (test evaluation tools), які підтримують оцінювання результатів виконання тестів і ступеня відповідності поведінки тестованого об'єкта очікуваній поведінки;

– засоби керування тестами (test management tools), які забезпечують інженерне керування процесом тестування ПЗ;

– інструменти аналізу продуктивності (performance analysis tools), кількісної її оцінки та оцінки поводження програм у процесі виконання.

Інструменти супроводу (Software Maintenance Tools) містять у собі:

– інструменти полегшення розуміння (comprehension tools) програм, наприклад, різні засоби візуалізації;

– інструменти реінженерії (reengineering tools) підтримують діяльність з перетворення програм і зворотної інженерії (reverse engineering) для відновлення (артефактів, специфікації, архітектури) застарілого ПЗ або генерації нового продукту.

Інструменти конфігураційного керування (Software Configuration Management Tools) – це:

– інструменти відстеження (tracking) дефектів;

– інструменти керування версіями;

– інструменти керування складанням, випуском версії (конфігурації) продукту та його інсталяції.

Інструменти керування інженерною діяльністю (Software Engineering Management Tools) підрозділяються на:

– інструменти планування і відстеження ходу проєктів, кількісної оцінки зусиль і вартості робіт у проєкті;

– інструменти керування ризиками, які використовуються для ідентифікації, моніторингу ризиків і оцінки нанесеного ушкодження;

– інструменти кількісної оцінки властивостей ПЗ шляхом вимірювань і розрахунків остаточного значення надійності і якості.

Інструменти підтримки процесів (Software Engineering Process Tools) розділені на:

– інструменти моделювання та опису моделей ПЗ (наприклад, UML і його інструменти);

– інструменти керування програмними проєктами;

– інструменти керування конфігурацією для підтримки версій і всіх артефактів проєкту.

Інструменти забезпечення якості (Software Quality Tools) діляться на дві категорій:

– інструменти інспектування для підтримки перегляду (review) і аудиту;

– інструменти статичного аналізу артефактів, даних, потоків робіт і перевірки їх властивостей на відповідність показникам.

Додаткові аспекти інструментального забезпечення (Miscellaneous Tool Issues) стосуються:

– техніки інтеграції інструментів (платформ, представлень, процесів, даних) для їх природного сполучення в інтегрованому середовищі;

– метаінструментів для генерації інших інструментів для ПЗ;

– оцінки інструментів при їх еволюції.

8. Якість програмного забезпечення

Якість ПЗ – набір властивостей продукту (сервісу або програм), що характеризують його здатність задовольнити встановлені або передбачувані потреби замовника. Поняття якості має різні інтерпретації залежно від конкретної програмної системи і вимог до неї. Крім того, у різних джерелах таксономія (класифікація) характеристик у моделі якості розрізняється.

Моделі мають різну кількість рівнів і повністю або частково збігаються щодо набору характеристик якості. Наприклад, модель якості МакКолла на найвищому рівні має три характеристики: функціональність, модифікованість і переносність, а на нижчих рівнях моделі – 11 підхарактеристик якості і 18 критеріїв (атрибутів) якості.

Стандарт ISO 9126:2001 регламентує *зовнішні і внутрішні характеристики* якості. Перші відображають вимоги до функціонування програмного продукту. Для кількісного встановлення критеріїв якості, за якими буде здійснюватися перевірка і підтвердження відповідності ПЗ заданим вимогам, визначаються відповідні зовнішні вимірювані властивості (зовнішні атрибути) ПЗ, метрики (наприклад, час виконання окремих компонентів), діапазони зміни значень і моделі їх оцінки. Метрики використовуються на стадії тестування або функціонування і називаються *зовнішніми метриками.* Вони являють собою моделі оцінки атрибутів.

Внутрішні характеристики якості і внутрішні атрибути ПЗ використовуються для складання плану досягнення необхідних зовнішніх характеристик якості продукту. Для квантифікації внутрішніх характеристик якості застосовують внутрішні метрики, як інструмент перевірки відповідності проміжних продуктів внутрішнім вимогам до якості, які формулюються на процесах, що передують тестуванню.

Зовнішні і внутрішні характеристики якості відображають властивості самого ПЗ (працюючого або не працюючого), а також погляд замовника і розробника на таке ПЗ. Безпосереднього кінцевого користувача ПЗ цікавить експлуатаційна якість ПЗ – сукупний ефект від досягнення характеристик якості, що виміряється строком результату, а не властивістю самого ПЗ. Це поняття ширше, ніж будь-яка окрема характеристика (наприклад, зручність використання або надійність).

Остаточна оцінка якості проводиться відповідно до стандарту ISO/IEC 14598. Якість може підвищуватися за рахунок постійного поліпшення використовуваного продукту виявленням, усуненням дефектів у ПЗ і їх запобіганням.

Область знань «Якість ПЗ (Software Quality)» складається з наступних розділів:

– концепції якості ПЗ (Software Quality Concepts);

– визначення і планування якості (Definition & Planning for Quality);

– техніки й види діяльності, що забезпечують гарантію якості, валідацію і верифікацію (Activities and Techniques for Software Quality Assurance, Validation & Verification –V&V);

– вимірювання при аналізі якості ПЗ (Measurement in Software Quality Analysis).

Концепції якості ПЗ – це зовнішні і внутрішні характеристики якості, їхні метрики, а також моделі якості, визначені на множині цих характеристик, що наведені в стандартах з якості і в [8, 9] – це шість характеристик і кожна з них має кілька атрибутів. До характеристик якості належать:

– функціональність (functionality);

– надійність (realibility);

– зручність застосування (usability);

– ефективність (efficiency);

– супровід (maitainnability);

– переносність (portability).

Базова модель якості містить у собі ці характеристики і вони притаманні будь-якому типу програмних продуктів. При розробці вимог замовник формулює такі вимоги до якості, які найбільшою мірою підходять для програмного продукту, який замовляється.

Визначення і планування якості ПЗ ґрунтується на положеннях стандартів у цій області, складанні планів і графіків робіт, процедурах перевірки і ін. План забезпечення якості містить у собі набір дій для перевірки процесів забезпечення якості (верифікація, валідація і ін.) і формування документа з керування якістю.

Планування якості призначено для підтримки керування процесами досягнення якості продуктів проєкту (зокрема проміжних робочих) і ресурсів – програмних, технічних, виконавських і ін. Воно також передбачає керування вимогами до процесів і продуктів і полягає в наступному:

– визначення продукту термінами заданими характеристиками якості;

– планування процесів для гарантії одержання необхідної якості;

– вибір методів оцінки запланованих характеристик якості і встановлення відповідності продукту сформульованим вимогам.

У стандарті ISO/IEC 12207 визначені спеціальні процеси забезпечення якості – верифікація, валідація (атестація), спільний аналіз і аудит.

Види діяльності і техніки гарантії якості містять у собі, зокрема: інспекцію, верифікацію і валідацію ПЗ.

Інспекція ПЗ – аналіз і перевірка різних видів подання системи і ПЗ (специфікації, архітектурної схеми, діаграм, початкового коду тощо). Виконується на всіх процесах ЖЦ розробки ПЗ.

Верифікація ПЗ – процес забезпечення правильної реалізації ПЗ відповідно до специфікацій, виконується протягом усього життєвого циклу. Верифікація дає відповідь на питання, чи правильно створюється система.

Валідація – процес перевірки відповідності ПЗ функціональним і нефункціональним вимог і очікуваним потребам замовника.

Верифікація і валідація (V&V) можуть виконуватися, починаючи з ранніх стадій ЖЦ. Вони орієнтовані на отримання правильних функцій ПЗ, плануються і забезпечуються визначеними ресурсами з чітким розподілом ролей. Перевірка ґрунтується на використанні відповідних технік тестування для виявлення тих або інших дефектів і збирання статистики. Після зібрання даних оцінюється правильність реалізації вимог і роботи ПЗ у заданих умовах.

Вимірювання при аналізі якості ПЗ ґрунтується на метриках продукту і даних, зібраних у процесі створення продукту при заданих ресурсах: оцінок процесів, ПЗ і його моделей, і передбачає документування вимірів. Оцінювання якості продукту полягає у вимірюванні і оцінюванні якісних показників за допомогою даних про різні типи помилок і відмов під час тестування ПЗ і виконання коду на тестових даних. Ці дані аналізуються, перевіряються і використовуються при якісній і кількісній оцінки ПЗ.

Для імітації роботи системи в режимі тестування розробляються тести з реальними вхідними даними для перевірки правильності роботи ПЗ на різних фрагментах програми і шляхах проходження в них операторів. У процесі тестування ПЗ виявляються різного роду помилки (відмови, дефекти, помилки тощо), кількість яких значною мірою може вплинути на одержання правильного і якісного результату.

З урахуванням типів виявлених помилок можна встановити наявність (або відсутність) відповідності реалізованих і нереалізованих функцій, заданих у вимогах до системи, а також оцінити способи реалізації нефункціональних вимог (продуктивності, надійності та ін.). Оцінюються процеси керування планами, інспекціями, прогонами і т.п. За цими оцінками приймаються рішення про завершення розробки продукту проєкту і передачі його замовнику в експлуатацію, під час якої можуть бути внесені зміни щодо усунення помилок, визначення адекватності планів і вимог, оцінки ризиків переroблення ПЗ тощо.

Мета інспекцій – виявлення різних аномальних станів у ПЗ незалежними фахівцями команди експертів із залученням авторів проміжного або кінцевого продукту. Експерти інспектують виконання вимог, інтерфейси, вхідні дані і т.п., а потім документують виявлені відхилення в проєкті.

Призначення аудита – незалежна оцінка продуктів і процесів на відповідність регулюючим і регламентуючим документам (планам, стандартам і ін.), формулювання звіту про випадки невідповідності і пропозицій для їх коригування.

Таким чином, розгляд розділів SWEBOK свідчить по те, що це ядро містить весь необхідний набір знань з програмної інженерії, який повинні мати фахівці різних профілів (аналітики, інженери, програмісти, оцінювачі, контролери тощо), що виробляють програмний продукт.

Зазначимо, що ядро знань SWEBOK не позбавлено недоліків тактичного характеру. Так, між областями знань у цьому ядрі існують перетини за методами, концепціями і стратегіями, а деякі важливі напрями програмної інженерії взагалі не відбиті у ньому наявними областями знань. Це стосується, наприклад, методів доведення правильності програм, еволюції програм, розподілених і неоднорідних середовищ, взаємодії систем, таких методів програмування, як аспектне, агентне, сервісне й інші, а також аспектів захисту, безпеки тощо.

Контрольні запитання

7. Назвіть основні задачі області знань «Проєктування ПЗ»
8. Визначте мету і задачі області знань «Методи і інструменти»
9. Наведіть базові поняття області знань «Тестування ПЗ».
10. Визначте мету і задачі області знань «Керування проєктом».
11. Наведіть приклади інструментів програмної інженерії.
12. Визначте мету і задачі області знань «Інженерія якості ПЗ».
13. Вкажіть, який зв'язок існує між ядром знань SWEBOK і стандартом ЖЦ.
14. Охарактеризуйте інфраструктуру програмної інженерії.

Завдання

Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з особливостями проєктування ПЗ, процесу визначення архітектури, набору компонентів, їх інтерфейсів, інших характеристик системи і кінцевого складу програмного продукту.

Дати відповіді на контрольні запитання.

Самостійна робота 3
DevOps

Теоретичні відомості

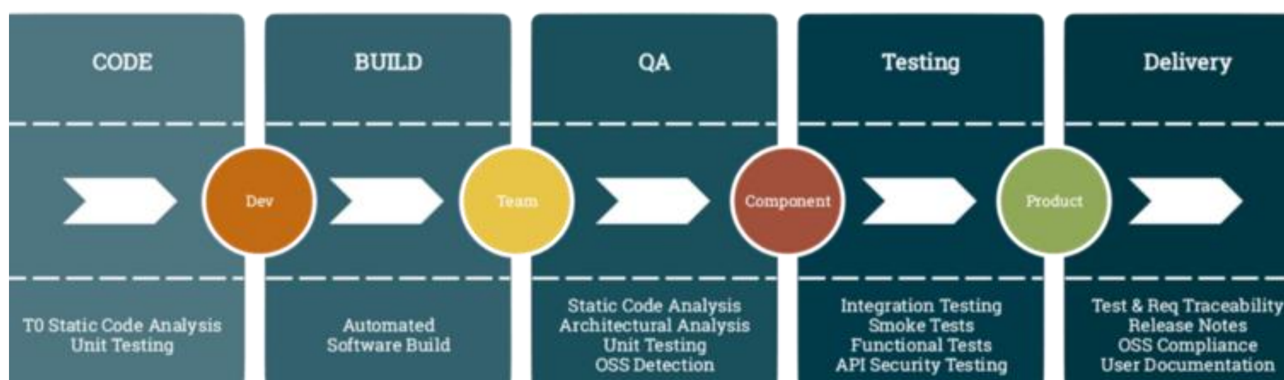
Швидкий розвиток і швидкий вихід на ринок є двома ключовими факторами, що визначають успіх будь-якої фірми в сучасній ІТ-індустрії.

Крім того, існують деякі інші проблеми, з якими сьогодні стикається більшість компаній у софтверній індустрії [5]:

– відсутність культурного балансу та зв'язку між розробкою програмного забезпечення та експлуатацією (software Development and Operations);
– відсутність гнучкої доставки для стимулювання інноваційності програмного забезпечення;
– подальше існування розрізненості між командами розробки та експлуатації;
– незалучення адміністраторів баз даних (DBA) до циклів релізу;
– відсутність прийняття цілісного уявлення про весь ланцюжок створення вартості для безперервної доставки програмного забезпечення.

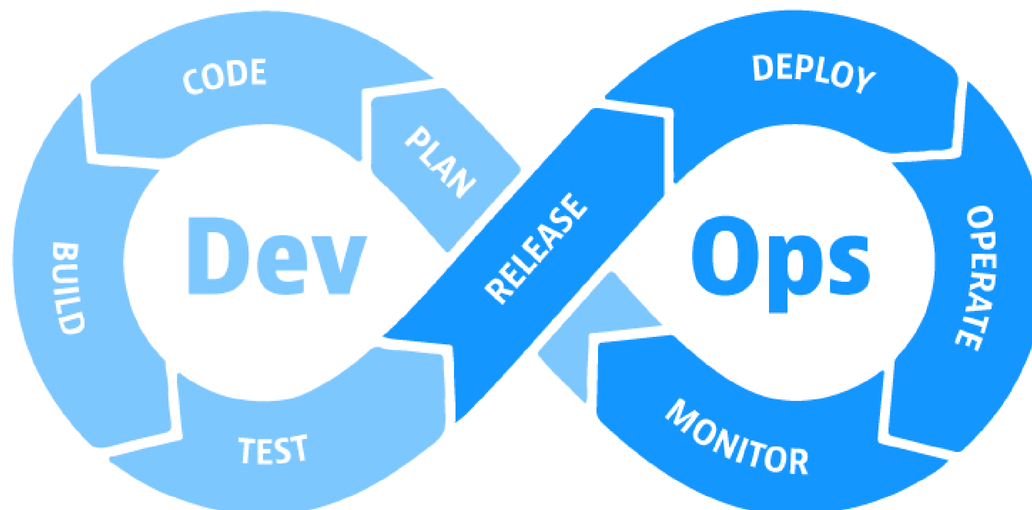
DevOps може вирішити ці проблеми. DevOps як послуга рекомендує нову «культуру», яка ідеально заповнює розриви між основними командами, наприклад, розробкою (Dev) та операційною (Ops), для покращення співпраці та підвищення продуктивності.

DevOps – це підхід, коли інженери-розробники та інженери-адміністратори разом беруть участь у всьому життєвому циклі програмного продукту, від проєктування та розробки до ретельної підтримки продукту. Тобто DevOps покликана усунути традиційну роз'єднаність, де одна команда пише код, інша його тестує, третя – розгортає, а четверта відповідає за експлуатацію. DevOps дозволяє розглядати розробку, доставку та експлуатацію ПЗ в єдиному контексті.



[5] DevOps Consulting Services. Application Delivery and Operational Efficiency at High Velocity. URL: https://www.veritis.com/solutions/devops/

В DevOps системна інженерія вибудовується як потік завдань розробки. Усі ресурси заносяться до системи обліку і покриваються відповідними тестами.



Простежується кілька ключових DevOps-тем: цінності, принципи, методи, практики та інструменти[6].

Цінності. Будь-який розробник зосереджений на пошуку рішення, і така спрямованість часом виливається в неприйняття нових технологій, небажання експериментувати з новими речами, що виражається по-різному: від синдрому «неприйняття чужої розробки» до контрпродуктивних спроб захищати свою нішу. Щоб по-справжньому перейти на DevOps, ці упередження слід спочатку визнати, а потім подолати. Жодна технологія, ні Docker, Kubernetes або Amazon Web Services не вирішить ваших проблем, якщо ви не розумієте, у чому полягає ціннісна пропозиція.

Принципи компанії бажано вибудовувати так, щоб це було середовище, в якому стимулюється системне мислення, посилюються цикли зворотного зв'язку, прищеплюється культура безперервного експериментування та навчання. Варто налагодити більше циклів зворотного зв'язку. Моніторинг, метрики та логування – ось три цикли, які допомагають адміністраторам брати участь у проєктуванні. У здоровому DevOps-середовищі стимулюються процеси, що сприяють створенню коротких та ефективних циклів зворотного зв'язку, приклади таких процесів – управління інцидентами, об'єктивний аналіз, забезпечення прозорості…

Методи

1. *Гнучке управління* полягає в поділі проєкту на невеликі ділянки роботи, їх збирання з обмеженням незавершеності роботи (progress limit), впровадження циклів зворотного зв'язку та візуалізація. Гнучкі прийоми управління дають результативніший вихід, у тому числі, покращують пропускну здатність і стабільність системи; співробітники відчувають на роботі не такий сильний стрес, одержують більше задоволення від роботи.

[6] DevOps: що ж це таке насправді. URL: https://habr.com/ru/company/piter/blog/430508/

2. Одна з перших методологій, запропонованих основоположниками DevOps, формулюється як «*спочатку люди, потім процеси, потім інструменти*». Рекомендується насамперед домовитися про те, хто відповідає за конкретне робоче завдання. Потім визначити, які процеси потрібні для вирішення цього завдання. Після цього підбирається інструментарій, необхідний реалізації процесу. Все це здається логічним, проте, інженери та менеджери часто піддаються на заклики «поспішайте придбати!» від постачальників інструментів, а тоді намагаються під куплений інструмент формувати весь потік завдань.

3. *Безперервна доставка*. Цей термін настільки у всіх на вустах, що його навіть помилково прирівнюють до DevOps. Це практика динамічного програмування і тестування ПЗ, що забезпечує швидкі релізи дрібних повноцінних готових фрагментів. Загалом безперервна доставка може підвищити загальну якість та збільшити швидкість роботи. Безперервна доставка – ключова складова проєкту, яку необхідно налагодити якомога раніше, оскільки вона є рушійним фактором успішного впровадження DevOps.

4. *Управління змінами.* Є пряма кореляція між тим, наскільки успішно експлуатується система і як організоване управління змінами. Це не означає, що потрібно впроваджувати традиційний контроль, який уповільнює розробку і, швидше, шкодить, ніж допомагає. Для цього необхідна платформа для безперервної доставки.

5. *Інфраструктура в коді*. Системні специфікації вносяться до систем контролю версій і рецензуються колегами. Застосовуючи сучасні механізми розгортання, зокрема Docker та Kubernetes, можна автоматично збирати, тестувати і створювати реальні системи на основі специфікації та програмно керувати ними.

Практики. Якщо в IT-організації підхід до проєктів «давайте щось напишемо… а потім доручимо комусь це тестувати та розгортати», то такий метод погано узгоджується з планами. Терміни зсуваються, а коли команда розробників переходить до наступного проєкту, експлуатаційні витрати стають непідйомними. Краще, щоб розробники організації продовжували тримати руку на пульсі створеного ними сервісу та частково відповідали за його експлуатацію. Тоді виходять ефективніші цикли зворотного зв'язку, що допомагають команді набагато оперативніше реагувати не тільки на баги, а й на нові фічі, що забезпечує розвиток продукту у правильному напрямку.

Інструменти допомагають інженеру програмувати, збирати, тестувати, компанувати, випускати, конфігурувати та відстежувати як системи, так і програми. До того як почала розвиватися парадигма DevOps, інновації та інструментарій перебували в стагнації. Розробники користувалися одним і тим самим інструментарієм роками. Багато інструментів, що застосовуються в DevOps, дуже багатофункціональні і допомагають абсолютно по-новому організувати життєвий цикл сервісу. Необхідно визначитися із надійним інструментарієм для DevOps. Немає єдиного інструменту на всі випадки життя, потрібна ціла лінійка інструментів, які можна комбінувати з урахуванням наявних потреб. Потрібно вибирати такі інструменти, які добре поєднуються з іншими. Інструменти по-

винні допомагати автоматизувати будь-яку роботу. Їх має бути легко викликати з API або командного рядка. При цьому інструменти, які сильно залежать від UI, не дуже вписуються навіть у добре інтегрований інструментарій.

Контрольні запитання

1. Що таке DevOps?
2. У чому потреба DevOps?
3. В чому різниця між DevOps та DevSecOps?
4. Назвати основні методи DevOps.
5. Навіщо запроваджувати DevOps?
6. На які ключові кроки потрібно звернути увагу при впровадженні DevOps?
7. Назвати основні інструменти DevOps.
8. За якими ознаками можна судити про успішність реалізації DevOps?
9. Які бізнес-переваги DevOps?

Завдання

Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними методами та інструментами DevOps.

Дати відповіді на контрольні запитання.

Самостійна робота 4
DevSecOps

Теоретичні відомості

1 Філософія інтеграції методів безпеки у процес DevOps

Останнім часом впровадження культури DevOps у роботу призвело до значних змін процесу обробки даних. Для організацій методи DevOps призводять до отримання низки безперечних переваг, а саме: гнучкість, оперативність, скорочення витрат, безсерверні обчислення, динамічний провіжиніг та оплата в міру використання/оплата за фактом надання послуги[7].

Незважаючи на величезну популярність, одного DevOps недостатньо у разі потреби безпечної доставки коду. Це призвело до розвитку нового підходу (відомого як **DevSecOps**), що інтегрує методи безпеки з DevOps.

У чому потреба DevSecOps?

DevOps дозволив розробляти кастомізоване програмне забезпечення для бізнесу за значно коротші терміни за рахунок об'єднання з командою розробки та управління. Однак, у більшості випадків, питання безпеки не є основним пріоритетом і часто вважається перешкодою на шляху до стрімкого розвитку.

Хоча нині компанії зосереджені на подоланні традиційних бар'єрів між командами розробки, тестування та управління, багато хто з них досі приділяє недостатньо уваги питанню інтеграції безпеки в процес розробки, що робить їх особливо уразливими до ризику виникнення загроз.

Тут допоможе метод/механізм DevSecOps, який впроваджує безпеку, створюючи різні засоби захисту для DevOps. Завдяки постійному моніторингу, оцінці та аналізу процесу, DevSecOps гарантує виявлення можливих неполадок та слабких місць вже на ранній стадії процесу розробки та їх миттєве усунення.

Відмінності між DevOps та DevSecOps

DevOps налагоджує ефективну взаємодію між командами розробки, тестування та управління з метою забезпечення безперервної і стабільної доставки застосунків. А DevSecOps, своєю чергою, інтегрує компонент безпеки у процеси DevOps.

DevSecOps здебільшого фокусується на вирішенні проблем, пов'язаних із забезпеченням безпеки автоматизованих процесів DevOps, таких як керування налаштуваннями, аналіз композиції програмного забезпечення та ін.

---

[7] DevSecOps Consulting Services. Integrating ‘Security as Code’ Culture within DevOps. URL: https://www.veritis.com/solutions/devops/devsecops-services/

Що таке DevSecOps?

DevOps широко відомий як набір функцій та інструментів для спрощення взаємодії між командами розробників програмного забезпечення та інфраструктури. Це, у свою чергу, забезпечує процес швидкої та надійної доставки застосунків та послуг між організаціями.

DevOps включає кілька основних сфер роботи: автоматизований провіжинінг, постійна інтеграція, стабільний моніторинг та розробка на базі тестування.

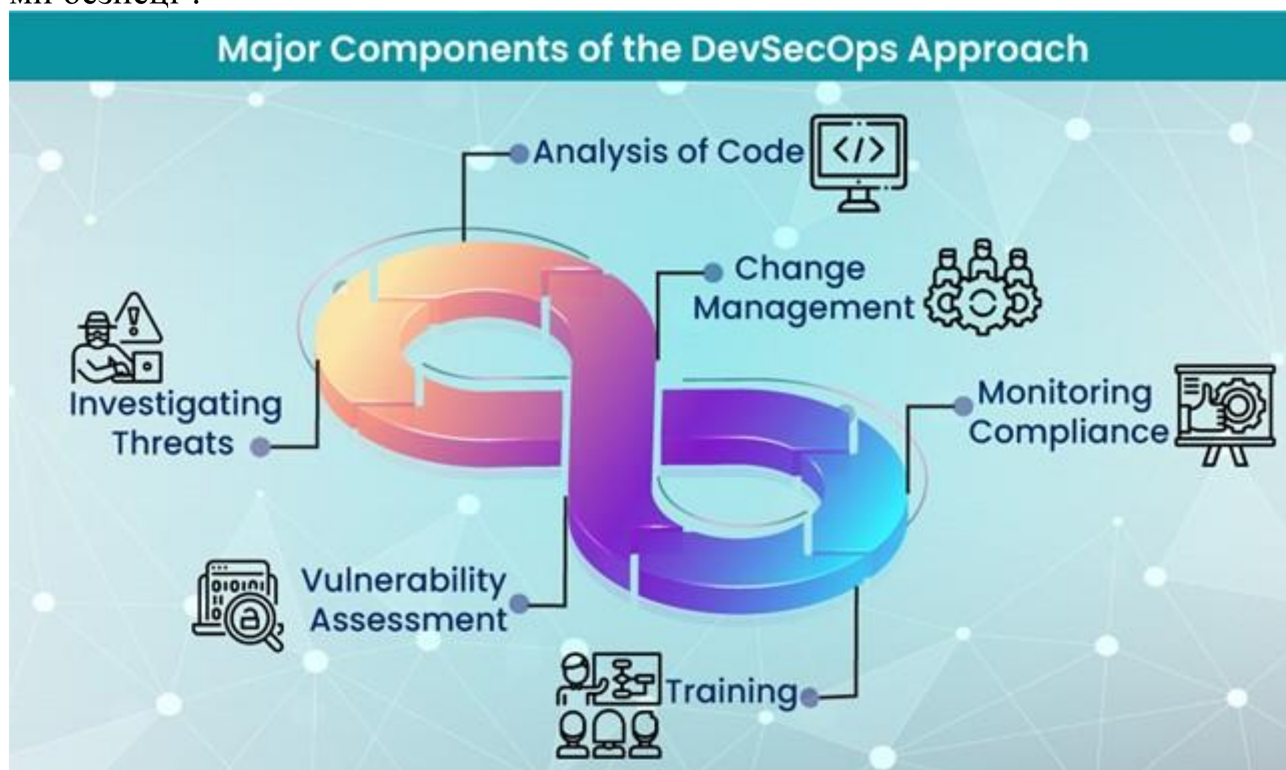
Будучи розширенням стратегії DevOps, DevSecOps впроваджує інструменти керування безпекою у робочий процес DevOps та автоматизує основні процеси, пов'язані із забезпеченням безпеки. Принципи безпеки запроваджуються на перших етапах процесу розробки та застосовуються протягом усього циклу розробки/виробництва.

На додаток до інтеграції безпеки в DevOps культуру, DevSecOps володіє унікальним набором інформації щодо розвитку програми та сприяють продуктивній співпраці між командами.

У середовищі IT немає одностайної думки з приводу назви DevSecOps, іноді зустрічаються **DevOpsSec**, **SecDevOps** або **Rugged DevOps**.

2. Основні компоненти DevSecOps

Компаніям критично важливо здійснити технічний та культурний зсув, взаємодіючи з DevSecOps, щоб ефективно справлятися з актуальними загрозами безпеці[8].



[8] Консалтинговые услуги DevSecOps. URL: https://habr.com/ru/company/nixys/blog/578890/

На практиці підхід DevSecOps охоплює шість основних компонентів:

– **Аналіз коду** забезпечує швидке виявлення уразливих місць за рахунок доставки коду невеликими порціями;

– **Управління змінами** – дозволяє користувачам не тільки вносити зміни, які можуть підвищити швидкість та ефективність процесів, а також оцінити якість їхнього впливу (позитивну чи негативну);

– **Моніторинг відповідності** – компаніям необхідно дотримуватись правил із Загального регламенту захисту даних (GDPR) та Стандарту безпеки даних індустрії платіжних карток (PCI DSS) і бути завжди напоготові до аудитів будь-якого характеру;

– **Аналіз загроз** – потенційно можливі загрози супроводжують кожне оновлення коду. Дуже важливо вміти якнайшвидше виявити їх і миттєво вжити необхідних заходів;

– **Аналіз уразливостей** – включає виявлення нових уразливостей і реагування на них;

– **Навчання** – компаніям необхідно залучати програмних та IT-інженерів до проведення тренінгів на тему забезпечення безпеки та забезпечувати їх основними принципами та інструкціями.

3. Впровадження стратегії DevSecOps

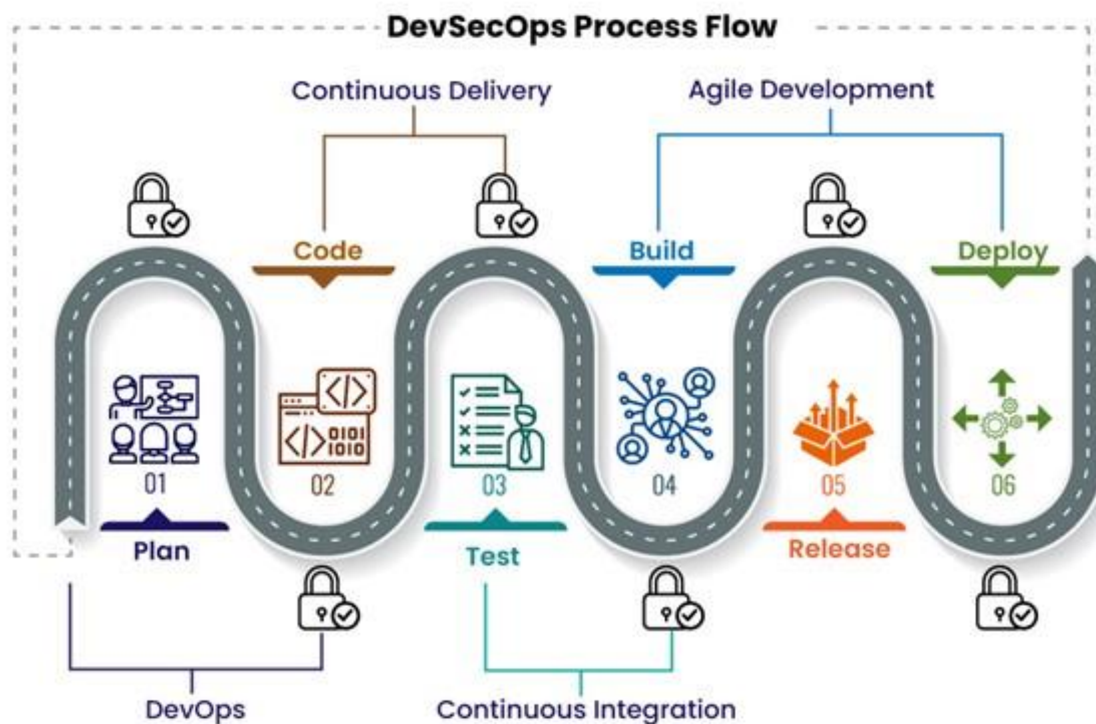
Перехід від **DevOps** до **DevSecOps** – не найпростіше завдання, але цілком здійсненне за наявності гарного плану дій.

Існують три ключові кроки, на які потрібно звернути увагу при впровадженні DevSecOps:

– **Оцінка поточних заходів безпеки** – служби безпеки здійснюють прогнозування загроз та оцінку ризиків, що допомагає з'ясувати реальний рівень захисту поточних даних. Плюс до всього це допомагає проаналізувати систему безпеки і виявити те, які модифікації терміново необхідні, а які можна відкласти;

– **Інтеграція заходів безпеки в DevOps** – інтеграція заходів безпеки в процес розробки охоплює аналіз флоу роботи і забезпечення мінімальних збоїв, завдяки впровадженню методів забезпечення безпеки та автоматизації;

– **Інтеграція DevSecOps з системою безпеки** – інтеграція DevSecOps може вважатися успішною, тільки якщо команди розробників, безпеки та управління прагнуть налагодити загальний процес роботи і впровадити нові інструменти забезпечення безпеки в абсолютно всі процеси DevOps. Безперервний моніторинг будь-яких проблем безпеки під час розробки і забезпечення швидкого реагування є критично важливими в інтеграції операцій безпеки DevSecOps.



4. Основні інструменти DevSecOps

Впровадження DevSecOps передбачає оцінку всіх можливих ризиків системи безпеки програми і при тестуванні коду, для чого необхідне володіння спеціальними інструментами.

Використання інструментів автоматизованого тестування в інтегрованому середовищі розробки (IDE) дозволяє розробникам впроваджувати елемент безпеки в робочі процеси DevOps, уникаючи потреби щоразу запускати нове середовище для тестування коду.

Існує кілька інструментів, розроблених для спрощення процесу інтегрування DevSecOps:

– **Інструменти візуалізації**: такі інструменти, як Kibana та Grafana, допомагають ідентифікувати, розвивати та обмінюватися інформацією про безпеку;

– **Інструменти автоматизації**: такі інструменти, як StackStorm, допомагають у виправленні несправностей системи безпеки за заздалегідь заготовленим шаблоном/сценарієм;

– **Hunting інструменти**: такі інструменти допомагають у виявленні аномалій безпеки. Ось деякі з них: Mirador, OSSEC, MozDef та GRR;

– **Інструменти тестування**: тестування є найважливішим елементом DevSecOps, для цієї мети використовується широкий спектр інструментів, якот: GauntIt, Spyk, Chef Inspect, Hakiri, Infer та Lynis;

– **Засоби оповіщення**: такі інструменти, як Elastalert, Alerta і 411, оповіщають про виявлення неполадок у системі безпеки, що потребують виправлення;

– **Засоби розпізнавання загроз**: такі інструменти збирають та зіставляють дані про загрози (OpenTPX, Critical Stack, Passive Total);

– **Інструменти моделювання атак**: допомагають у реалізації моделювання атак та засобів захисту.

DevSecOps – головний пріоритет бізнесу

Елемент безпеки в DevOps – це не виключно технічний інструмент, він також стосується діяльності людей та налагодженого функціонування всіх процесів. Поряд зі зміцненням безпеки, успішне впровадження DevSecOps залучає аналіз усіх можливих бізнес-ризиків. Очікується, що у глобальному масштабі ринок DevSecOps зросте на 33.7% у період до 2023 року.

Сервіси DevSecOps забезпечують не тільки безпечну, а й значно швидшу доставку застосунків.

Активне впровадження DevSecOps і переосмислення поточного підходу до управління та безпеки допомагає компаніям досягти небувалий рівень успіху.

Як впровадити DevSecOps?

DevSecOps зібрав все в єдиний оптимізований процес, включивши безпеку на рівні коду, забезпечивши тим самим захист застосунків на всіх рівнях процесу виробництва.

5 ознак успішної реалізації DevSecOps:

– обов'язкова безпека на всіх рівнях;
– ретельний аналіз та оцінка перед впровадженням елементів безпеки;
– зміни, пов'язані з безпекою, на рівні коду;
– автоматизація всіх можливих процесів;
– безперервний моніторинг за допомогою попереджень та інформаційних панелей.

Бізнес-переваги DevSecOps

Переходячи від DevOps до DevSecOps, організації удосконалюють процеси конвеєра розробки/пайплайну. Підвищення ефективності співробітництва команд розробки і безпеки забезпечує виявлення уразливостей та мінімізацію загроз на ранніх стадіях.

DevSecOps також надає низку інших переваг для компаній, наприклад: велика гнучкість і швидкість реагування у питаннях безпеки, вдосконалена комунікація між командами, а також швидке виявлення уразливостей коду. Це дозволяє організаціям швидко реагувати та адаптуватися до нових змін.

Деякі додаткові переваги також є результатом впровадження DevSecOps, наприклад:

– **Автоматичний захист коду** – DevSecOps знижує ризик виникнення проблем з безпекою через людський фактор за рахунок автоматизації тестів, високого охоплення, узгодженості та максимальної автоматизації процесів. Будь-які проблеми будуть виявлені та усунуті миттєво після їх виявлення;

– **Безперервне забезпечення безпеки** – завдяки засобам автоматизації, організації можуть максимально налагодити механізми тестування та подання звітності, тим самим гарантуючи негайне вирішення всіх проблем безпеки, що виникають;

– **Використання ресурсів безпеки** – DevOps автоматизує більшість стандартних процесів безпеки, таких як моніторинг подій, керування обліковими записами, безпека коду та оцінка уразливостей. Це дозволяє фахівцям з безпеки заощадити час і зосередити свою увагу на виявленні загроз та усуненні стратегічних ризиків.

Контрольні запитання

1. Що таке DevSecOps?
2. У чому потреба DevSecOps?
3. В чому різниця між DevOps та DevSecOps?
4. Назвати шість основних компонентів DevSecOps.
5. Навіщо переходити від DevOps до DevSecOps?
6. На які ключові кроки потрібно звернути увагу при впровадженні DevSecOps?
7. Назвати основні інструменти DevSecOps.
8. Яке призначення інструментів Kibana і Grafana?
9. За якими ознаками можна судити про успішність реалізації DevSecOps?
10. Які бізнес-переваги DevSecOps?

Завдання

Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними компонентами та інструментами DevSecOps.

Дати відповіді на контрольні запитання.

Лабораторна робота 4
Діаграми UML. Ознайомлення з інтерфейсом StarUML

Мета роботи: ознайомитися з інтерфейсом програми StarUML та проаналізувати інструменти програми.

Теоретичні відомості

1 Візуальне моделювання та UML

Нині **UML** (Unified Modeling Language – уніфікована мова моделювання) – одна з найпопулярніших технологій у галузі програмної інженерії. Саме UML дозволяє системним архітекторам подавати своє бачення системи у вигляді набору стандартних діаграм, які, до того ж, є гарним засобом комунікації в команді розробників і незамінним помічником у спілкуванні із замовником. І при цьому, UML є логічною і простою для вивчення нотацією, навичками використання якої, поза сумнівом, має оволодіти будь-який фахівець у галузі програмної інженерії. Знання UML потрібні розробникам, системним архітекторам, менеджерам, QA-фахівцям[9] та багатьом іншим.

Розглянемо детально складові терміну UML – уніфікована мова моделювання.

По-перше, UML – це мова. Мова як така – це знакова система для зберігання і передавання інформації. Розрізняють мови *формальні*, правила вживання яких строго і явно визначені, та *неформальні*, що використовуються здебільшого у розмові, ніж при написанні. Розрізняються також мови *природні*, що з'являються як би самі собою як результат неперсоніфікованих зусиль маси людей, і мови *штучні*, які є плодом видимих зусиль певних осіб.

UML можна охарактеризувати як формальну штучну мову, хоча і не в такому ступені, як багато поширених мов програмування. Ознакою штучності є наявність трьох загальновизнаних авторів UML – Граді Буча, Івара Якобсона та Джеймса Рамбо. Водночас у формування мови внесли вклад інші численні теоретики і розробники. Мовотворча практика стосовно UML триває, що дає підґрунтя вважати UML певною мірою природною мовою.

Опис UML здебільшого формальний, але містить і явно неформальні складові. Для опису формальних штучних мов (зокрема, для опису мов програмування) придумано і використовується безліч різних способів, проте на практиці склалася загальноприйнята структура таких описів. Вважається, що

---

[9] QA-фахівець (Quality Assurance engineer) – це фахівець із забезпечення якості, діяльність якого спрямована на поліпшення процесу розробки ПЗ, запобігання дефектам і виявлення помилок в роботі продукту. Це людина з певним набором технічних знань, яка власноруч тестує нові версії програм, щоби випустити якісний продукт.

формальна штучна мова описана належним чином, якщо цей опис має щонайменше такі складові:

– синтаксис (syntax) – визначення правил складання конструкцій мови;

– семантика (semantics) – правила тлумачення мовних конструкцій (операторів), які пояснюють смислове значення і характер дій, виконуваних комп'ютером під час їх виконання;

– прагматика (pragmatics) – визначення правил використання конструкцій мови для досягнення певної мети.

Як формальна штучна мова UML має синтаксис, семантику і прагматику, хоча ці частини можуть називати інакше, ніж це прийнято в текстових мовах програмування, позаяк, по-перше, UML – мова графічна, а не текстова, а по-друге, UML – мова моделювання, а не програмування.

По-друге, UML – це мова моделювання. Слово "моделювання", що входить у назву UML, має безліч смислових відтінків і способів вживання. Зокрема, англійські слова modeling і simulation обидва перекладають як "моделювання", хоча означають вони різні речі. Перше з них позначає складання моделі, яка використовується тільки для опису модельованого об'єкта або явища, друге – складання моделі, що може бути використана для здобуття суттєвої інформації про модельований об'єкт або явище. При цьому для першого зазвичай додають уточнювальний прикметник – чисельне моделювання, математичне моделювання та ін. UML є мовою моделювання саме в першому з розглянутих контекстів, хоча відомі деякі успішні спроби використання UML і в другому сенсі.

Щодо розроблення програмного забезпечення так склалося, що результати фаз аналізу і проєктування, оформлені засобами певної мови, прийнято називати моделлю. Діяльність зі складання моделей природно назвати моделюванням. Саме у цьому сенсі UML є мовою моделювання.

Отже, **модель UML** – це сукупність конструкцій мови, які описують певний об'єкт або явище засобами сутностей та стосунків поміж ними.

По-третє, UML – це уніфікована мова моделювання. Поштовхом до створення UML стало, насамперед, масове поширення об'єктно-орієнтованого підходу до розроблення програмних систем, унаслідок чого виникла потреба у відповідних засобах. Іншими словами, появи чогось подібного до UML з нетерпінням чекали практики. Саме тому три видатні фахівці в цій галузі Граді Буч, Івар Якобсон та Джеймс Рамбо наважилися об'єднати зусилля саме з метою уніфікації своїх (і не лише своїх) розробок відповідно до соціального замовлення. В результаті плідної співпраці, за підтримки і сприяння усієї міжнародної громадськості, в 1996 році автори змогли уніфікувати значну частину того, що було відомо і до них, а світ отримав теоретично витончений і практично корисний засіб – UML. Надалі з'ясувалося, що більшість великих компаній (DEC, HP, IBM, Microsoft, Oracle, Rational Software та ін.) готові розглянути UML як основну стратегію свого бізнесу, і було створено некомерційний консорціум OMG (Object Modeling Group), який об'єднав провідних виробників ПЗ, і вже в січні 1997 був виданий UML 1.0. Згодом 2003 року була прийнята версія 1.5, а 2004 – версія 2.0.

Як мова графічного візуального моделювання UML має свою нотацію – прийняті позначення. **Нотація** забезпечує семантику мови, є засобом уніфікації позначень візуального моделювання, що забезпечує графічну інтерпретацію системи, яка порівняно легко і вільно сприймається людиною. Нотація UML, в першу чергу, необхідна аналітикам, вона дозволяє наочно відобразити вимоги, які вдалося зібрати у замовника.

Нині діюча остання версія нотації UML 2.5 була опублікована в червні 2015 року.

Моделювання засобами UML здійснюється поетапною побудовою набору діаграм, кожна з яких відображає якусь частину системи або її задуму.

Діаграма – це графічне подання набору елементів. Зазвичай діаграма зображується у вигляді графа з вершинами (сутностями) і ребрами (стосунками). Діаграми підпорядковуються нотації і зображуються у відповідності з нею. Наприклад, на рис. 4.1[10] наведено структуру діаграм нотації UML 2.3, які можна подати як діаграму класів UML:

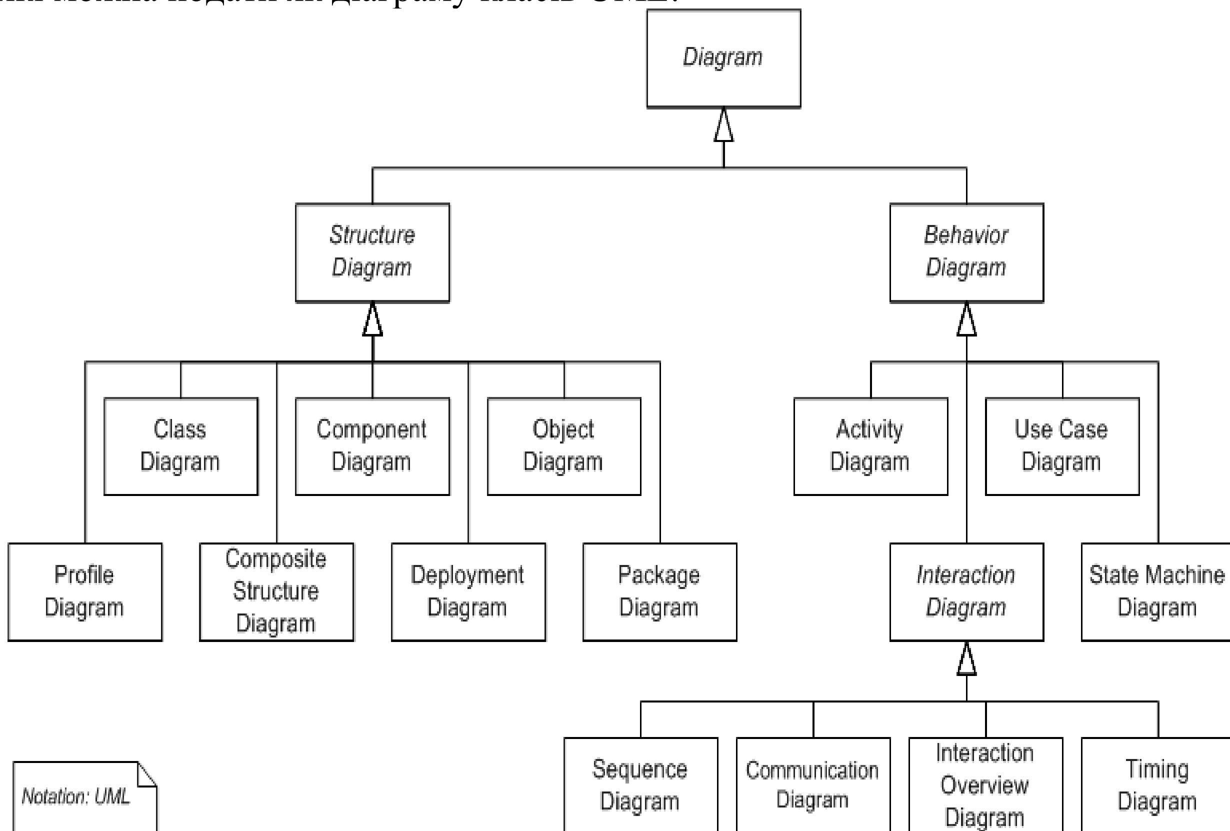


Рисунок 4.1 – Структура діаграм UML 2.3

UML визначає три категорії діаграм, розділені на типи[11]:

Структурні UML-діаграми:

– діаграма класів (Class Diagram);
– діаграма об’єктів (Object Diagram);
– діаграма компонентів (Component Diagram);

[10] Unified Modeling Language. URL: http://wikipedia.ua.azmaster.biz/wiki/UML.
[11] What is UML. URL: https://www.uml.org/what-is-uml.htm.

- діаграма складеної структури (композитів) (Composite Structural Diagram);
- діаграма кооперацій (Collaboration);
- діаграма розгортання (Deployment Diagram);
- діаграма пакетів (Package Diagram).

Поведінкові UML-діаграми:

- діаграма діяльності (Activity Diagram);
- діаграма варіантів використання (прецедентів) (Use case diagram);
- діаграма станів (кінцевого автомата) (State Machine diagram).

Діаграми взаємодії (Interaction Diagram):

- діаграма послідовності (Sequence Diagram);
- діаграма комунікації (зв’язків) (Communication Diagram);
- діаграма огляду взаємодії (Interaction Overview Diagram);
- діаграма синхронізації (часу) (Timing Diagram).

UML забезпечує підтримку всіх *етапів життєвого циклу* ІС та використовує для цих цілей діаграми.

На етапі створення *концептуальної моделі* для опису бізнес-діяльності використовуються моделі бізнес-прецедентів і діаграми діяльності, для опису *бізнес-об'єктів* – моделі бізнес-об'єктів і діаграми послідовностей.

На етапі створення *логічної моделі ІС* опис вимог до системи задається у вигляді моделі та опису системних прецедентів, а попереднє проєктування здійснюється з використанням діаграм класів, діаграм послідовностей і діаграм станів.

На етапі створення *фізичної моделі* детальне проєктування виконується з використанням діаграм класів, діаграм компонентів, діаграм розгортання.

Далі наводяться визначення і описується призначення перерахованих діаграм і моделей стосовно завдань *проєктування ІС* (у дужках наведено *альтернативні* назви діаграм, що використовуються в сучасній літературі).

Діаграми прецедентів (діаграми варіантів використання, use case diagrams) – це узагальнена модель функціонування системи в навколишньому середовищі.

Діаграми видів діяльності (діаграми діяльностей, activity diagrams) – модель бізнес-процесу або поведінки системи в рамках прецеденту.

Діаграми взаємодії (interaction diagrams) – модель процесу обміну повідомленнями між об'єктами, подається у вигляді **діаграм послідовностей** (sequence diagrams) або **кооперативних діаграм** (collaboration diagrams).

Діаграми станів (statechart diagrams) **–** модель динамічної поведінки системи та її компонентів при переході з одного стану в інший.

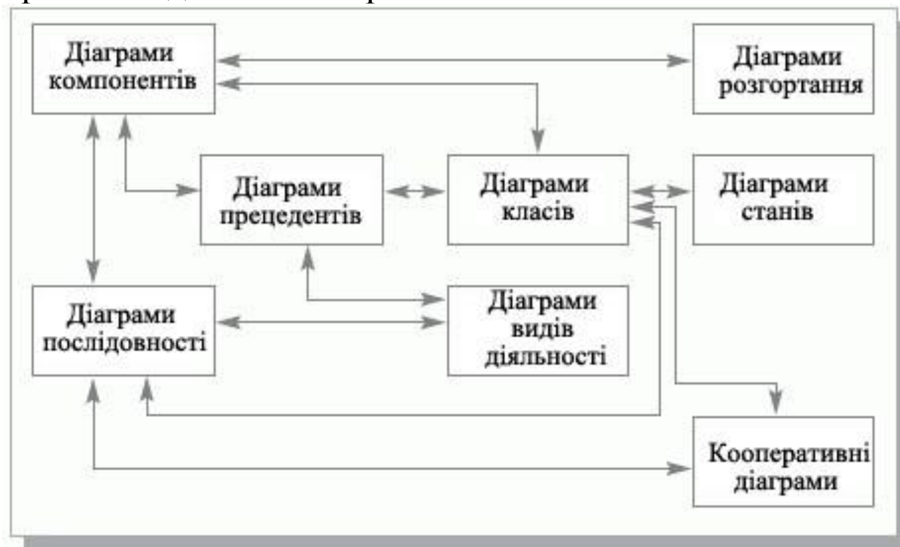
Діаграма класів (class diagrams) – логічна модель базової структури системи, відображає статичну структуру системи і зв'язків між її елементами.

Діаграми бази даних (database diagrams) – модель структури бази даних, відображає таблиці, стовпці, обмеження тощо.

Діаграми компонентів (component diagrams) **–** модель ієрархії підсистем, відображає фізичне розміщення баз даних, застосунків та інтерфейсів ІС.

Діаграми розгортання (діаграми розміщення, deployment diagrams) – модель фізичної архітектури системи, відображає апаратну конфігурацію ІС.

Далі на рисунку показані стосунки між різними видами діаграм UML. Стрілки можна інтерпретувати як стосунок "є джерелом вхідних даних для..." (наприклад, діаграма прецедентів є джерелом даних для діаграм видів діяльності і послідовності). Наведена схема є наочною ілюстрацією ітеративного характеру розробки моделей з використанням UML.



Далі розглянемо основні елементи нотації діаграм та принципи їхньої побудови. Архітектурний базис UML визначає базові поняття, якими оперує мова: сутності, стосунки та діаграми.

Сутності – це певні абстракції, які є базовими елементами моделей. В UML є чотири типи сутностей: структурні (актори, класи, інтерфейси, компоненти, вузли), поведінки (преценденти, діяльності, стани і повідомлення), групування та анотаційні.

Структурні сутності – це статичні поняття, які відповідають концептуальним, логічним чи фізичним елементам системи. Структурні сутності, зазвичай, позначають іменниками. Розрізняють п’ять головних структурних сутностей: актори, класи, інтерфейси, компоненти, вузли. Кожна з сутностей може мати свої підвиди[12].

Актор (Actor) – це суб’єкт, який перебуває поза системою, що моделюється, і безпосередньо з нею взаємодіє. Графічно на діаграмах акторів зображають стилізованими людськими фігурками, під якими записуються їхні імена.

Клас (Class) – це сукупність однотипних сутностей предметної області (об’єктів) зі спільними атрибутами, операціями, стосунками та семантикою. В UML класи зображають прямокутником, розділеним на три секції, в яких записують назву класу, атрибути та операції, відповідно (детальніше діаграми класів будуть розглядатися та створюватись у відповідній лабораторній роботі). Назву абстрактного класу позначають курсивом. Атрибути та операції мають чітко визначені формати запису, які відображають їхні найважли-

[12] Поділ на головні сутності та їхні підвиди строго не фіксується, отож у літературі трапляється й дещо інша класифікація сутностей.

віші характеристики (назви, типи тощо).

Об'єкти (Objects) – це екземпляри класів з конкретними значеннями атрибутів. Об'єкт має зображення, подібне до зображення класу, проте назву об'єкта підкреслюють і записують у вигляді: <назва об'єкта>:<назва класу>. Якщо ідентифікація об'єкта неважлива, то вказують лише назву класу, до якого належить об'єкт::<назва класу>. При зображенні об'єктів секції атрибутів та операцій, здебільшого, опускають.

Анотаційна сутність – це коментар для пояснення чи зауваження до будь-якого елемента моделі. Є тільки один тип анотаційної сутності – **примітка** (Note). Графічно примітку зображають прямокутником із загнутим правим верхнім кутом.

2 Вибір CASE-засобів проєктування інформаційних систем

Для супроводу процесу побудови, аналізу та документування моделі, а також перевірки моделі і генерації програмних кодів розробники використовують спеціально для цих цілей створені CASE-інструменти проєктування систем.

У загальному сенсі CASE (Computer-Aided Software Engineering) – це набір інструментів та методів програмної інженерії для проєктування програмного забезпечення, який допомагає забезпечити високу якість програм, відсутність помилок і простоту в обслуговуванні програмних продуктів.

Далі як приклад моделювання системи розглянемо програмний інструмент моделювання StarUML. Дана програмна платформа має ліцензію GNU GPL[13] і доступна для установки з офіційного сайту StarUML – http://staruml.io/.

StarUML підтримує одинадцять різних типів діаграм (Class, Object, Use Case, Component, Deployment, Composite Structure, Sequence, Communication, Statechart, Activity and Profile Diagram), прийнятих у нотації UML 2.0, а також підхід MDA[14], пропонує налаштування параметрів користувача для адаптації середовища розробки, підтримує розширення, надає різного роду модулі, StarUML виконує кодогенерацію мовами C++, C#, Java.

3 Створення нового проєкту в StarUML

Основна структурна одиниця в StarUML – це проєкт. проєкт зберігається в одному файлі у форматі XML з розширенням ".uml". проєкт може містити одну або декілька моделей і різні подання цих моделей (View) – візуальні подання інформації, що міститься в моделях. Кожне подання моделі містить діаграми – візуальні образи, які відображають певні аспекти моделі. Новий проєкт буде автоматично створений при запуску програми StarUML. При

[13] Ліцензія на вільне програмне забезпечення , створена в рамках проєкту GNU 1988 року, за якою автор передає програмне забезпечення в суспільну власність.

[14] MDA (Model Driven Architecture) – модельно-налаштовувана архітектура, створена консорціумом OMG як різновид модельно-орієнтованого підходу до розробки програмного забезпечення. Її суть полягає в побудові абстрактної метамоделі управління та обміну метаданими (моделями) і задаванні способів її трансформації в підтримувані технології програмування (Java, CORBA, XML та ін.).

цьому буде запропоновано в діалоговому вікні вибрати один з підходів (Approaches), підтримуваних StarUML (рис. 4.2).

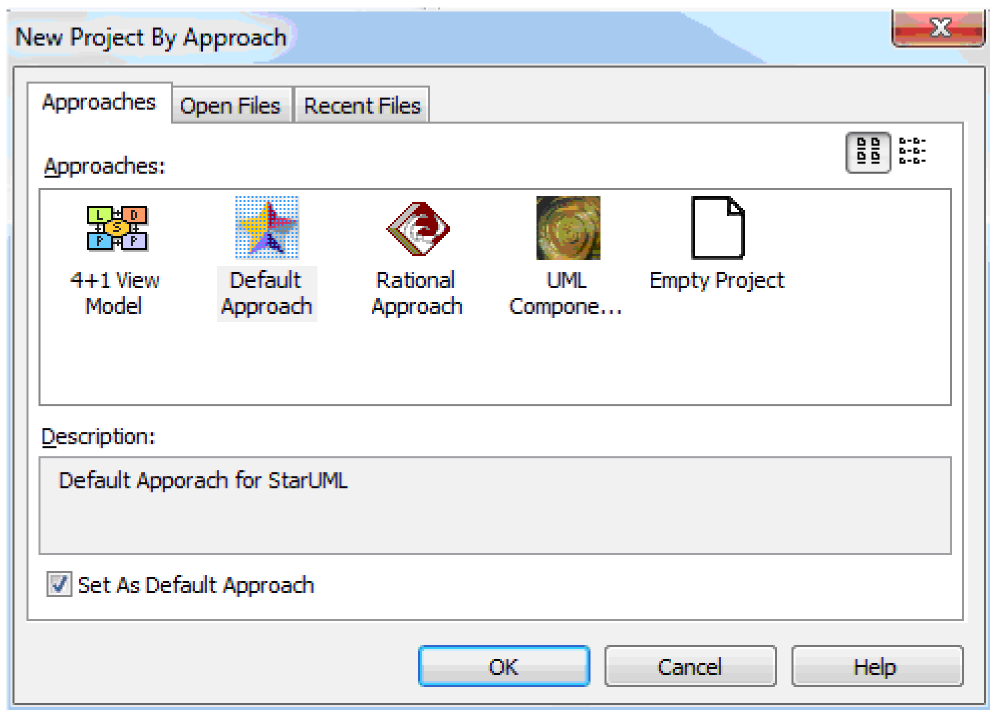


Рисунок 4.2 – Вибір підходу

Існують різні методології моделювання інформаційних систем, компанії-розробники систем також можуть розробляти свої методології. На початковій стадії проєктування варто визначити основні положення методології або вибрати одну з уже наявних. Для того щоб узгодити різні елементи та етапи моделювання, StarUML пропонує концепцію підходів.

Після того як вибрано один із запропонованих підходів, в StarUML з'явиться головне вікно програми (рис. 4.3).

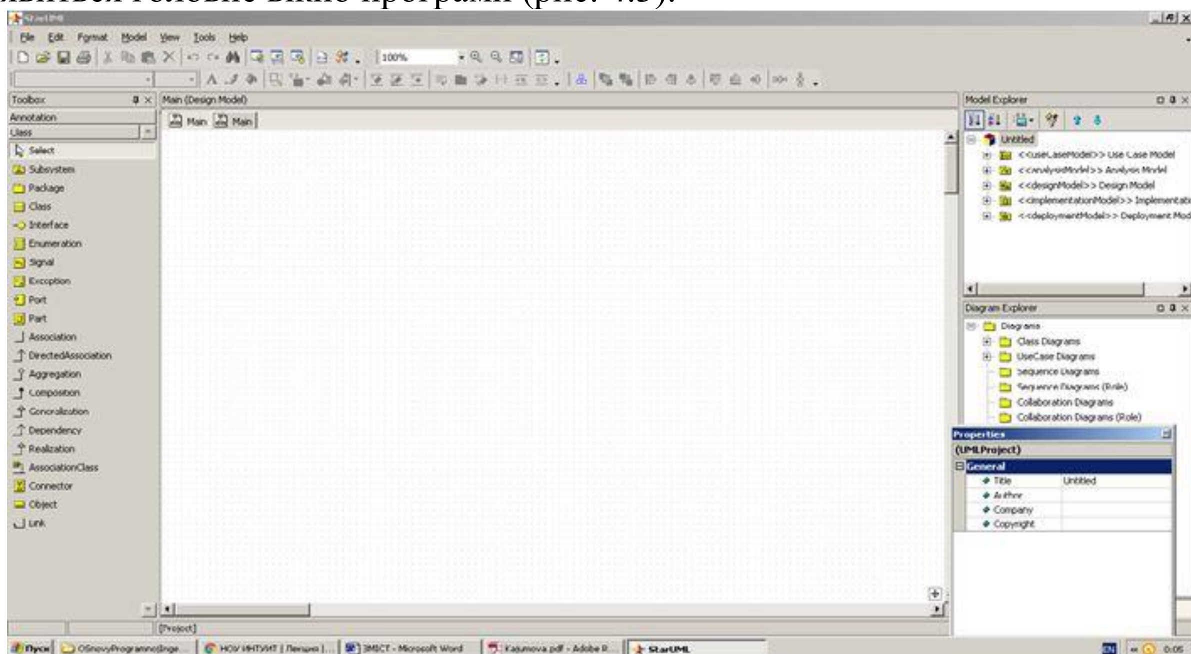
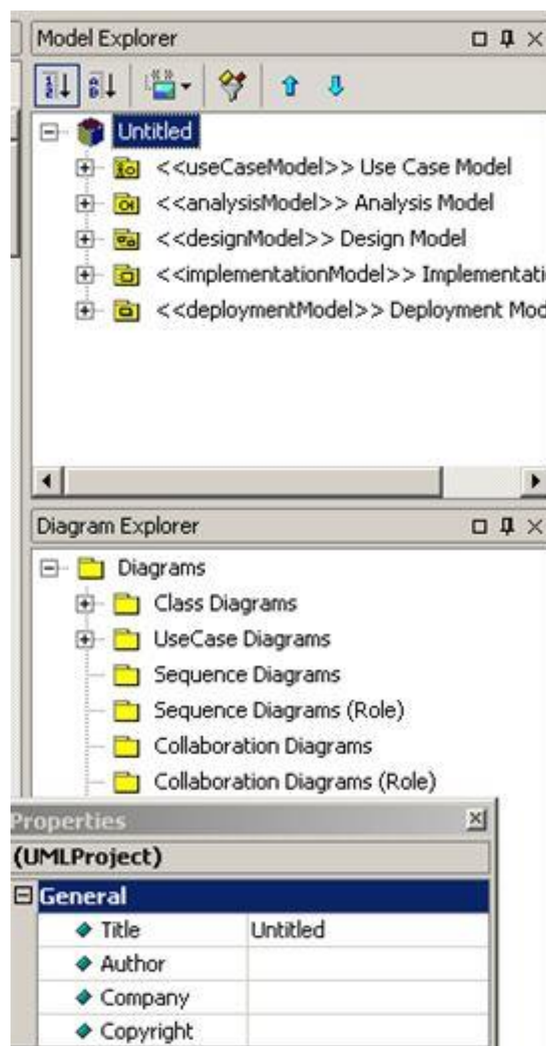


Рисунок 4.3 – Головне вікно програми

У верхній частині вікна розташовано головне меню, кнопки швидкого доступу. Ліворуч розташована панель елементів (Toolbox) із зображеннями елементів діаграми, відповідно до типу вибраної діаграми. В центрі розташовано робоче поле діаграми, на якому вона може бути побудована з використанням відповідних елементів на панелі інструментів.



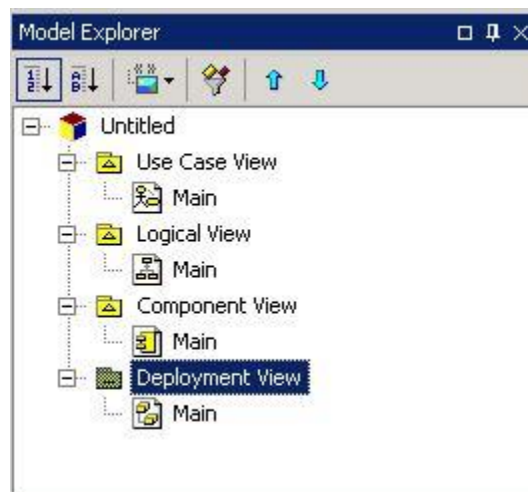
Праворуч розташовано панель інспектора моделі, на якому розміщені вкладки навігатора моделі Model Explorer, навігатора діаграм Diagram Explorer, вікно редактора властивостей Properties, вікно документування елементів моделі Documentation і редактор вкладень Attachments.

Керувати виглядом інспектора моделі, панелі елементів, закривати і відкривати редактори інспектора можна за допомогою пункту меню View. Якщо поруч із пунктом меню стоїть «галочка», цей елемент активний і його можна бачити у вікні програми або відкрити на доступних вкладках інспектора моделі.

Ієрархічна структура проєкту відображається праворуч на навігаторі моделі (Model Explorer). Залежно від вибраного підходу на навігаторі моделі будуть відображені різні пакети подання моделі. Кожен пакет подання буде містити елементи створюваних моделей і діаграм.

Якщо при створенні нового проєкту моделювання вибрано підхід Rational Approach, то при такому підході в навігаторі будуть присутні чотири пакети подання моделі системи:

– Use Case View – подання вимог до системи з описом того, що система повинна робити;



– Logical View – логічне подання системи з описом того, як система повинна бути побудована;

– Component View – подання реалізації з описом залежності між програмними компонентами;

– Deployment View – подання розгортання з описом апаратних елементів, пристроїв та програмних компонентів.

На рисунку праворуч кожне подання містить одну діаграму з ім'ям Main. Якщо

клацнути по ній двічі, відкриється робоче поле цієї діаграми і відповідна панель інструментів. Так, якщо клацнути двічі по діаграмі Main подання Use Case View, то відкриється робоче цієї діаграми та її панель елементів (рис. 4.4).

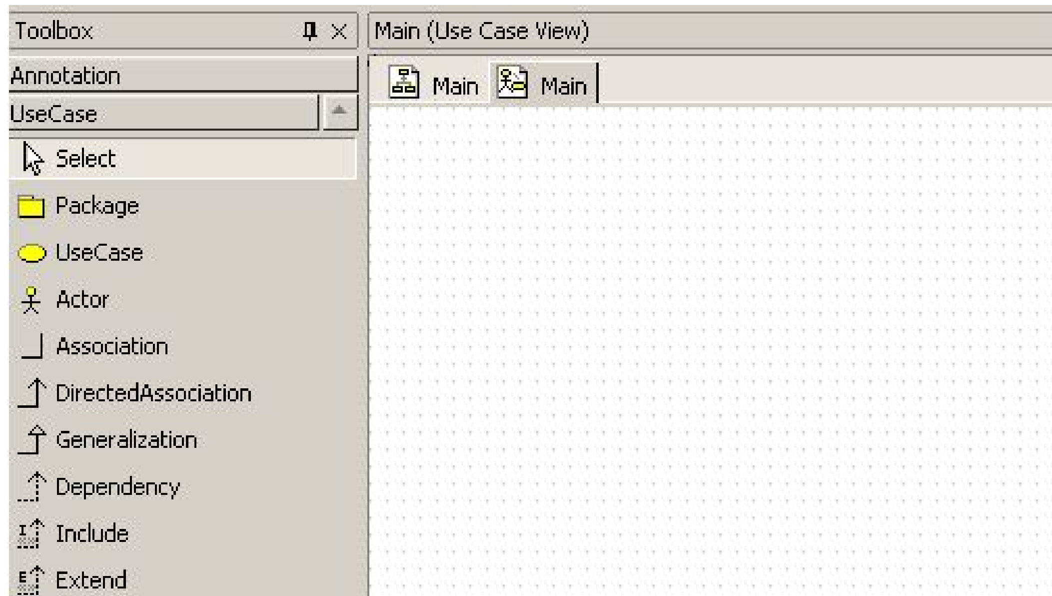


Рисунок 4.4 – Робоче поле діаграми прецедентів Main та її панель елементів

Властивості виділеного елемента моделі або діаграми відображаються праворуч внизу під навігатором моделі. На рис. 4.5 виводяться властивості діаграми Main, позаяк вона виділена в навігаторі моделі.

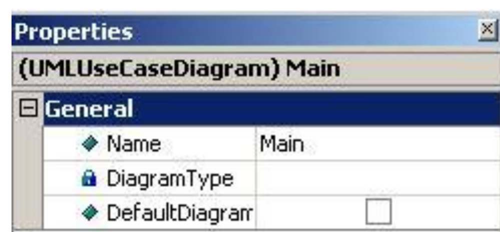


Рисунок 4.5 – Редактор властивостей

Контрольні запитання для самоконтролю

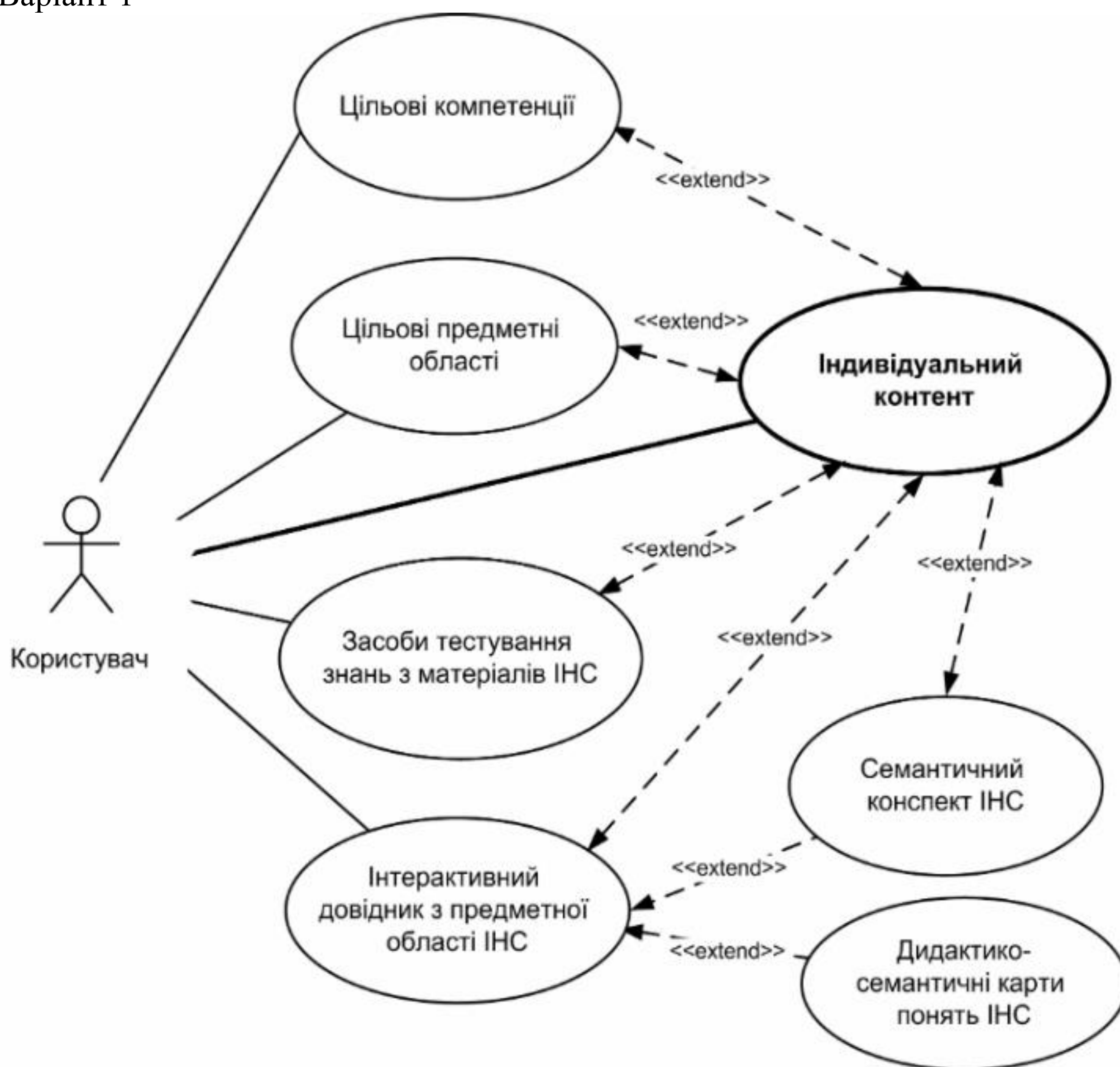
1. Що означає назва UML?
2. Що означає назва MDA?
3. Назвати різні категорії та типи діаграм UML.
4. Що таке модель у програмі StarUML?
5. Що таке подання?
6. Що таке діаграма?
7. Що таке проєкт?
8. Які елементи входять в структуру проєкту?
9. Яке розширення мають файли проєкту в StarUML?

Лабораторне завдання

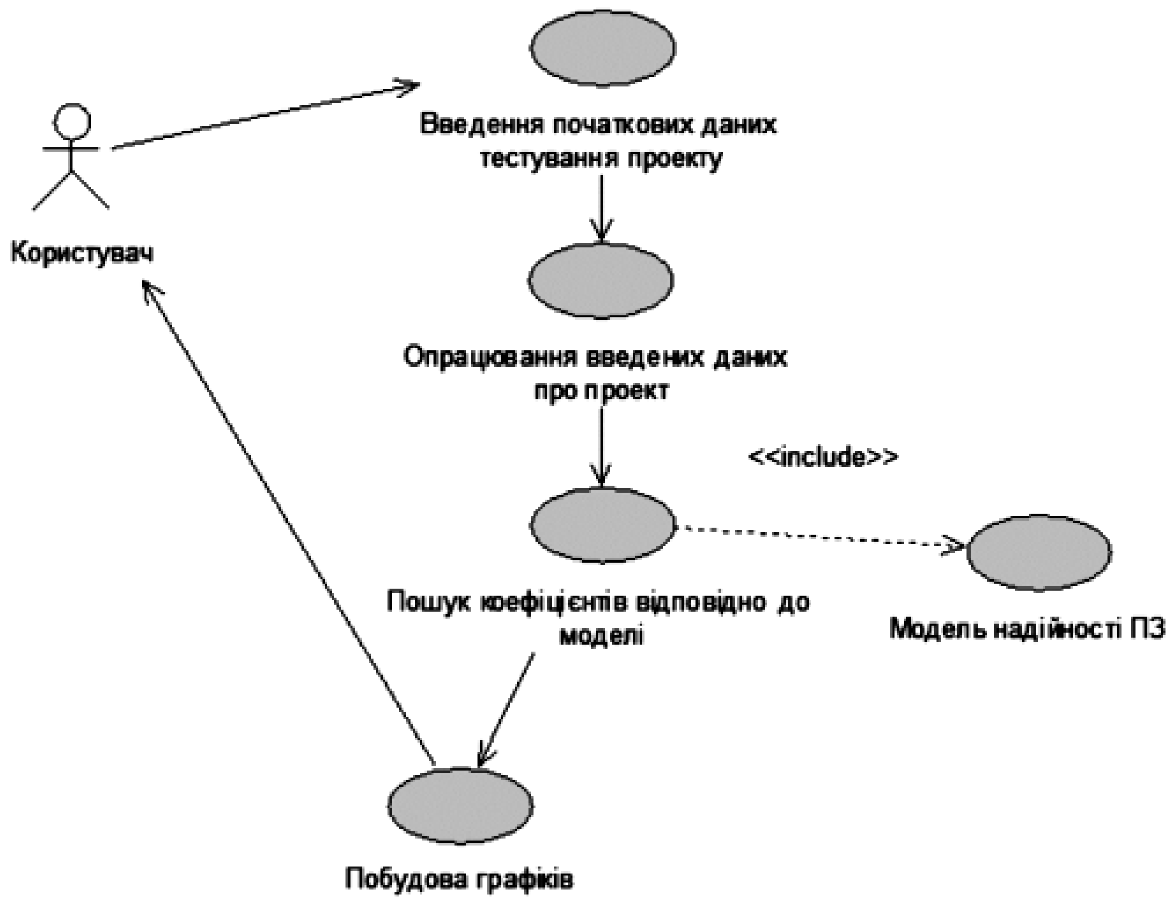
1. Дати відповіді на контрольні питання.
2. Створити діаграму UML за зразком відповідно до варіанта.
3. Оформити протокол лабораторної роботи.

Індивідуальні варіанти

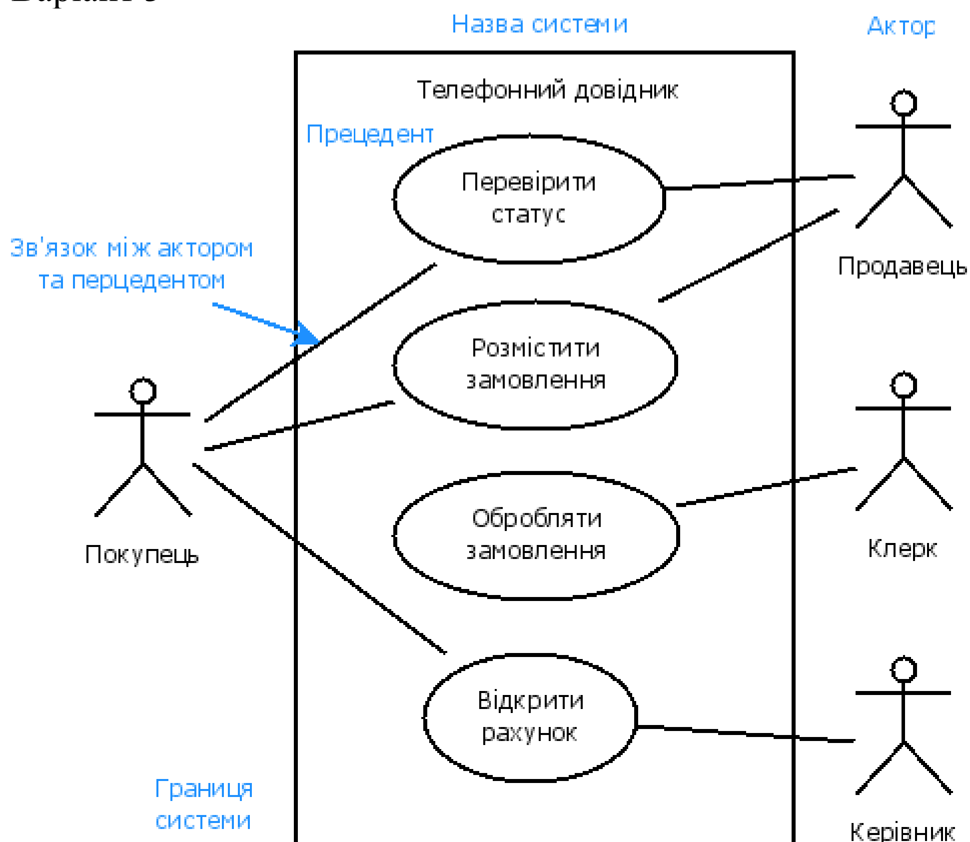
Варіант 1



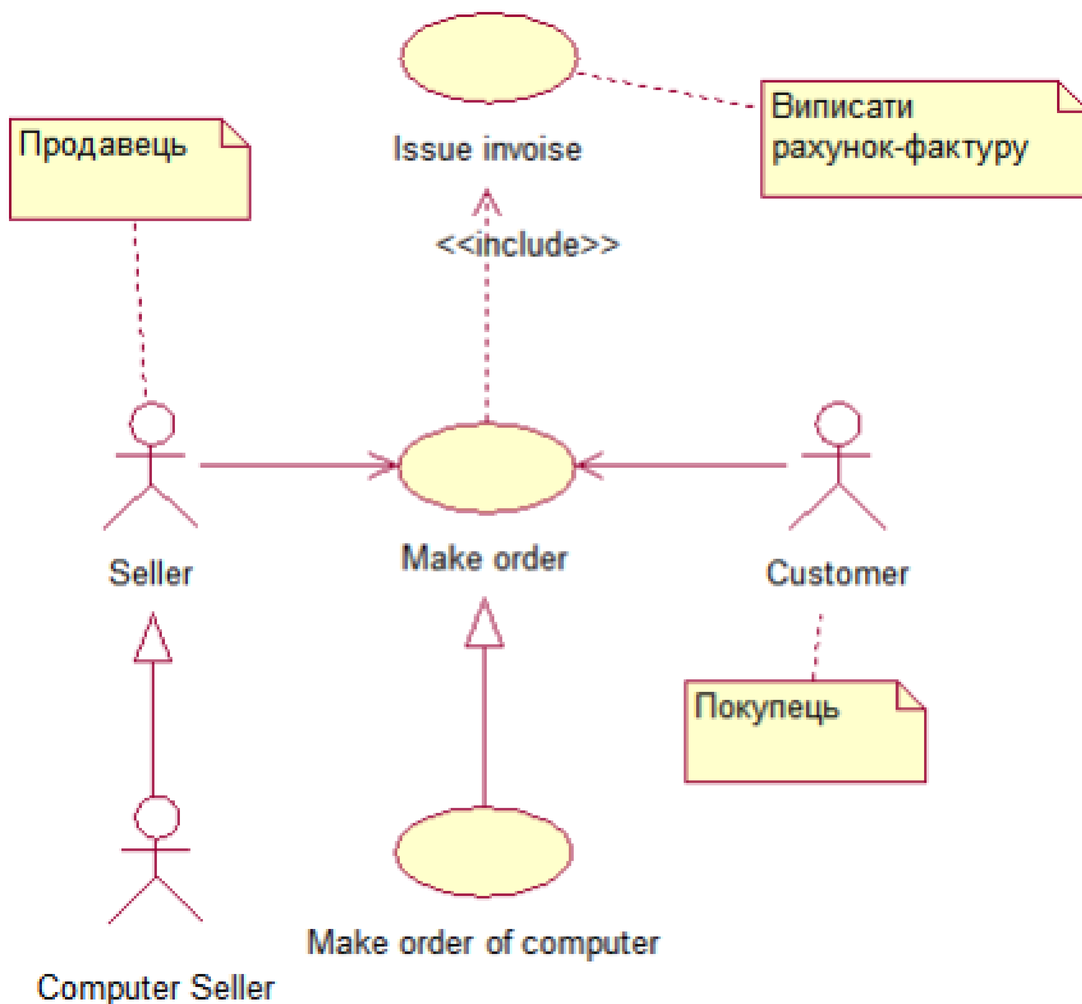
Варіант 2



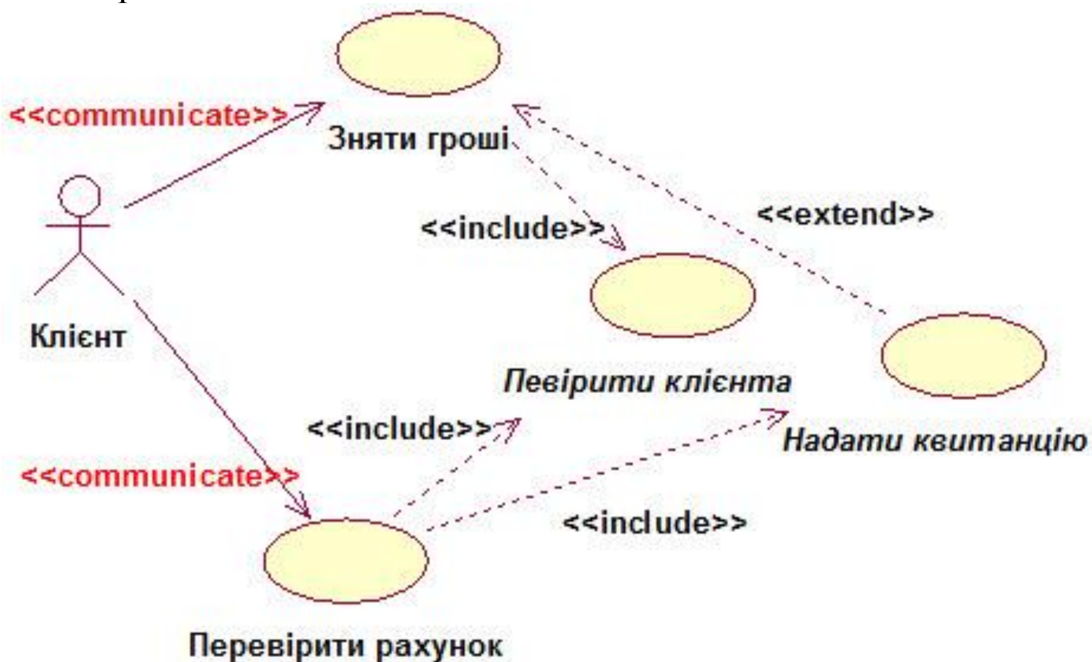
Варіант 3



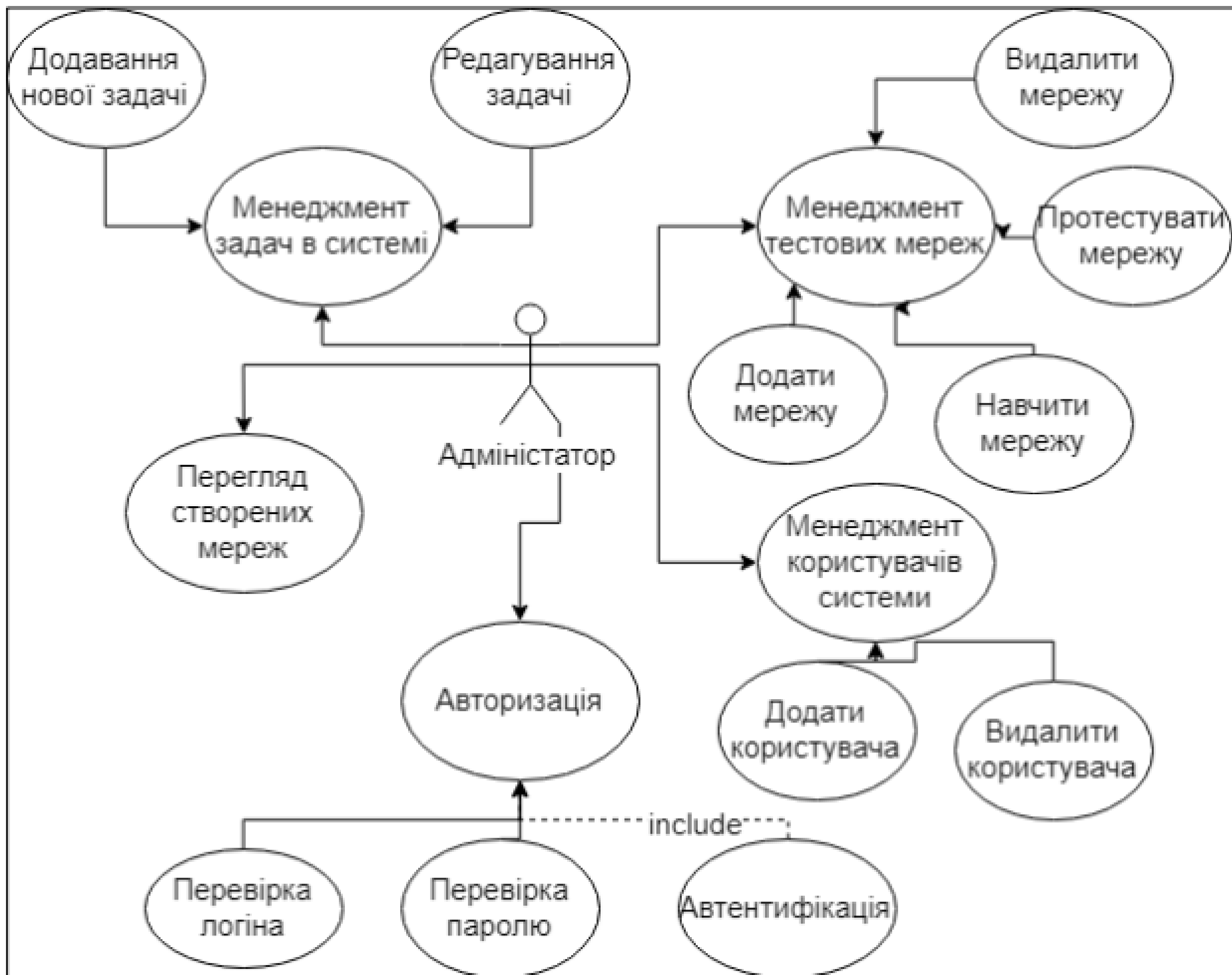
Варіант 4



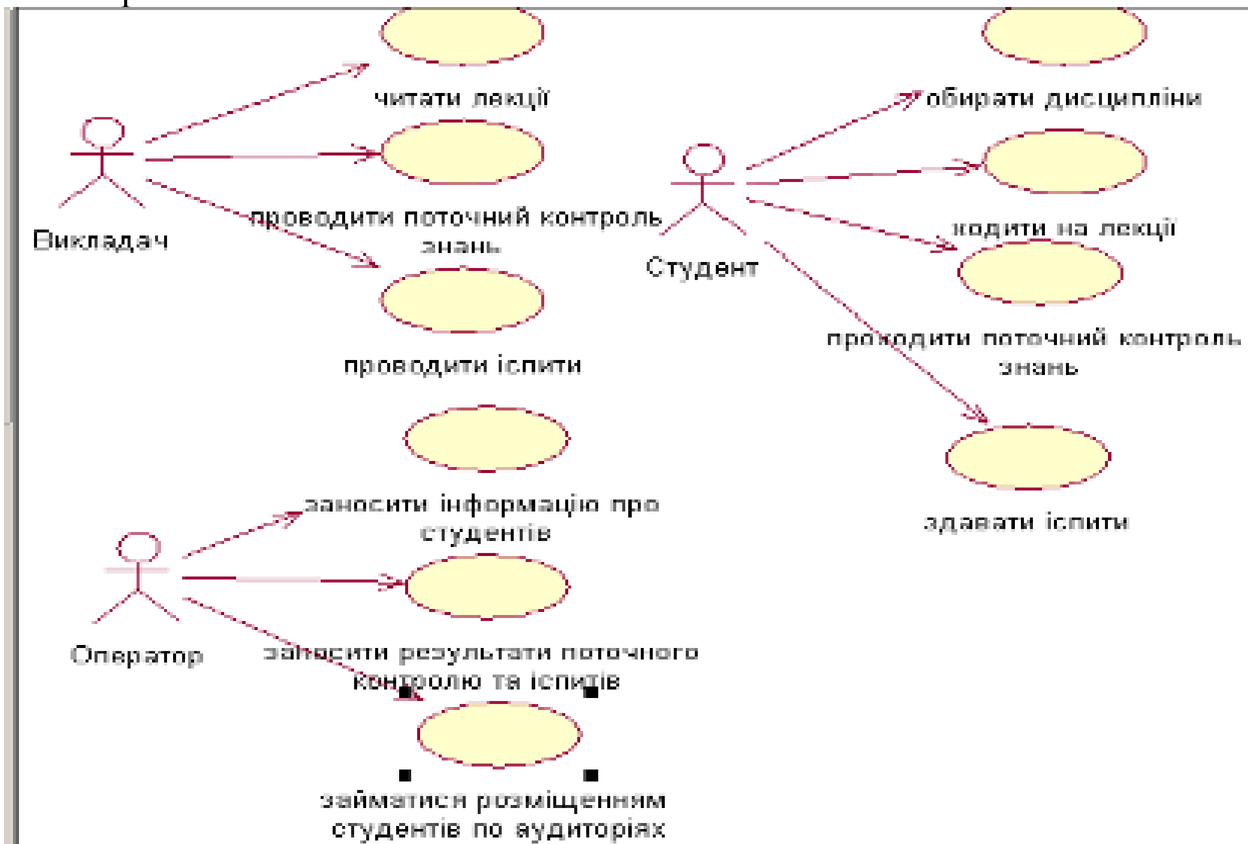
Варіант 5



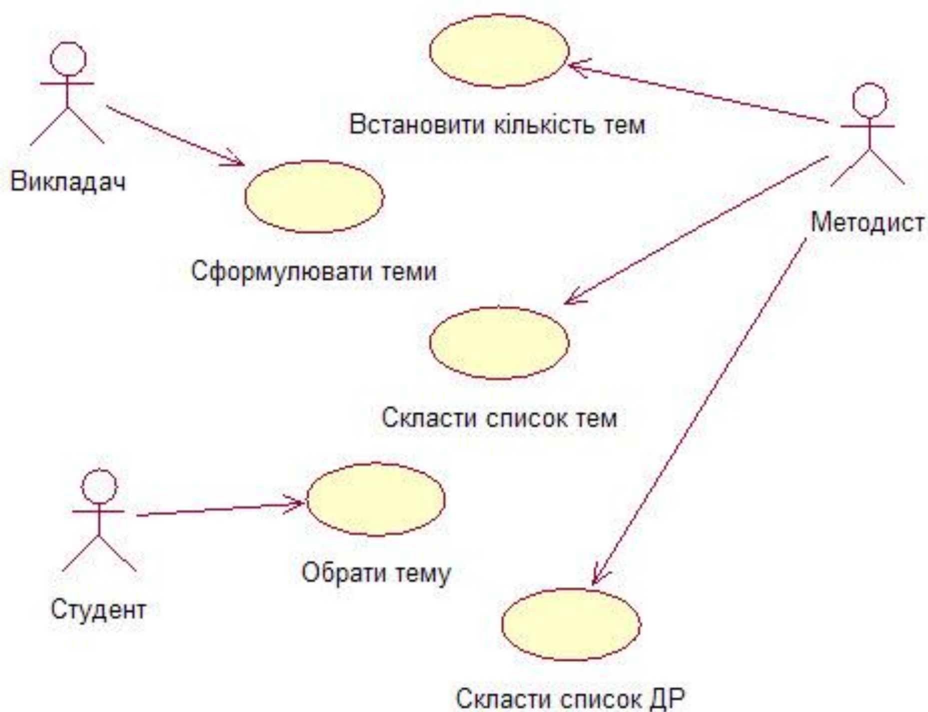
Варіант 6



Варіант 7



Варіант 8



Варіант 9

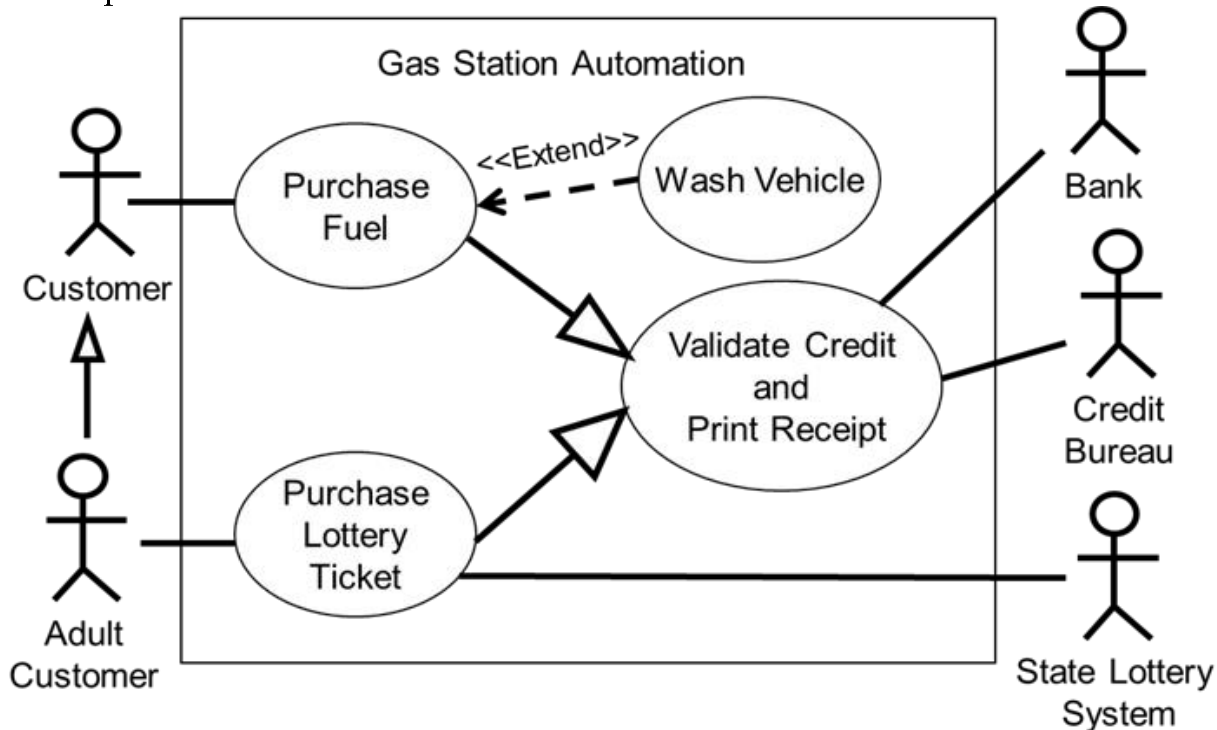
User Sign-In
Customer Authentication
Remember Me
«Service» Authentication
Single Sign-On
Identity Provider
«include»
Checkout
«include»
View / Update Shopping Cart
«include»
Calculate Total
«include»
Calculate Taxes and Shipping
«include»
Payment By Credit
Credit Payment Service
Payment
Payment By PayPal
PayPal

Варіант 10

Getting Payment
<<extends>>
Check Payment
<<extends>>
Bank Payment
Daily sales per product
Old Customer
Manager
<<includes>>
Daily sales per customer
<<includes>>
Making the Reports
Ordering Products
<<includes>>
Monthly sales per product
<<includes>>
New Customer
Credit & collector
Monthly sales per customer
Delivering the orders
Give order slips
<<includes>>
Give receipts
Dispatcher

Варіант 11

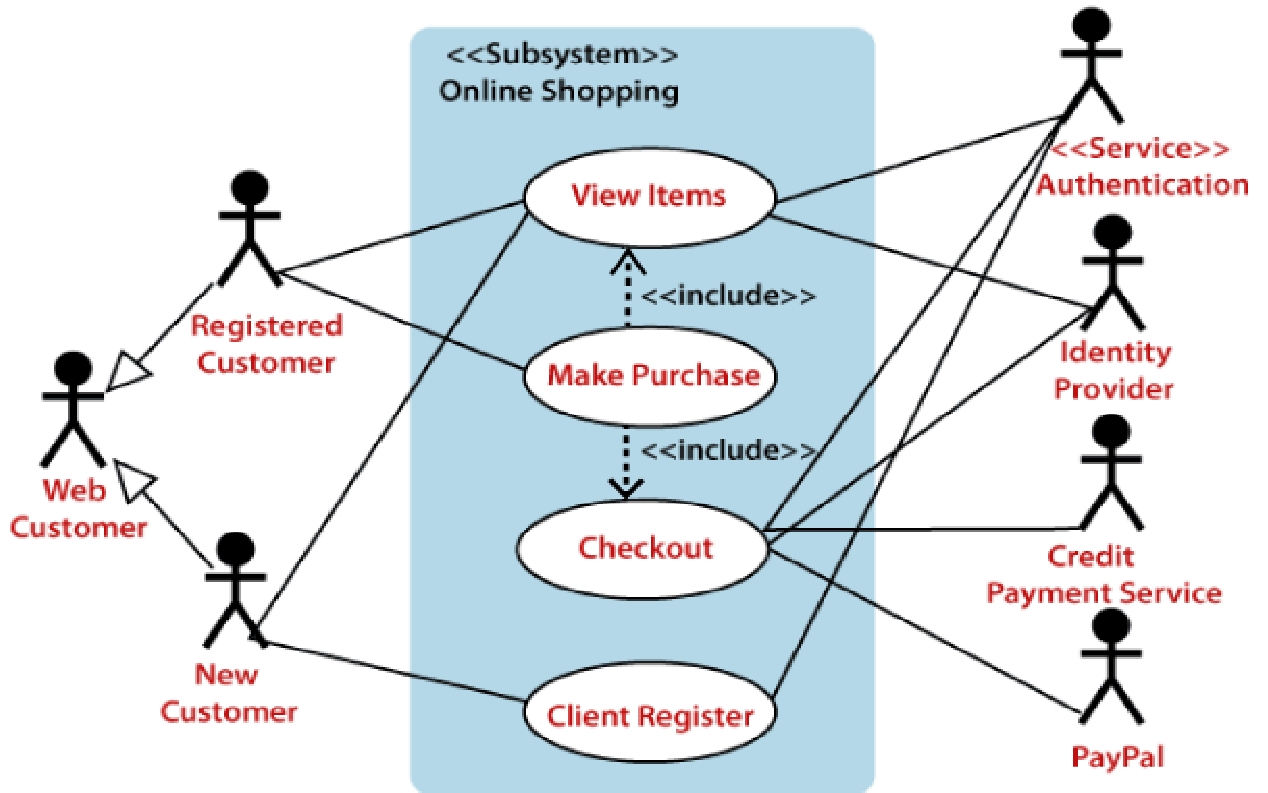
Варіант 12



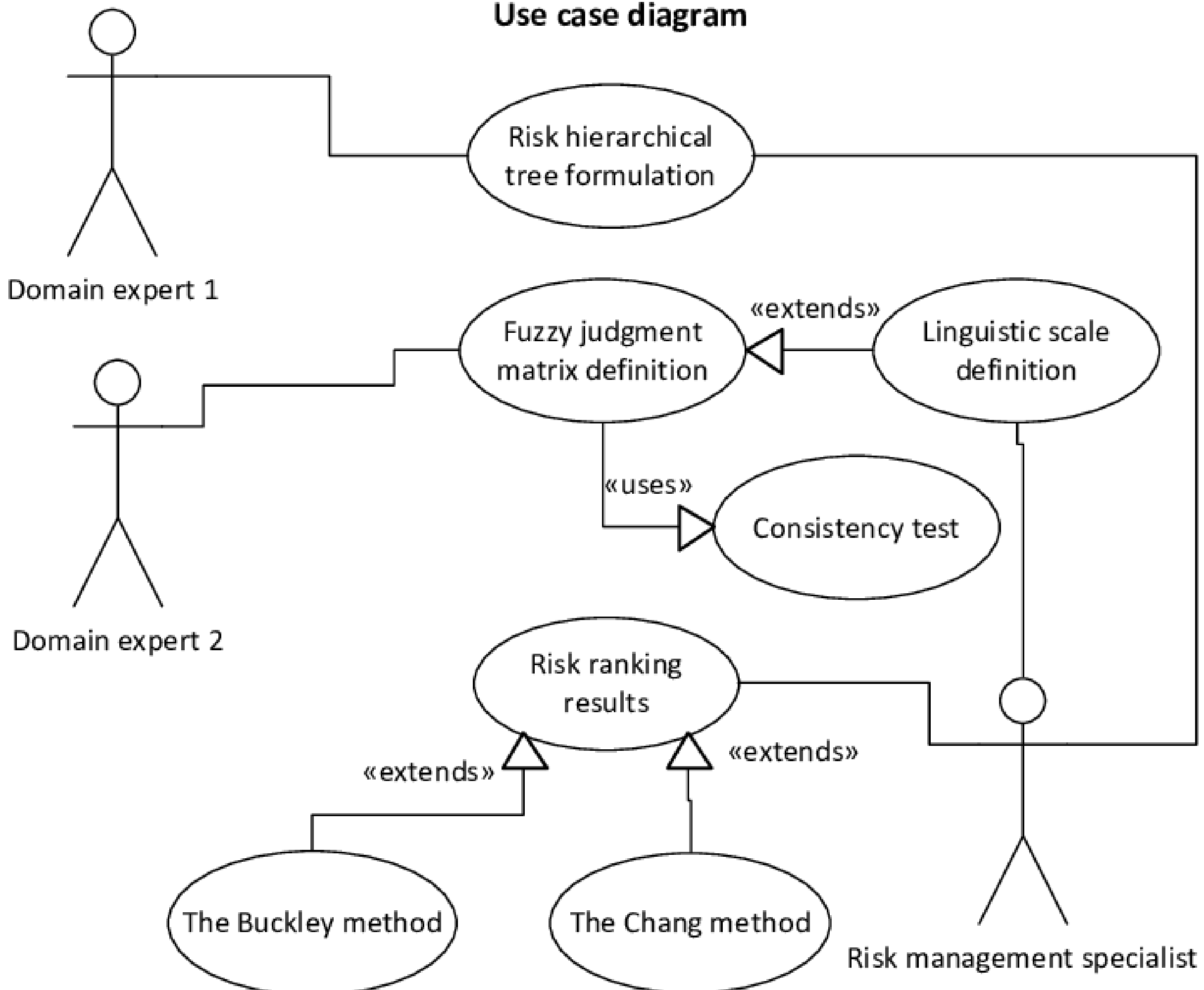
Варіант 13

User Sign-In
Customer Authentication
<<Service>> Authentication
Remember Me
<<include>>
Single Sign-On
Identity Provider
<<include>>
<<include>>
Checkout
View/Update Shopping Cart
Calculate Total
<<include>>
Calculate Taxes and Shipping
<<include>>
Payment By Credit
Credit Payment Service
Payment
Payment By PayPal
PayPal

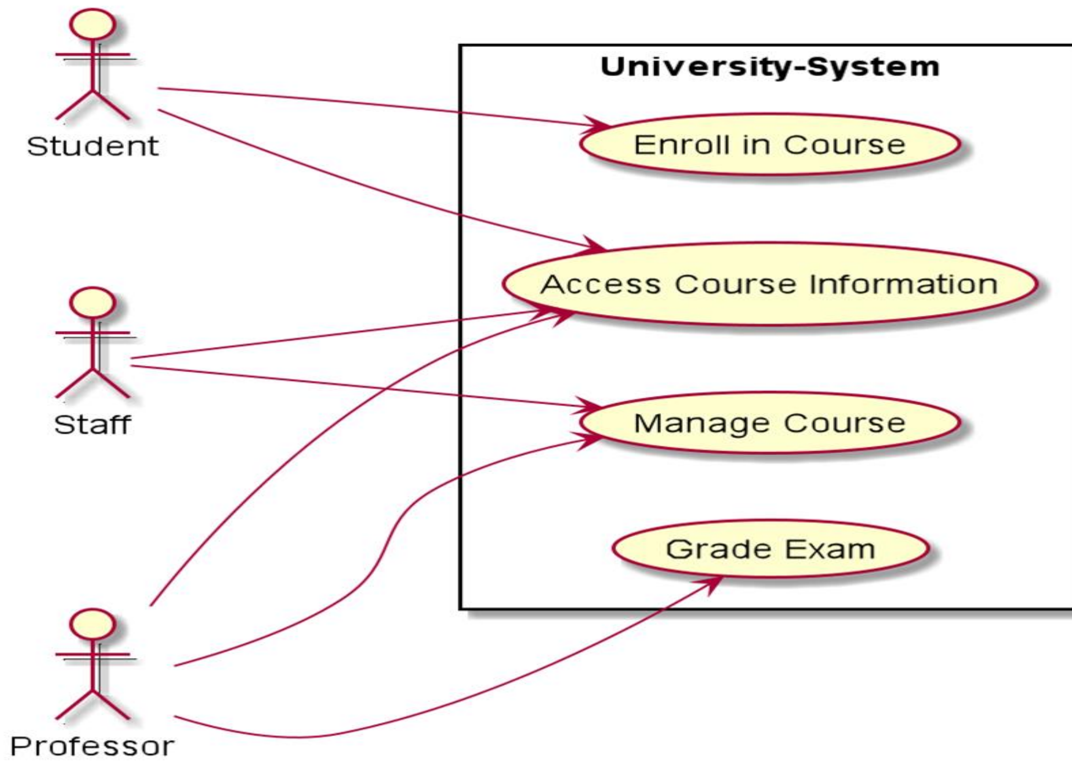
Варіант 14



Варіант 15



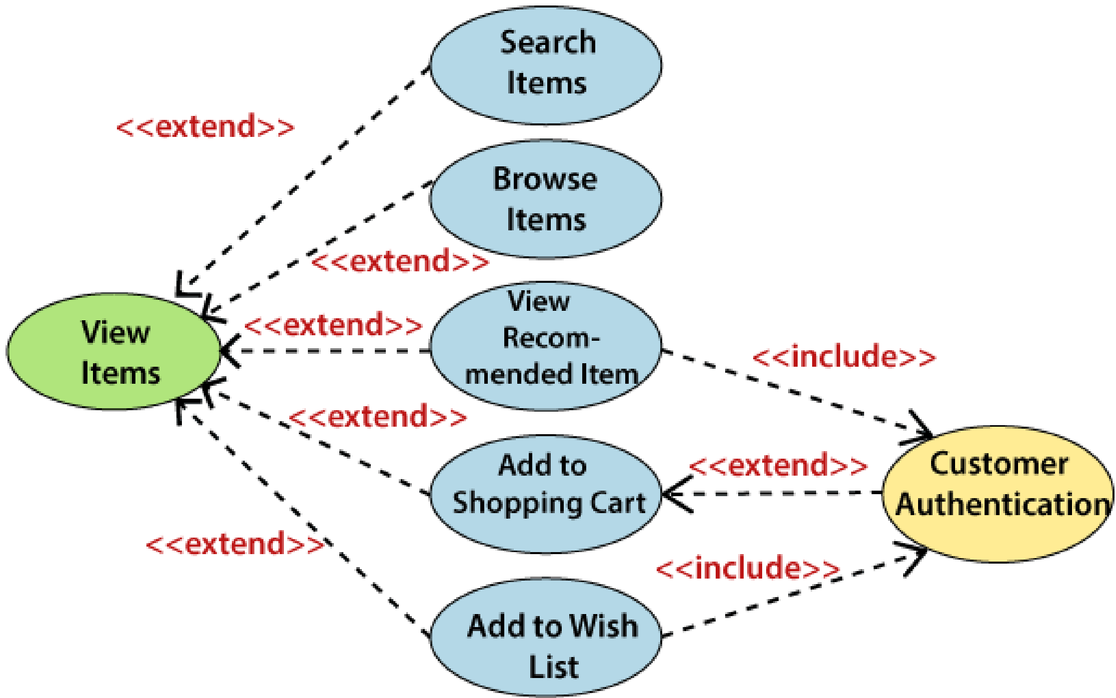
Варіант 16



Варіант 17

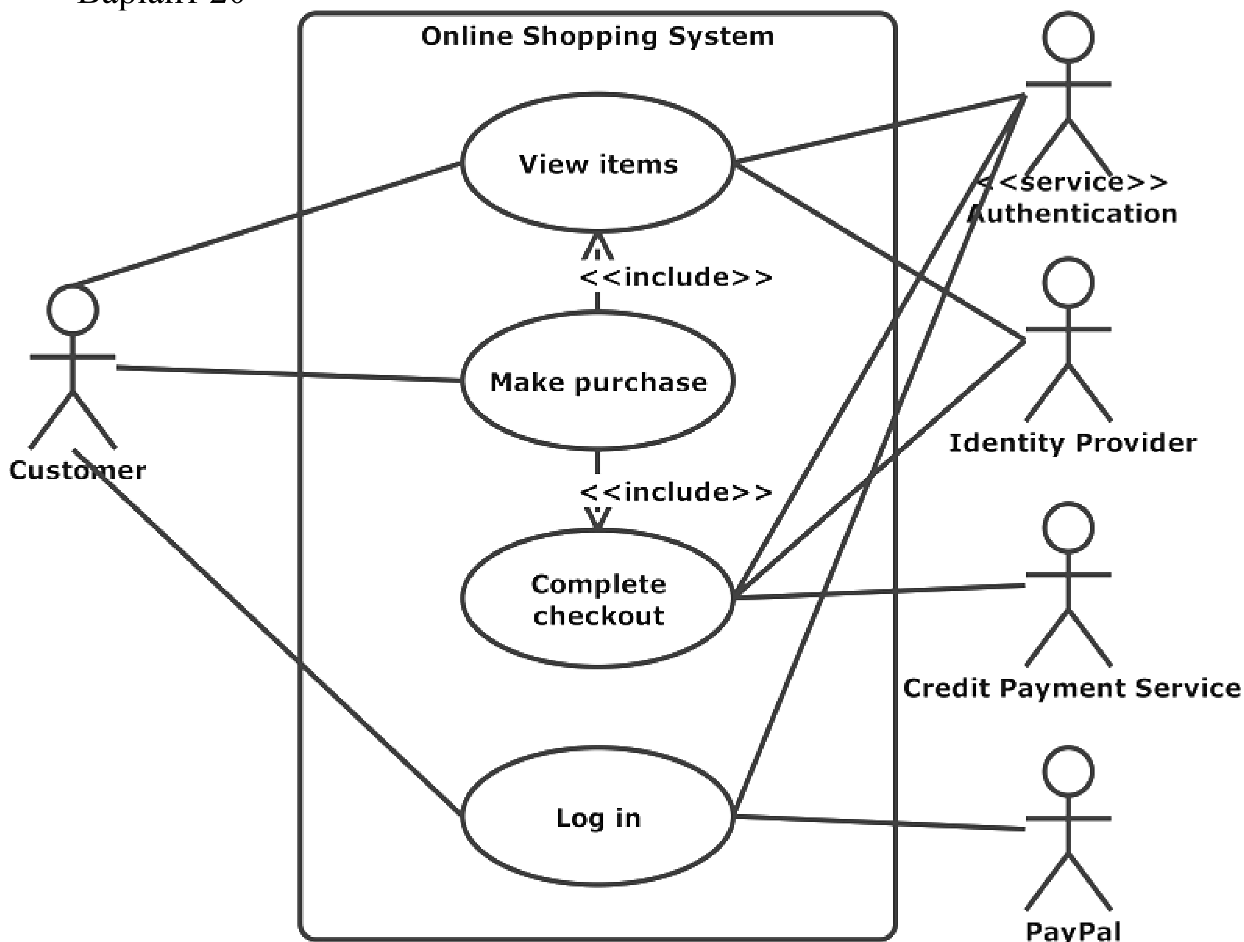
Simple ATM Machine System
Administrator (Third-party)
Bank Customer
Bank
System Maintainence
System Reporting
System Shutdown
<<include>>
<<include>>
Login
Bad PIN
<<include>>
<<extend>>
Transaction
Withdraw
Check Balance
Deposit
Print Receipt

Варіант 18

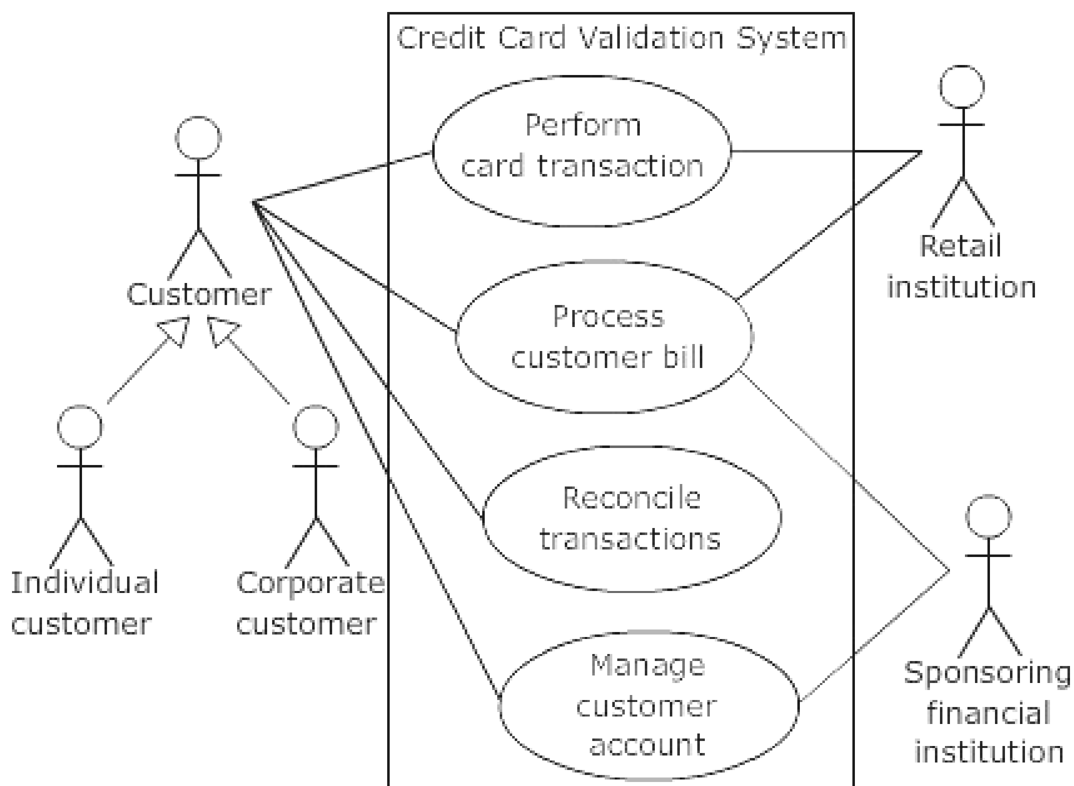


Варіант 19

Варіант 20

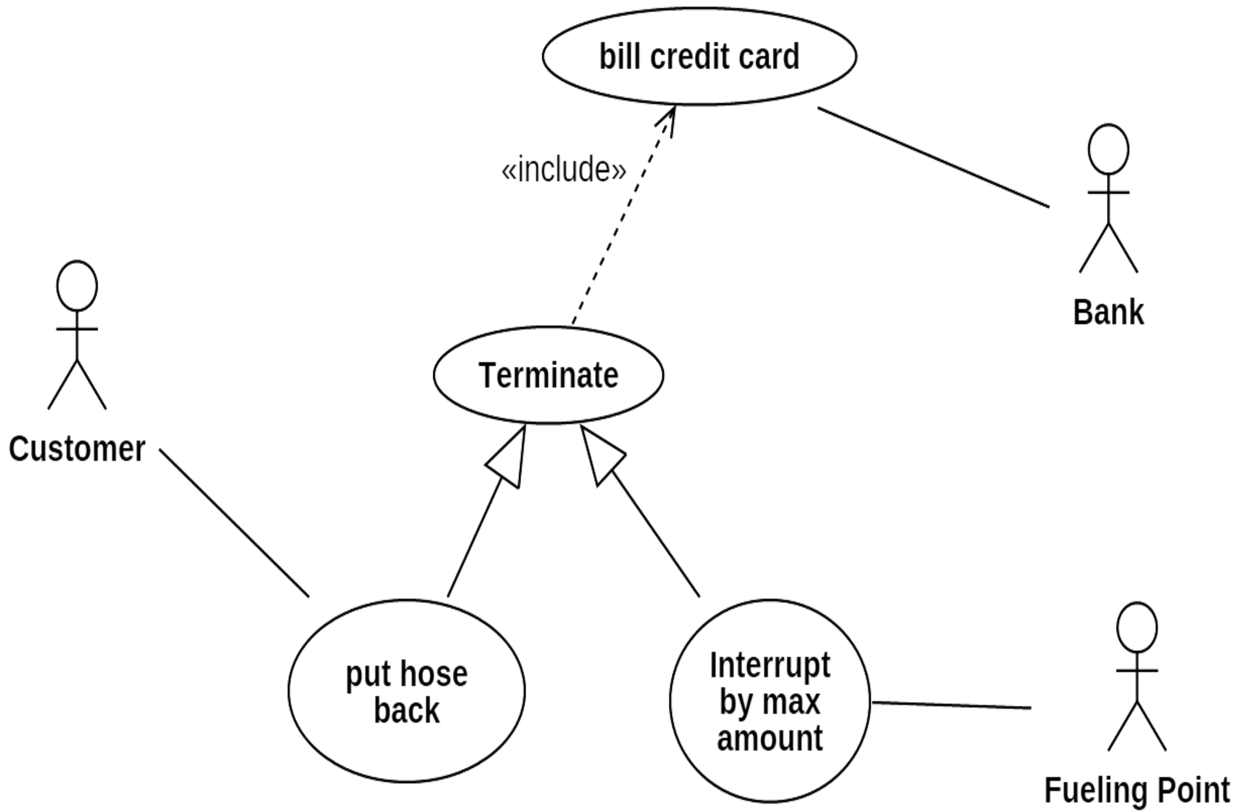


Варіант 21



Modeling the Context of a System

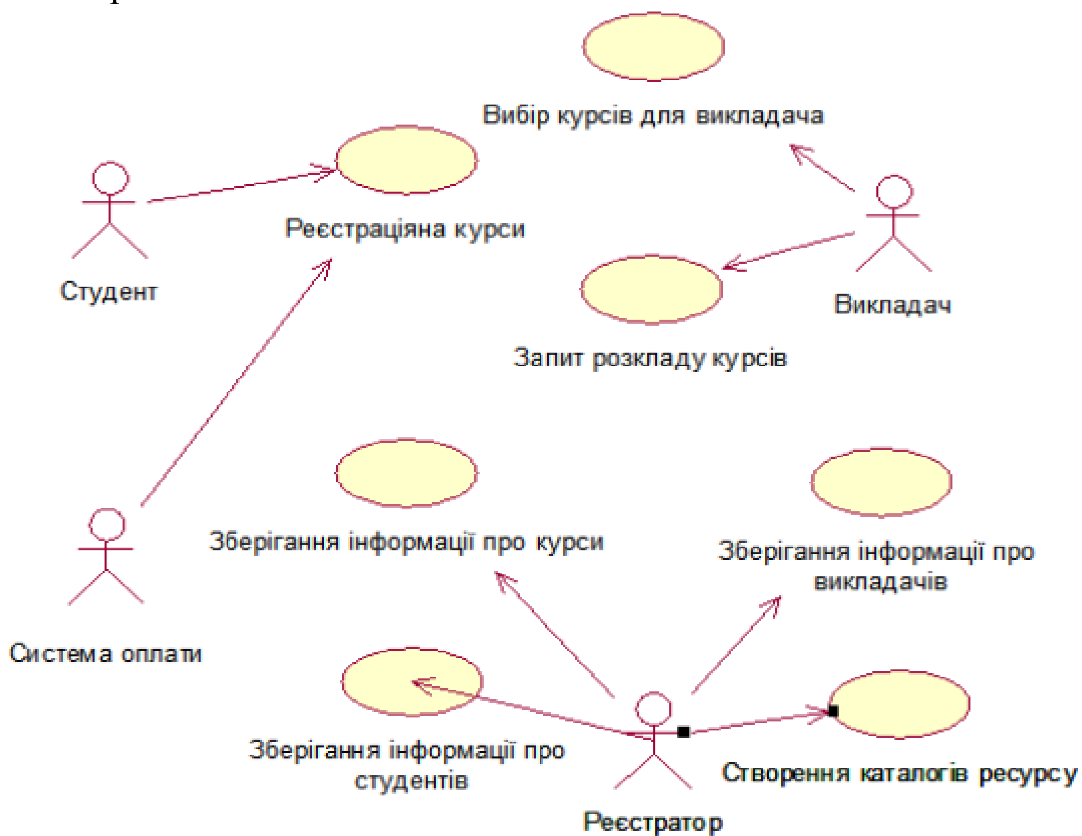
Варіант 22



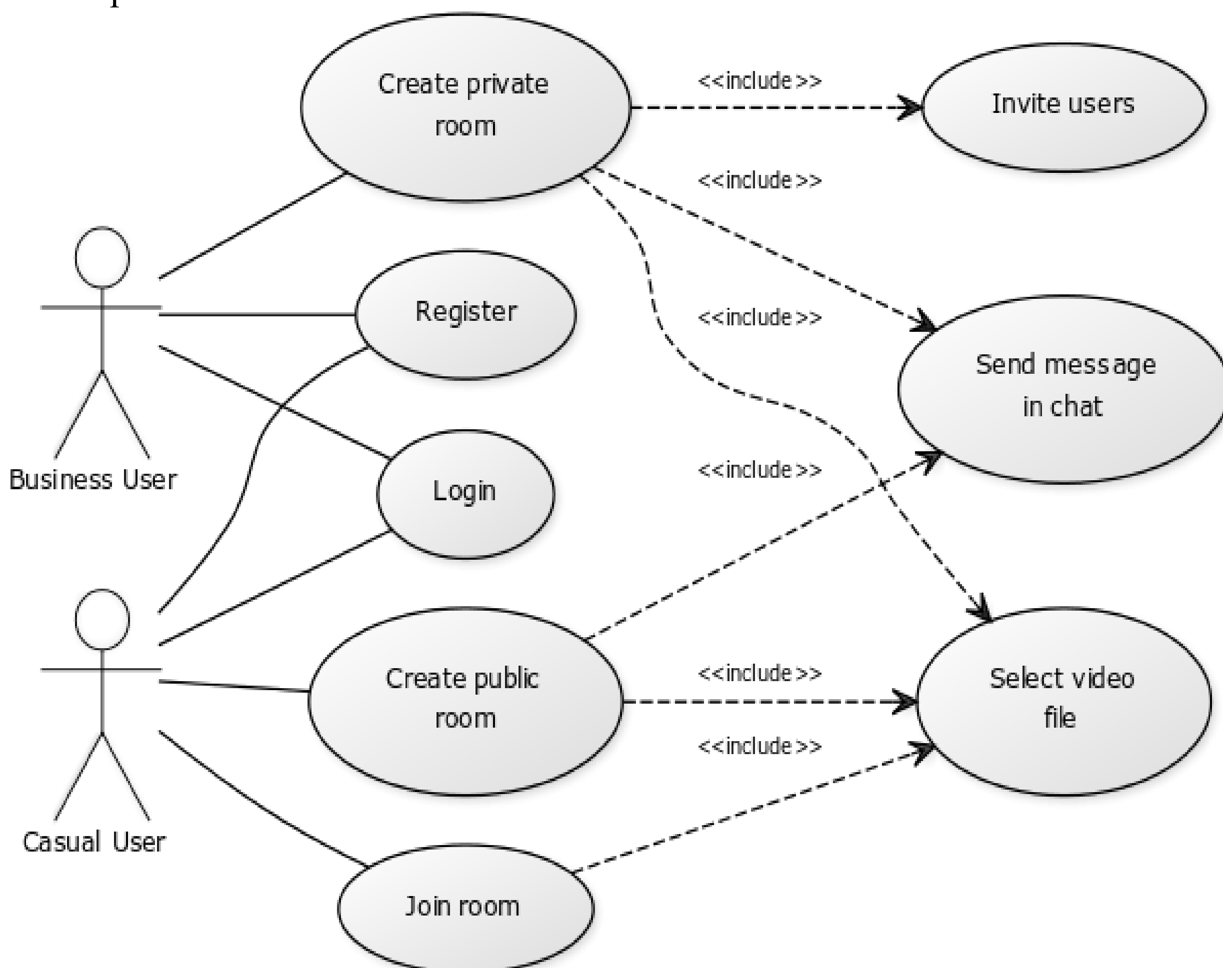
Варіант 23

«Subsystem»
Online Shopping
View Items
«include»
Make Purchase
«include»
Checkout
Client Register
Registered Customer
Web Customer
New Customer
«Service» Authentication
Identity Provider
Credit Payment Service
PayPal

Варіант 24



Варіант 25



Лабораторна робота 5
Створення діаграми варіантів використання (Use Cases Diagrams)

Мета роботи: навчитися створювати діаграми варіантів використання.

Теоретичні відомості

1 Принципи моделювання та елементи UML

Моделювання за допомогою мови UML ґрунтується на таких принципах:

– **абстрагування** – у модель варто долучати тільки ті елементи проєктованої системи, які мають безпосередній стосунок до виконання нею своїх функцій;

– **багатомодельність** – жодна модель не може з достатнім ступенем точності описати різні аспекти системи. Можна описувати систему кількома взаємозалежним поданнями, кожне з яких відображатиме певний бік її структури або поведінки;

– **ієрархічність** – при описі системи використовуються різні рівні абстрагування і деталізації у рамках фіксованих подань. При цьому перше подання системи описує її в найбільш загальних рисах і є поданням концептуального рівня, а наступні рівні розкривають різні сторони системи із зростаючим ступенем деталізації аж до фізичного рівня. Модель фізичного рівня в мові UML відображає компонентний склад проєктованої системи з точки зору її реалізації на апаратній і програмній платформах конкретних виробників.

Елементи (класифікатори) мови UML можна поділити на групи:

1. **Сутності** (entity) – абстракції, що є основними об’єктно-орієнтованими елементами мови UML, за допомогою яких будуються моделі.

В UML визначено чотири типи сутностей: структурні, поведінкові, сутності групування та сутності-примітки.

– *Структурні сутності* – це іменники в моделях UML. Здебільшого вони є статичними частинами моделей, які відповідають концептуальним або фізичним елементам системи. Існує сім різновидів структурних сутностей: Клас (Class), Інтерфейс (Interface), Кооперація (Collaboration), Варіант використання/Прецедент (Use case), Активний клас (Active class), Компонент (Component), Вузол (Node).

– *Поведінкові сутності* є динамічними складовими моделі UML, які описують поведінку моделі в часі і просторі. Це дієслова мови. Існують два основних типи поведінкових сутностей: Взаємодія (Interaction) та Автомат (State).

– *Сутності групування* є організуючими частинами моделі UML. Це блоки, на які можна розкласти модель. Первинна сутність групування – пакет (Package).

– *Сутності-примітки* – пояснювальні частини моделі UML. Це коментарі для додаткового опису, роз'яснення або зауваження до будь-якого елемента моделі. Є тільки один базовий тип анотаційних елементів – примітка (Note).

2. **Стосунки**:

– *Залежність* (dependency) – це семантичний стосунок між двома сутностями, при якому змінення однієї з них, незалежної, може вплинути на семантику іншої, залежної.

– *Асоціація* (association) – структурне відношення, що описує сукупність змістовних або логічних зв'язків між об'єктами.

– *Узагальнення* (generalization) – це стосунок, при якому об'єкт-нащадок (child) може бути підставлений замість об'єкта-батька (parent). При цьому відповідно до принципів об'єктно-орієнтованого програмування нащадок успадковує структуру і поведінку свого батька.

– *Реалізація* (realization) є семантичним стосунком між класифікаторами, при якому один класифікатор визначає зобов'язання, а інший гарантує його виконання.

Діаграми варіантів використання використовуються для відображення сценаріїв використання системи (usecases) та користувачів системи (actors), які використовують її функції.

2 Діаграми варіантів використання (Use Cases Diagram)

Поведінку системи (тобто функціональність, яку вона забезпечує) описують за допомогою функціональної моделі, що відображає системні прецеденти (use cases – випадки використання), системне оточення (actors – дійові особи, актори) і зв'язки між ними (use cases diagrams).

Діаграма варіантів використання (Use Case Diagram – діаграма прецедентів) – це діаграма, на якій зображуються стосунки між акторами і варіантами використання.

За допомогою цієї діаграми можна:

– визначити спільні межі і контекст модельованої предметної області на початкових етапах проєктування системи;

– сформулювати загальні вимоги до функціонального поведінки проєктованої системи;

– розробити вихідну концептуальну модель системи для її подальшої деталізації у формі логічних і фізичних моделей;

– підготувати вихідну документацію для взаємодії розробників системи з її замовниками і користувачами.

Діаграма варіантів використання є графом, у вершинах якого розташовані актори чи прецеденти, а зв'язки між вершинами – це різного роду стосунки.

Діаграми варіантів використання створюються на етапі аналізу вимог до системи. Аналіз починається з ідентифікації діячів (actors – діючі особи, акто-

ри) як потенційних користувачів системи. Для визначення діячів треба розглянути ситуації, типові для використання системи. Користувачами системи не обов'язково мають бути люди; це можуть бути інші (зовнішні) системи, що звертаються до даної системи чи до яких звертається дана система. Варто зауважити, що між діячами й користувачами системи є важлива відмінність: діячі, по суті, – це типи, тоді як користувачів слід розглядати як конкретних екземплярів таких типів. Сутності, що знаходяться поза системою і взаємодіють із нею, складають її контекст. Отже, визначення діячів діаграм прецедентів дозволяє моделювати контекст системи.

На діаграмах варіантів використання діячі позначаються стилізованими людськими фігурками, під якими записуються їхні імена (рис. 5.1).

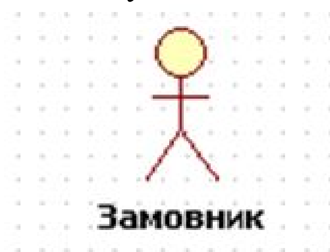


Рисунок 5.1. Дійова особа (актор)

Варіант використання (use case – **прецедент**) – опис послідовності дій, які система виконує при взаємодії з актором. У діаграмах прецедент зображується еліпсом. Здебільшого для іменування прецедентів використовуються дієслова або короткі дієслівні фрази (“Створення таблиці”, “Створити таблицю”).

Прецедент є специфікацією загальних особливостей поведінки або функціонування модельованої системи без розгляду внутрішньої структури цієї системи. Незважаючи на те, що кожен варіант використання визначає послідовність дій, які повинні бути виконані системою при взаємодії її з відповідним актором, самі ці дії не зображуються на діаграмі.

Потоки подій описуються мовою предметної області, а не в термінах реалізації. Найчастіше для опису потоку подій пропонується така структура:

– заголовок (наприклад, “Потік подій для прецеденту <Зняти гроші>”);

– короткий опис потоку (наприклад, “Дозволяє клієнту зняти гроші з його рахунку”);

– передумови (pre-conditions) (діаграми прецедентів не дозволяють відображати послідовний характер використання прецедентів!);

– головний потік подій та, можливо, його підпотоки;

– альтернативні потоки подій;

– постумови (post-conditions).

Зміст варіанта використання може бути подано у формі додаткового пояснювального тексту, який розкриває семантику дій при виконанні даного варіанта використання. Такий пояснювальний текст має назву тексту-сценарію або просто сценарію.

Окремий варіант використання позначається на діаграмі еліпсом, всередині якого міститься коротке ім'я у формі іменника або іменника з пояснюваль-

ними словами. Сам текст імені варіанти використання повинен починатися з великої літери (рис. 5.2).

Ім'я (name) – рядок тексту, який використовується для ідентифікації будь-якого елемента моделі.

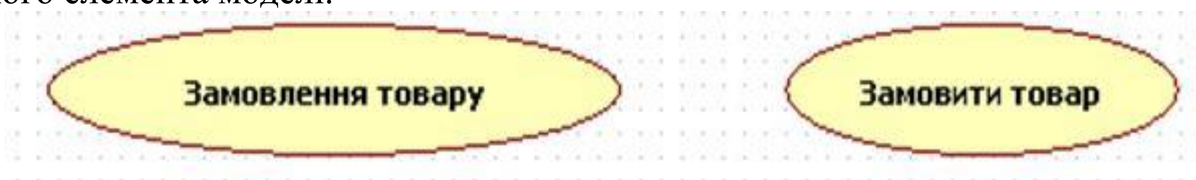


Рисунок 5.2. Варіанти використання (прецеденти)

Одним з найважливіших етапів проєктування інформаційних систем є етап визначення вимог до системи. Якщо вимоги замовника інформаційної системи розробниками будуть визначені некоректно, то в підсумку замовник може отримати зовсім не ту систему, яку він замовляв.

Моделювання прецедентів та акторів допомагає краще зрозуміти вимоги, що висуваються до системи, і погодити їх із замовником за допомогою демонстрації та обговорення діаграми прецедентів. Прецеденти та актори – це відображення вимог до системи, вони показують, хто і для чого буде використовувати майбутню систему.

3 Стосунки між прецедентами та акторами

Між елементами діаграми варіантів використання можуть існувати різні стосунки, які описують взаємодію примірників одних акторів і варіантів використання з примірниками інших акторів і варіантів. Один актор може взаємодіяти з декількома варіантами використання, при цьому цей актор зазвичай звертається до кількох сервісів даної системи. У свою чергу один варіант використання може взаємодіяти з кількома акторами, надаючи для всіх них свій сервіс.

Водночас два варіанти використання, визначені в рамках однієї системи, що моделюється, також можуть взаємодіяти один з одним, проте характер цієї взаємодії буде відрізнятися від взаємодії з акторами. Але в обох випадках способи взаємодії елементів моделі припускають обмін сигналами або повідомленнями, які ініціюють реалізацію функціональної поведінки модельованої системи.

У мові UML є декілька стандартних **видів стосунків** між акторами і варіантами використання:

– асоціації (association relationship);
– включення (include relationship);
– розширення (extend relationship);
– узагальнення (generalization relationship).

При цьому загальні властивості варіантів використання можуть бути подані трьома різними способами, а саме – за допомогою стосунків включення, розширення та узагальнення.

Стосунок асоціації – одне з фундаментальних понять у мові UML, яке використовується при побудові всіх графічних моделей систем у формі каноніч-

них діаграм. Так на діаграмах варіантів асоціації використовують для позначення специфічної ролі актора при його взаємодії з окремим варіантом використання. Іншими словами, асоціація специфікує семантичні особливості взаємодії акторів і варіантів використання графічної моделі системи.

На діаграмі варіантів використання, так само як і на інших діаграмах, стосунок асоціації позначається суцільною лінією між актором і варіантом використання (рис. 5.3). Ця лінія може мати деякі додаткові позначення, наприклад, ім'я і кратність.

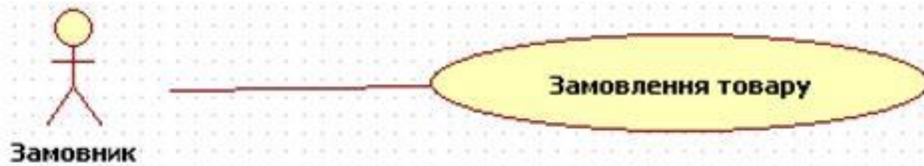


Рисунок 5.3. Стосунок асоціації між актором і прецедентом

У контексті діаграми варіантів використання стосунку асоціації між актором і варіантом використання може вказувати на те, що актор ініціює відповідний варіант використання. Такого актора називають головним. В інших випадках подібна асоціація може вказувати на актора, якому надається довідкова інформація про результати функціонування модельованої системи. Таких акторів часто називають другорядними.

Стосунок включення (include) в UML – це різновид стосунків залежності між базовим варіантом використання та його спеціальним випадком. При цьому стосунком залежності (dependency) є такий стосунок між двома елементами моделі, при якому змінення одного елемента (незалежного) призводить до змінення іншого елемента (залежного).

Стосунок включення встановлюється тільки між двома варіантами використання і вказує на те, що задана поведінка для одного варіанта використання долучається як складовий фрагмент у послідовність поведінки іншого варіанта використання.

Так, наприклад, стосунок включення, спрямований від варіанта використання "Надання кредиту в банку" до варіанта використання "Перевірка платоспроможності клієнта", вказує на те, що кожен примірник першого варіанта використання завжди включає в себе функціональну поведінку або виконання другого варіанта використання. У цьому сенсі поведінка другого варіанта використання є частиною поведінки першого варіанта використання на даній діаграмі. Графічно цей стосунок позначається у формі пунктирної лінії зі стрілкою, спрямованої завжди від базового варіанта використання у бік того елемента, від якого щось потрібно, чиїми сервісами користуються. При цьому дана лінія позначається стереотипом <<include>>, як показано на рис. 5.4.

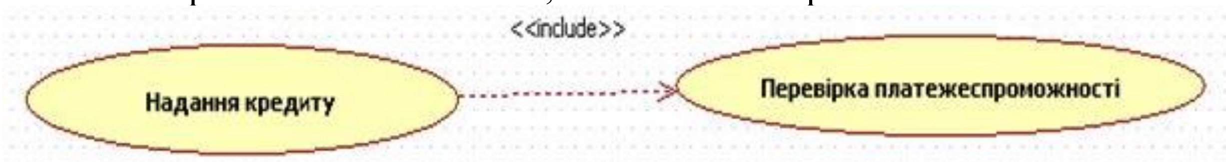


Рисунок 5.4. Стосунок включення між прецедентами

Стосунок розширення (extend) визначає взаємозв'язок базового варіанта використання з іншим варіантом використання, функціональна поведінка якого задіюється базовим варіантом не завжди, а лише за виконання додаткових умов.

У мові UML стосунок розширення є залежністю, спрямованою до базового варіанта використання і з'єднаною з ним у так званому місці розширення. Стосунок розширення між варіантами використання позначається як стосунок залежності у формі пунктирної лінії зі стрілкою, спрямованою від того варіанта використання, який є розширенням, до базового варіанта використання. Така лінія зі стрілкою повинна бути позначена стереотипом <<extend>>, як показано на рис. 5.5.

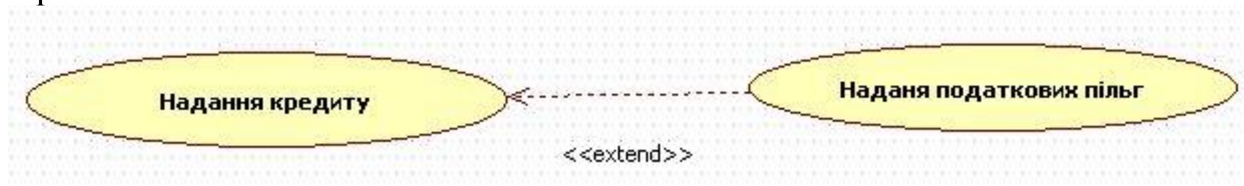


Рисунок 5.5. Приклад графічного зображення стосунків розширення між варіантами використання

У наведеному фрагменті має місце стосунок розширення між базовим варіантом використання "Надання кредиту" і варіантом використання "Надання податкових пільг". Це означає, що властивості поведінки першого варіанта використання в деяких випадках можуть бути доповнені функціональністю другого варіанта використання. Для того щоб це розширення мало місце, має виконуватись певна логічна умова даного стосунку розширення.

Щоб усвідомити собі сенс розширення, уявімо собі, що ми говоримо про оплату деякого купленого нами товару. Ми можемо оплатити товар готівкою, якщо сума не перевищує 1000 грн. Або оплатити кредитною карткою, якщо сума у межах від 1000 до 10000 грн. Якщо ж сума перевищує 10000 грн, доведеться брати кредит. Отже ми розширили розуміння операції оплати купленого товару і на випадки, коли використовуються інші засоби оплати, ніж готівка. Але самі ці випадки виникають тільки за строго певних умов: коли ціна товару потрапляє в певні рамки.

Отже, розширення доповнює прецедент іншими прецедентами, які "спрацьовують" за певних умов, – просто додає в вихідний прецедент послідовність дій, що міститься в іншому прецеденті. Стосунок розширення прецеденту А до прецеденту В означає, що екземпляр прецеденту В може мати (за певних умов, які можуть бути описані в розширенні) поведінку, описану в прецеденті А.

Позначення стосунків <<include>> та <<extend>> є не що інше як позначення стереотипів, які широко використовуються в UML для створення нових елементів моделі шляхом розширення функціональності базових елементів.

Стереотип (Stereotype) – це механізм, що дозволяє класифікувати елементи моделі. За допомоги стереотипів можна створювати свого роду підтипи типів. Це дозволяє UML мати мінімальний набір елементів, які за потреби можуть бути доповнені задля створення сполучних базових елементів у системі. В UML

стереотип позначається ім'ям, яке записується в подвійних кутових дужках: <<ім'я стереотипу>>.

В UML можна створювати власні стереотипи на базі вже наявних типів, проте є стандартні стереотипи, заздалегідь визначені у нотації UML. Так, стосунок залежності розширюється для прецедентів та акторів за допомогою двох стереотипів <<include>> та <<extend>>.

Узагальнення (Generalization) – це стосунок між загальною сутністю та її конкретним втіленням. На діаграмах узагальнення позначається стрілкою з незамальованим трикутником наприкінці, спрямованим від приватного елемента до загального.

Стосунок узагальнення між варіантами використання застосовується, коли необхідно відзначити, що дочірні варіанти використання володіють усіма особливостями поведінки батьківських варіантів. При цьому дочірні варіанти використання беруть участь в усіх стосунках батьківських варіантів. У свою чергу, дочірні варіанти можуть наділятися новими властивостями поведінки, які відсутні у батьківських варіантів використання, а також уточнювати або модифікувати успадковані від них властивості поведінки.

Стосунок узагальнення можуть формуватися як між акторами (рис. 5.6), так і між прецедентами (рис. 5.7).

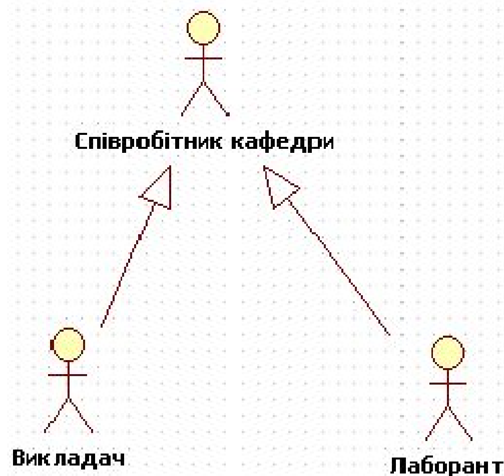


Рисунок 5.6. Стосунок узагальнення між акторами

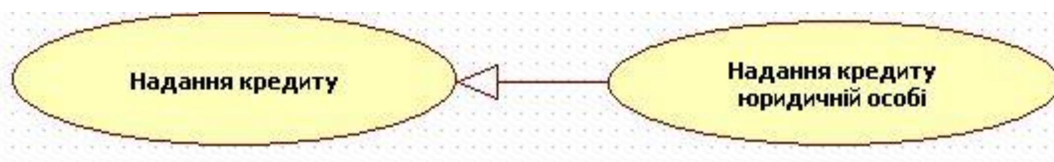
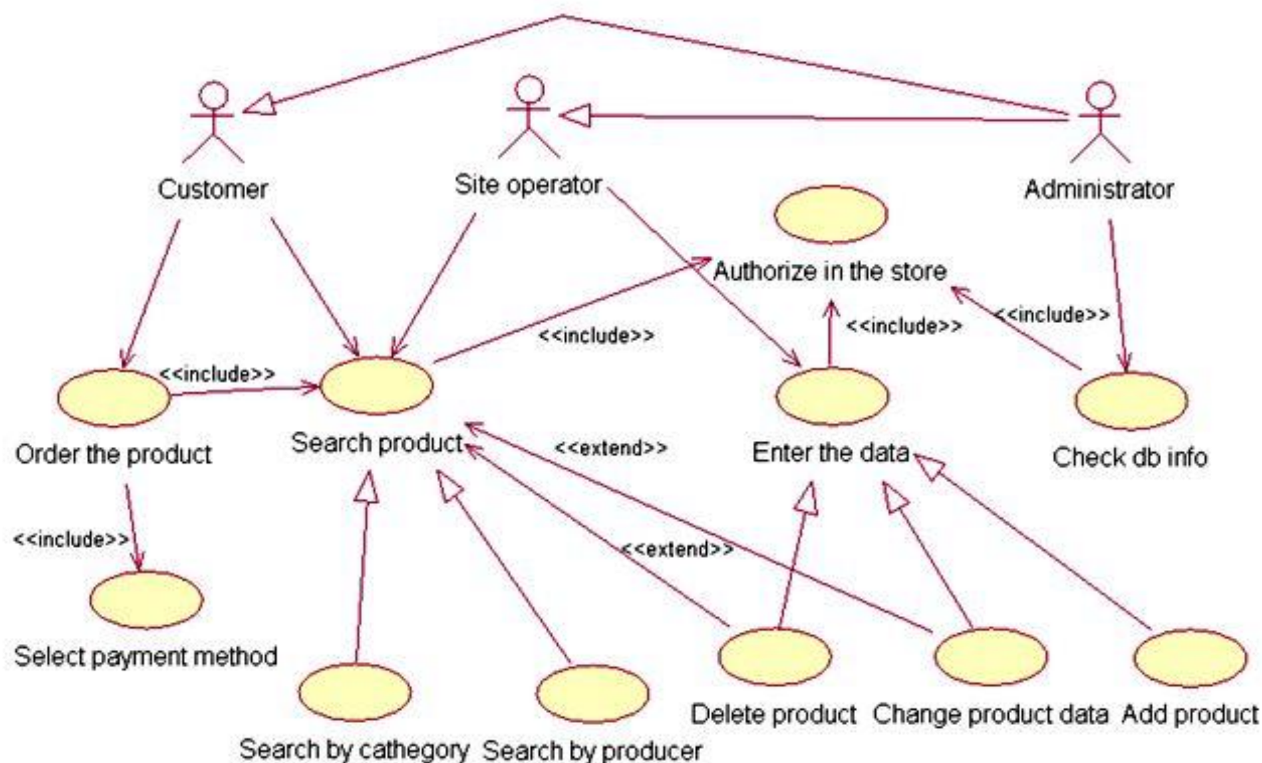


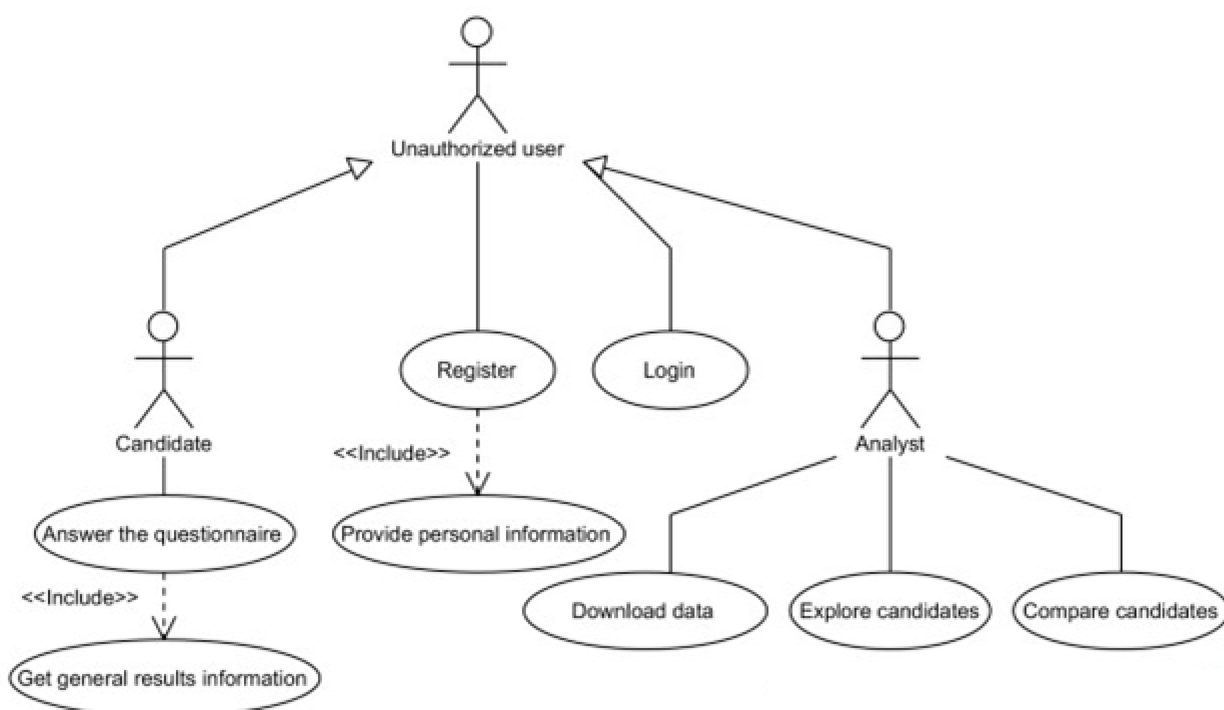
Рисунок 5.7. Стосунок узагальнення між прецедентами

Актори, прецеденти і стосунки – це основні елементи нотації діаграм варіантів використання. Діаграма варіантів використання допомагає відобразити основні вимоги до системі і забезпечити взаєморозуміння функціональності системи між розробником і замовником. Можна побудувати одну головну діаграму прецедентів, на якій будуть відображені межі системи (актори) та її основна

функціональність (прецеденти). Для більш детального подання системи допускається побудова допоміжних діаграм прецедентів.

Далі на рисунках показано діаграми варіантів використання для інтернет-магазину.





4 Побудова діаграми прецедентів у StarUML

У StarUML головна діаграма прецедентів називається *Main* і розташовується в поданні *Use Case*. Якщо в навігаторі моделі клацнути двічі по імені цієї діаграми, то відкриється робоче поле. Для того щоб створити прецедент, треба клацнути по овальному символу прецеденту на панелі елементів ліворуч від робочого поля діаграми, а потім вибрати місце на робочому полі діаграми, куди треба помістити прецедент. Аналогічно створюється актор. Коли елемент розміщений на полі діаграми, він стає доступним для редагування імені та деяких властивостей. У виділене поле треба ввести нове ім'я прецеденту або актора (рис. 5.8).

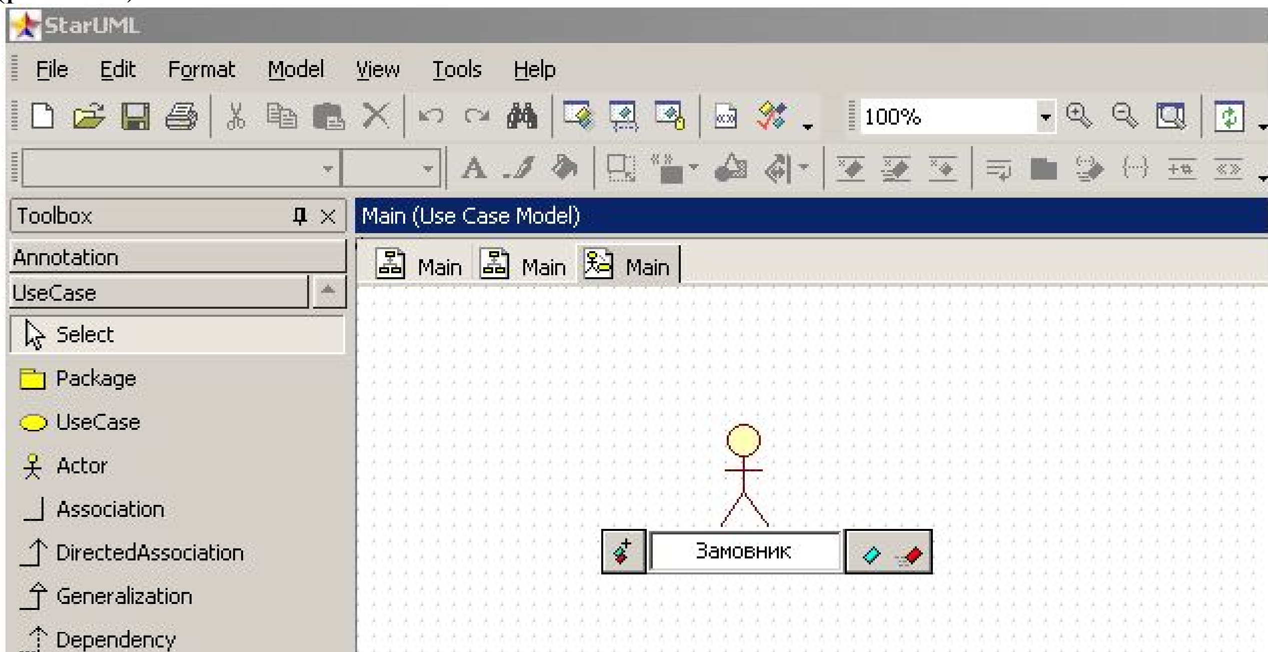


Рисунок 5.8. Іменування елементів у StarUML

Для створення зв'язків між елементами діаграми треба клацнути по зображенню відповідного стосунку на панелі елементів праворуч, а потім провести лінію від одного елемента до іншого, утримуючи ліву кнопку миші (рис. 5.9).

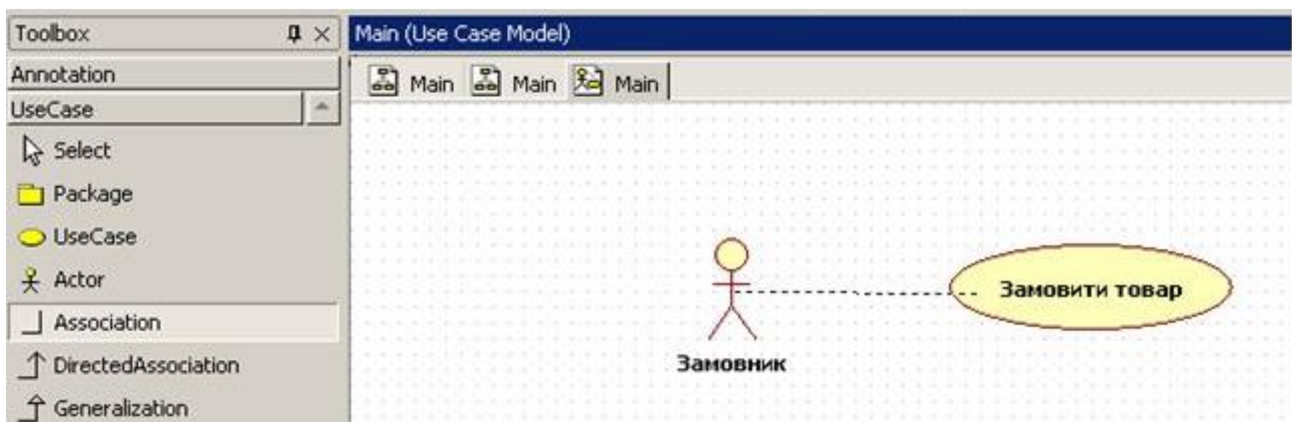


Рисунок 5.9. Створення зв'язків між елементами в StarUML

Щоб видалити елемент з діаграми достатньо клацнути лівою кнопкою миші по цьому елементу, а потім натиснути кнопку *Delete* або клацнути правою кнопкою миші по елементу і вибрати команду контекстного меню *Edit / Delete*.

При цьому елемент буде видалений з діаграми, але не з моделі. Його можна найти в навігаторі моделі, не зважаючи на те, що на діаграмі він більше не відображається. Тобто можна повернути елемент на діаграму. Для цього треба просто перетягнути елемент з навігатора моделі на поле діаграми.

Для того щоб видалити елемент з моделі потрібно клацнути по ньому на діаграмі або по його зображенню у навігаторі моделі правою кнопкою миші і з контекстного меню вибрати пункт *Delete from Model*. Елемент буде повністю видалений.

Описані вище способи додавання та видалення елементів і стосунків можуть бути використані для побудови діаграм будь-яких типів.

5 Документування елементів моделі у StarUML

У StarUML для додавання опису елементу моделі треба виділити цей елемент, клацнувши по ньому мишкою, і відкрити редактор *Documentation*. Якщо він не відображається праворуч на одній із вкладок інспектора моделі, то відкрити його, використовуючи меню *View / Documentation*. Напроти пункту *Documentation* має стояти галочка. Далі варто ввести опис елемента у вікно документування (рис. 5.10).

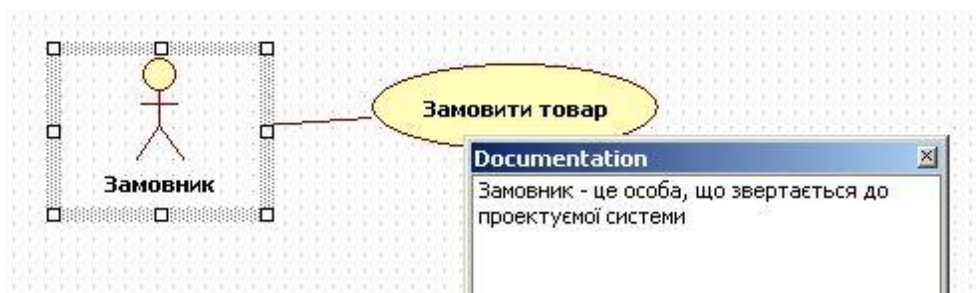


Рисунок 5.10. Документування елемента моделі в StarUML

Усі елементи моделі повинні бути задокументовані. Описаний вище спосіб підходить для будь-якого елемента будь-якої діаграми.

Для того щоб створити ще одну діаграму (будь-якого типу), наприклад, для деталізації прецеденту, треба клацнути правою кнопкою миші на папці *Use Case Model* і з контекстного меню вибрати *Add Diagram*, а потім вибрати з списку діаграму, яку потрібно додати. Наприклад, можна створити додаткову діаграму прецедентів, вибравши пункт *Use Case Diagram* (рис. 5.11).

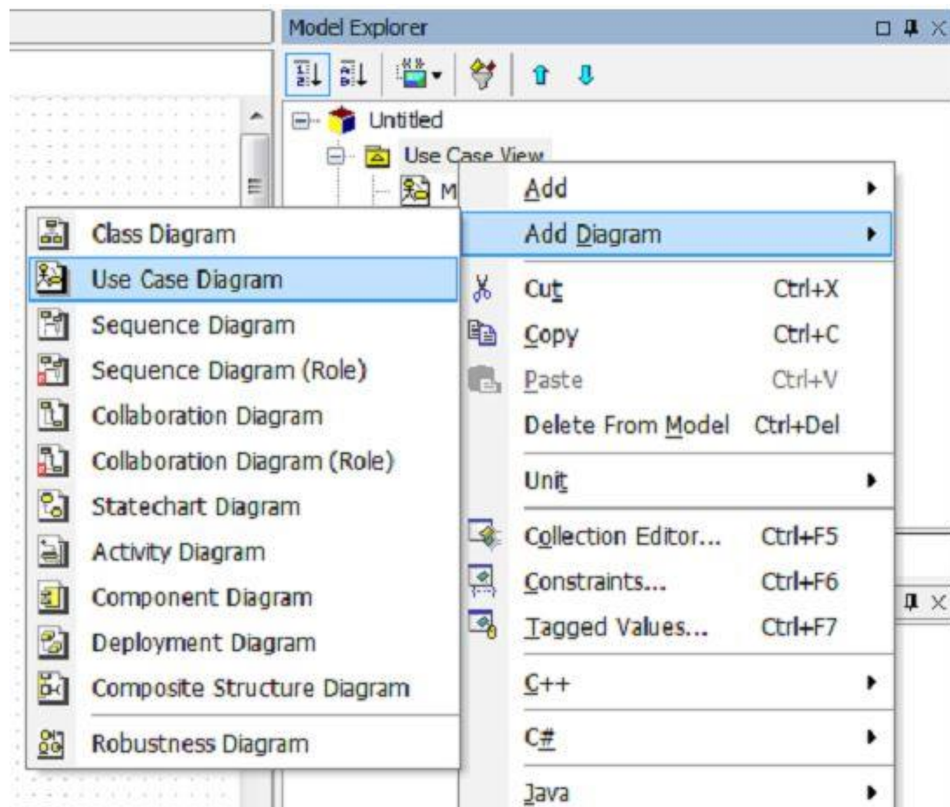


Рисунок 5.11. Створення додаткової діаграми прецедентів

Контрольні запитання для самоконтролю

1. Яке призначення діаграми варіантів використання (Use Case Diagram)?
2. Дати визначення актора (actor) програмної системи. Хто може виступати в ролі акторів?
3. Що таке асоціація в діаграмі варіантів використання?
4. Дати визначення варіанта використання (usecase). Як варіант використання визначається на діаграмі?
5. Перелічити стосунки, які можуть використовуватися на діаграмі.
6. Як показують стосунок включення в діаграмі варіантів використання?
7. Що пов'язує стосунок розширення?
8. Дати визначення стосунку асоціації (association). Навести приклад.
9. У чому відмінність між стосунком узагальнення між акторами та між варіантами використання?
10. Які стосунки можуть використовуватися для поєднання тільки акторів, акторів та варіантів використання, тільки варіантів використання?

Лабораторне завдання

1. Дати відповіді на контрольні питання.

2. Створити діаграму варіантів використання відповідно до індивідуального варіанта.

3. Оформити протокол лабораторної роботи.

Вар.	Завдання
1.	Розробити модель ІС бібліотеки (елементи: керівник, бібліотекар, читач)
2.	Розробити модель ІС рекламної фірми (елементи: керівник, співробітник по роботі з клієнтами, клієнт)
3.	Розробити модель ІС відеосалону (елементи: керівник, співробітник, клієнт)
4.	Розробити модель ІС магазину парфумерії (елементи: керівник, співробітник, клієнт)
5.	Розробити модель ІС ресторану (елементи: керівник, офіціант, клієнт)
6.	Розробити модель ІС організації по роботі з абітурієнтами (елементи: керівник, співробітник організації, абітурієнт)
7.	Розробити модель ІВ середньої школи (елементи: директор, викладач, учень)
8.	Розробити модель ІС провайдера Інтернет (елементи: керівник, співробітник по роботі з клієнтами, клієнт)
9.	Розробити модель ІС роботи військкомату (елементи: керівник, співробітник по роботі з призовниками, призовник)
10.	Розробити модель ІС роботи центру зайнятості (елементи: керівник, співробітник по роботі з безробітними, безробітний)
11.	Розробити модель ІС системи охорони підприємства (елементи: керівник підприємства, співробітник підприємства, охоронець)
12.	Розробити модель ІС університету (елементи: ректор, декан, студент)
13.	Розробити модель ІС супермаркету (елементи: директор, касир, покупець)
14.	Розробити модель ІС чемпіонату з футболу (елементи: співробітник по роботі з футболістами, футболіст, голова федерації футболу)
15.	Розробити модель ІС олімпійських ігор (елементи: уболівальник, співробітник по роботі з уболівальниками, керівник олімпійського комітету)
16.	Розробити модель ІС зоопарку (елементи: відвідувач, директор, касир)
17.	Розробити модель ІС театру (елементи: актор, директор, адміністратор)
18.	Розробити модель ІС страхового агентства (елементи: клієнт, директор, юрист)
19.	Розробити модель ІС весільного салону (елементи: директор, співробітник по роботі з клієнтами, клієнт)
20.	Розробити модель ІС туристичної фірми (елементи: директор, співробітник по роботі з клієнтами, клієнт)
21.	Розробити модель ІС перукарні (елементи: керівник, перукар, клієнт)
22.	Розробити модель ІС піцерії (елементи: керівник, офіціант, клієнт)
23.	Розробити модель ІС аукціонного будинку (елементи: керівник, співробітник, аукціоніст)
24.	Розробити модель ІС автопарк (елементи: директор, клієнт, водій)
25.	Розробити модель ІС салону з продажу мобільних телефонів (елементи: директор, продавець-консультант, клієнт)

Лабораторна робота 6
Створення діаграми послідовностей (Sequence Diagrams)

Мета роботи: навчитися створювати діаграми послідовностей.

Теоретичні відомості

1 Діаграми взаємодії

Діаграми взаємодії відображають один із процесів оброблення інформації: які об'єкти потрібні потоку, якими обмінюються повідомленнями об'єкти, які дійові особи ініціюють потік і в якій послідовності відправляються повідомлення. Для одного потоку подій може бути побудовано декілька діаграм взаємодії.

Основний елемент діаграм взаємодії – це **об'єкт**. Об'єктом описують щось, що містить в собі дані і поведінку. Цим терміном описують реальні, конкретні предмети або абстрактні сутності. Об'єкти зображуються у вигляді прямокутників, імена об'єктів підкреслюються. Усередині прямокутника, що позначає об'єкт, записується з великої літери ім'я об'єкта. Ім'я об'єкта підкреслюється. Якщо воно містить декілька слів, то всі вони повинні починатися з великої літери.

Як приклад розглянемо систему телефонного зв’язку, для якої треба побудувати діаграму взаємодії з використання об’єктів типу Телефон. Крім того, можна вказати якийсь конкретний об’єкт типу телефона, наприклад дротовий телефон (рис. 6.1).

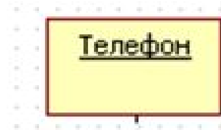


Рисунок 6.1. Приклад іменування об'єктів

Взагалі об'єкти на діаграмі взаємодії здебільшого є об'єктами системи і стосуються програмного забезпечення. При проєктуванні таких діаграм об'єкти можна подавати як екранні форми або частини програми, які відповідають за виконання певних дій, або ж об'єктом може бути запис у таблиці бази даних.

Якщо перед тим, як будувати діаграми взаємодії, були побудовані діаграми класів, то пошук об'єктів спрощується, позаяк об'єкти відповідають своїм класам або їх операціями, і можна створювати і розташовувати їх на діаграмі послідовності дій або кооперативної діаграмі.

Щоб визначити взаємодію об'єктів у разі, коли класи ще не розглядались, пошук об'єктів можна почати з вивчення імен іменників у потоці подій. Більшість з них стануть кандидатами в об'єкти. Також можна виділити об'єкти-суті,

граничні об'єкти і керуючі об'єкти на базі вибраних класів.

Існує два типи діаграм взаємодії – діаграми послідовності (Sequence Diagram) і діаграми кооперації (Collaboration). Перші відображають обмін повідомленнями між об'єктами в часі, а другі відображають структуру взаємодії. На обох діаграмах зображується одна й та сама інформація, але в різний спосіб: діаграма послідовностей зображує потік управління, а кооперативна діаграма – потік даних.

2 Діаграми послідовності

Звичайно потік подій описує не одну послідовність дій, а кілька можливих. Це відбивається наявністю головного потоку подій та альтернативних потоків. Найчастіше неможливо описати прецедент за допомогою тільки однієї послідовності дій. Наприклад, для прецеденту Замовлення товарів можливе оформлення замовлення без змінення вмісту кошику, або ж покупець може повернутися до каталогу і вибрати у кошик (або тільки переглядати) інші товари, після чого повернутися у кошик та оформити замовлення. Кожен такий варіант можна описати своєю послідовністю дій, своїм сценарієм. А, отже, один прецедент описує кілька послідовностей – сценаріїв, кожен з яких описує один з варіантів можливого потоку подій.

Сценарій (Scenario) – це деяка послідовність дій, що ілюструє поведінку системи. Сценарій – це екземпляр потоку подій. Він являє собою одиночний прохід по потоку подій для прецеденту. Для графічного відображення сценарію використовуються діаграми послідовностей.

Діаграма послідовності дій відображає взаємодію об'єктів, упорядковану за часом.

2.1 Основні елементи нотації діаграм послідовності

На діаграмах послідовності зображуються об'єкти, класи і послідовність повідомлень, якими обмінюються об'єкти в ході виконання сценарію (рис. 6.2).

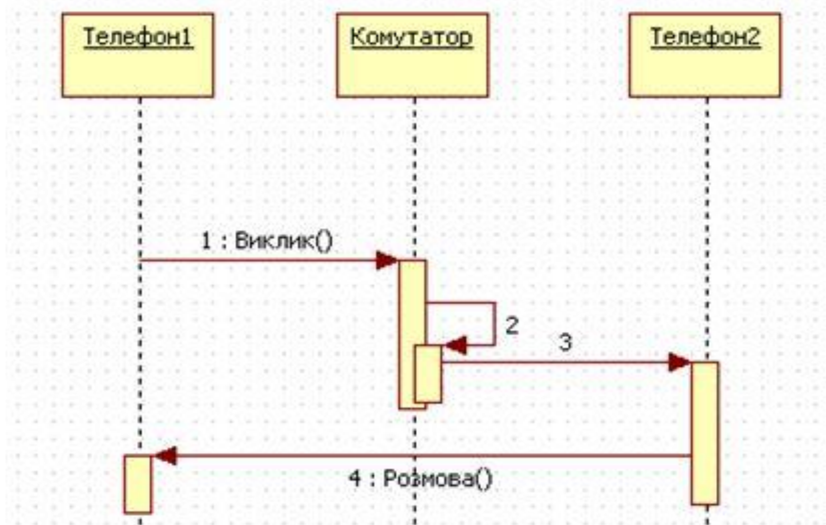


Рисунок 6.2. Загальний вигляд діаграми послідовності

На діаграмі послідовностей можуть зображуватися примірники дійових осіб. Для того щоб помістити дійову особа на діаграму, потрібно знайти його в навігаторі моделі праворуч і перетягнути на поле діаграми послідовностей (рис. 6.3).

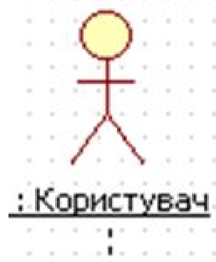


Рисунок 6.3. Примірник дійової особи на діаграмі послідовності

Діючі особи, присутні на діаграмах взаємодії, виділяються з потоку подій як сутності, що запускають процеси. На одній діаграмі їх може бути кілька. Для того, щоб помістити примірник вже створеного раніше на діаграмі взаємодії дійової особи на діаграму послідовності, треба просто перетягнути його з навігатора моделі на робоче поле діаграми.

Кожний об'єкт або дійова особа на діаграмі послідовностей має свою лінію життя, яка позначається пунктиром.

Лінія життя об'єкта (object lifeline) – вертикальна пунктирна лінія на діаграмі послідовності, яка задає існування об'єкта протягом певного періоду часу.

Фокус управління (активність, focus of control) – спеціальний символ на діаграмі послідовності, що вказує період часу, протягом якого об'єкт виконує певну дію, перебуваючи в активному стані. Фокус управління зображується тонким прямокутником, розташованим на лінії.

Іноді відображення фокуса активності і нумерації повідомлень на діаграмі можуть зробити її важкою для читання. Щоб фокус управління і нумерація повідомлень не відображалися на діаграмі послідовності, в StarUML потрібно відкрити редактор властивостей цієї діаграми в інспекторі моделі і в розділах ShowSequenceNumber і ShowActivation прибрати «галочки» (рис. 6.4).

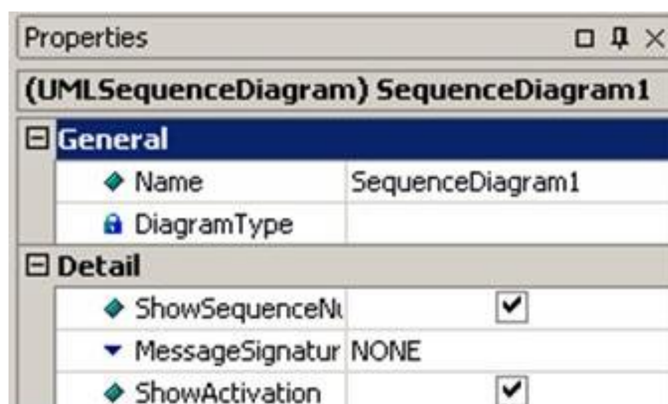


Рисунок 6.4. Налаштування відображення фокуса управління і нумерації повідомлень

Об'єкти та дійові особи на діаграмах послідовності обмінюються повідомленнями. Повідомлення позначаються стрілками, що йдуть від відправника до одержувача.

Повідомлення (message) – специфікація передачі інформації від одного елемента моделі до іншого з очікуванням виконання певних дій з боку приймаючого елемента (рис. 6.5).

Рисунок 6.5. Повідомлення

Для повідомлень на діаграмах послідовностей, як і для інших елементів моделі, доступний ряд специфікацій.

По-перше, у кожного повідомлення має бути ім'я, відповідне його меті.

По-друге, повідомлення на діаграмах послідовностей можна зіставити з операціями, визначеними для класів. Якщо від одного об'єкта до іншого направлено повідомлення, то це означає, що об'єкт-джерело викликає операцію об'єкта-приймача. Об'єкт не може викликати будь-яку операцію: вона має бути доступна цьому об'єкту.

В особливих випадках повідомлення не стає операцією: наприклад, введення логіна і пароля передбачає їхнє посимвольне набирання у відповідних полях, і повідомлення при цьому буде реалізовано у вигляді поля у вікні програми. Процедура створення операцій з повідомлень буде описана далі.

По-третє, для кожного повідомлення можна встановити тип синхронізації. Кожному типу відповідає його позначення.

Виклик операції (процедури) (call) викликає операцію того об'єкта, до якого направлено. Об'єкт може викликати свою операцію. Тоді стрілка починається і закінчується на лінії життя одного і того самого об'єкта, таке повідомлення називається рефлексивним.

Синхронне повідомлення позначається стрілкою з замальованою стрілкою.

Асинхронне повідомлення (send) надсилає об'єкту сигнал. При цьому джерело не чекає відгуку приймача або підтвердження отримання, а продовжує свою роботу. Позначається незаповненою (нежирною) стрілкою (рис. 6.6).

Рисунок 6.6. Асинхронне повідомлення

Повідомлення відповіді (return) повертає значення з процедури того об'єкта, до якого направлено. Позначається пунктирною стрілкою (рис. 6.7).

Рисунок 6.7. Відповідь на це повідомлення

Створення об'єкта (create) створює новий об'єкт і позначається стрілкою зі стереотипом <<create>> (рис. 6.8).

Рисунок 6.8. Створення об'єкта

Знищення об'єкта (destroy) видаляє об'єкт. Об'єкт може знищити сам себе. Позначається стрілкою зі стереотипом <<destroy>>. При знищенні об'єкта на його лінії житті з'являється символ руйнування, який позначається хрестом (рис. 6.9).

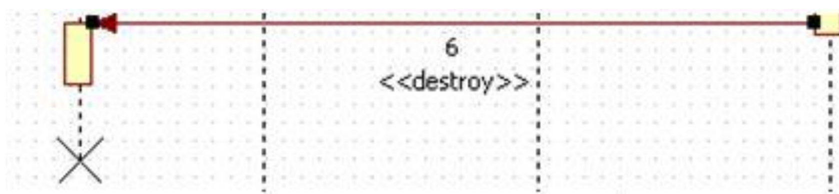


Рисунок 6.9. Знищення об'єкта

Для визначення типу повідомлення в StarUML потрібно виконати такі дії: виділити повідомлення, натиснувши відповідну стрілку один раз лівою кнопкою миші, відкрити редактор властивостей, клацнути на ньому розділ ActionKind і з розкривного списку вибрати потрібний тип синхронізації (рис. 6.10).

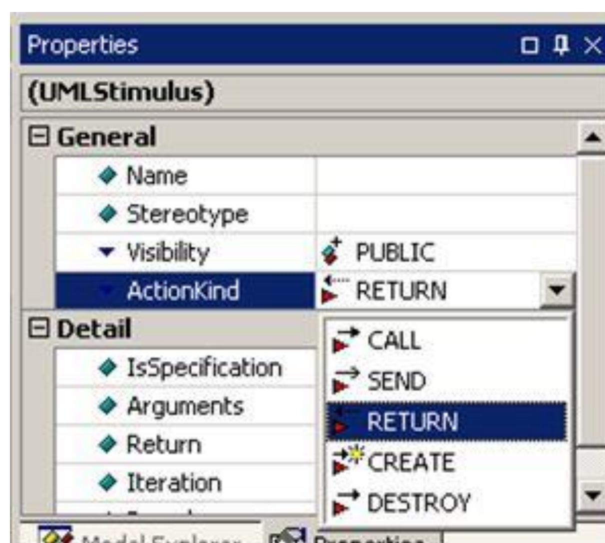


Рисунок 6.10. Вибір типу повідомлення

2.2 Додавання діаграми послідовності в модель

Для створення нової діаграми послідовності потрібно виконати такі кроки: клацнути правою кнопкою миші на папці подання Logical View у навігаторі моделі, з контекстного меню вибрати пункт Add Diagram, зі списку вибрати діаграму послідовності Sequence Diagram (рис. 6.11).

Також діаграму послідовності можна використовувати для деталізації прецеденту. Для цього потрібно створити діаграму з прецедентом. Для цього треба клацнути правою кнопкою миші по прецеденту, а не по папці *Logical View*. Однак, якщо будується діаграма послідовності для аналізу системи, то краще все-таки поміщати її в *Logical View*.

Зазвичай для основного потоку подій більшості прецедентів будується одна діаграма послідовності, для альтернативних потоків – додаткові діаграми, що описують всі інші сценарії. Так роблять тому, що на діаграмі послідовності

дій складно відобразити логіку ЯКЩО-ТО-ІНАКШЕ. Однак, якщо це необхідно і не захаращує діаграму, то це можна зробити за допомогою умов.

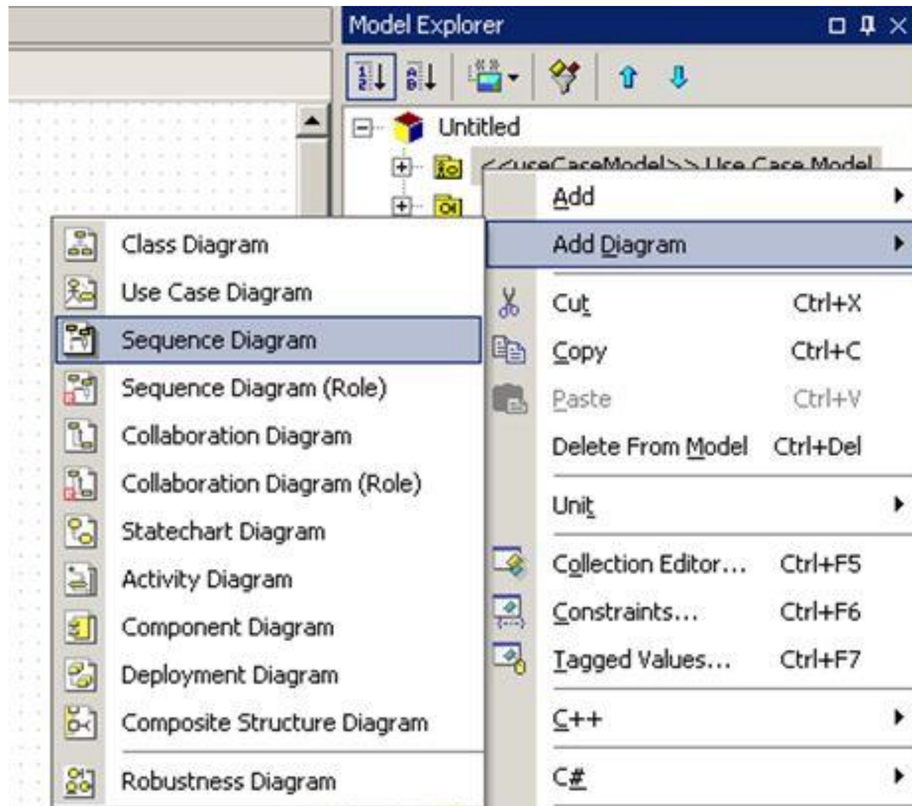


Рисунок 6.11. Додавання діаграми послідовності

2.3 Взаємозв'язок діаграм класів і послідовності

Процес побудови моделі системи є ітеративним. Особливо добре це можна бачити при створенні діаграм класів і послідовності. Яку діаграму створювати першою: класів або послідовності? Одні розробники починають з діаграм класів, інші – навпаки, з послідовності. І в тому і в іншому випадку, швидше за все, обидві ці діаграми, побудовані для одного сценарію, будуть надалі піддаватися змінам. Після побудови діаграм послідовності на діаграмах класів можуть з'явитися нові класи, а на діаграмах послідовності – нові об'єкти, яких раніше там не було, але вони прийдуть туди з діаграм класів. Можливо, що деякі об'єкти і класи будуть, навпаки, видалені.

3 Кооперативні діаграми

Діаграма кооперації – це альтернативний спосіб зображення сценарія варіантів використання. Цей тип діаграм загострює увагу на зв'язках між об'єктами, відображаючи обмін даними в системі. А діаграми послідовності відображають взаємодію об'єктів у часі, тому її слід читати зверху вниз і зліва направо.

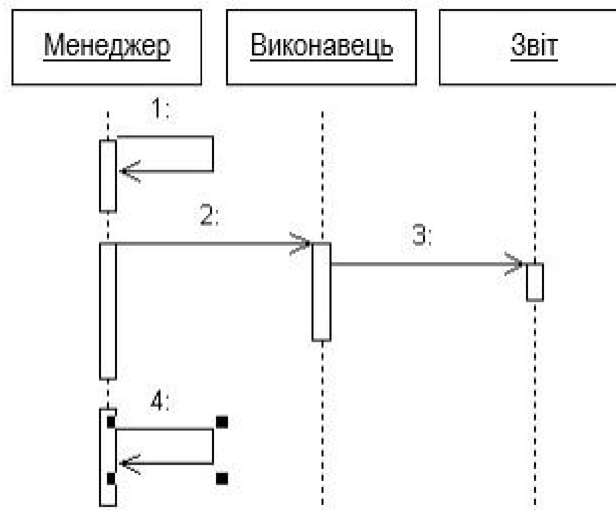
Діаграми кооперації містять всі ті самі елементи, що і діаграми послідовності: об'єкти, діючі особи, зв'язки між ними та повідомлення, якими вони обмінюються, але вони вже не впорядковані в часі.

Для того щоб додати діаграму кооперації подання Logical View, треба клацнути правою кнопкою миші по папці, що містить діаграму послідовності (якщо ви її не перейменовували, то вона носить ім'я *CollaborationInstanceSet1*), з контекстного меню вибрати пункт Add Diagram, після чого зі списку вибрати діаграму кооперації Collaboration diagram (рис. 6.12).

Рисунок 6.12. Додавання кооперативної діаграми

4 Послідовність розроблення і приклади діаграм послідовності

Діаграма послідовностей є UML-діаграмою взаємодії, що описує поведінкові аспекти системи, але розглядає взаємодію об'єктів в часі. Головними елементами діаграм послідовності є об’єкти, які є логічними сутностями, що подають окремі елементи системи та повідомлення, якими вони обмінюються. Діаграма послідовностей відображає взаємодію об'єктів в динаміці. На діаграмі послідовності об’єкт відображається у вигляді прямокутника на вершині пунктирної вертикальної лінії – лінії життя об’єкта. Іншими словами, діаграма послідовностей відображає часові особливості передачі і прийому повідомлень об'єктами. Вона являє собою фрагмент життєвого циклу об’єкту в процесі взаємодії. Кожне повідомлення подається у вигляді стрілки між лініями життя двох об’єктів. Повідомлення з’являються у тому порядку, як вони показані. Повідомлення, якими обмінюються об’єкти на діаграмі послідовності, пронумеровані, тобто мають чітку послідовність.



Розглянемо ще один приклад діаграми послідовності «Створити новий документ» (рис. 6.13). У верхній частині діаграми наведено перелік об'єктів (логічні сутності), які взаємодіють між собою у процесі виконання сценарію. Часова шкала на даній діаграмі направлена згори донизу. Повідомлення пронумеровані відповідно до черги їх пересилання між об'єктами.

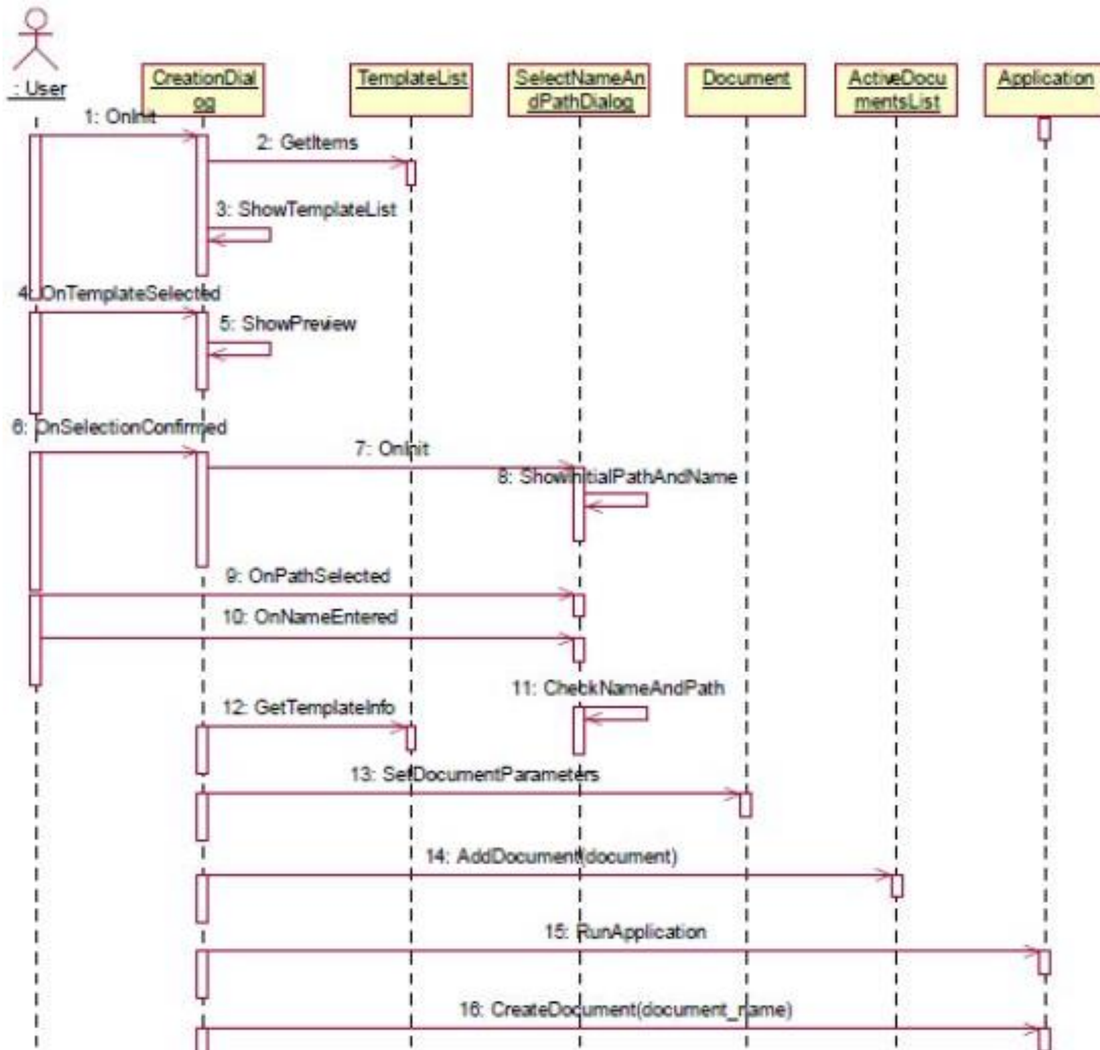


Рисунок 6.13. Діаграма послідовності для варіанта використання «Створити новий документ»

Наведемо короткий опис подій, що відбуваються при виконанні даного варіанта використання (рис. 6.13). При ініціалізації діалогу для створення нового документа (1) завантажується (2) та відображається (3) список шаблонів документів. При виборі користувачем одного з шаблонів (4) у діалозі відображається його початковий вигляд (5). Після того, як користувач підтверджує свій вибір остаточно, наприклад, скориставшись методом DoubleClick (6), система ініціалізує діалог (7), де просить ввести шлях до директорії та ім'я файла, у якому документ буде зберігатися. При цьому відображаються default-значення для шляху та імені файла (8). Користувач вводить шлях (9) та ім'я файла (10), система перевіряє введені дані (11). Система зчитує інформацію шаблона (12) і записує всі параметри до об'єкта, який представляє документ (13). Потім документ додається до списку активних документів (14) і відповідно до параметрів документа запускається прикладна програма, яка відповідає формату шаблона документа (15). При цьому в прикладній програмі створюється новий документ з ім'ям, яке ввів користувач (16) (див. рис. 6.13).

Додатковими елементами діаграми послідовності є лінії життя об'єктів (довгі вертикальні пунктирні лінії), які відображають час життя об'єкта від його створення до знищення; фокус керування (прямокутники на лініях життя об'єктів) відображає, який об'єкт виконує операції в певний момент часу; та символи знищення об'єктів (хрест на лінії життя об'єкта), що відображає процес знищення об'єкта (рис. 6.14). Наприклад, об'єкт «Authorization Record» на діаграмі створюється на 2-му кроці виконання сценарію, використовується протягом 4-6 кроків та знищується після шостого кроку.

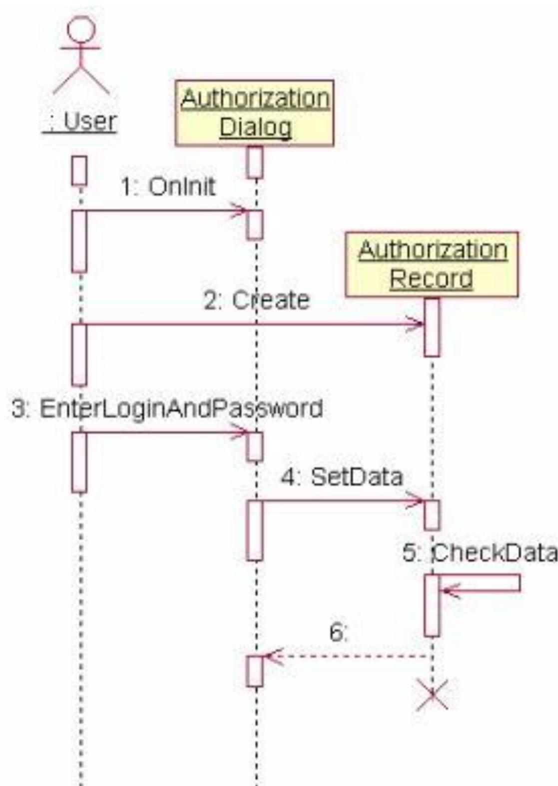


Рисунок 6.14. Фрагмент діаграми послідовності для процесу авторизації користувача

Основні типи повідомлень на діаграмі послідовності:

– пряме – повідомлення, яке об’єкт-ініціатор надсилає об’єкту-приймачу (див. 1-4 на рис. 6.14);
– рефлексивне, яке об’єкт надсилає сам собі (5);
– зворотнє, при якому управління повертається об’єкту ініціатору (6), часто повідомлення даного типу не мають назви.

Розглянемо ще один приклад діаграми послідовностей вибору семінару. Сенс діаграми (рис. 6.15) цілком зрозумілий: студент хоче записатися на якийсь семінар, пропонований у рамках деякого навчального курсу. З цією метою проводиться перевірка підготовленості студента, для чого запитується список (історія) семінарів курсу, вже пройдених студентом (перейти до наступного семінару можна, лише пропрацювавши матеріал попередніх занять – знайома картина, чи не так?). Після отримання історії семінарів об'єкт класу "Семінар" отримує статус підготовленості, на основі якої студенту повідомляється результат (статус) його спроби запису на семінар.

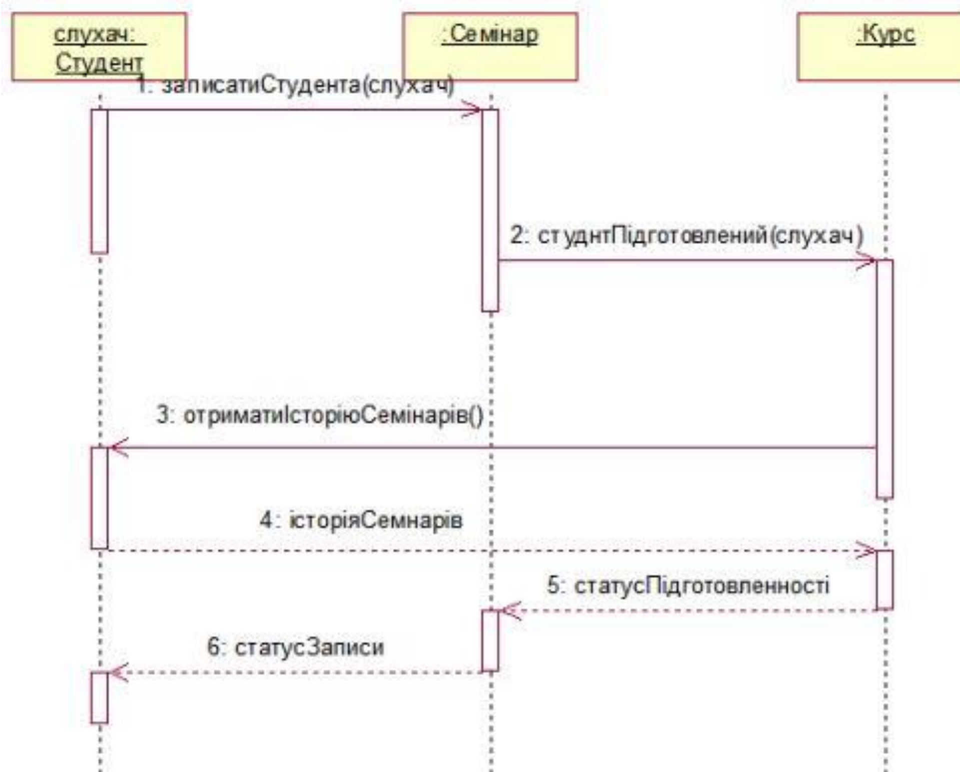


Рисунок 6.15. Діаграма послідовностей вибору семінару

Ще один приклад діаграми послідовностей подано на рис. 6.16, де змодельовано роботу деякої інтернет-сторінки на якій відбувається обчислення певних полів.

Рисунок 6.16. Діаграма послідовностей роботи сторінки з обчислювальним полем

Діаграма на рис. 6.17 показує, як приготовати яєчню.

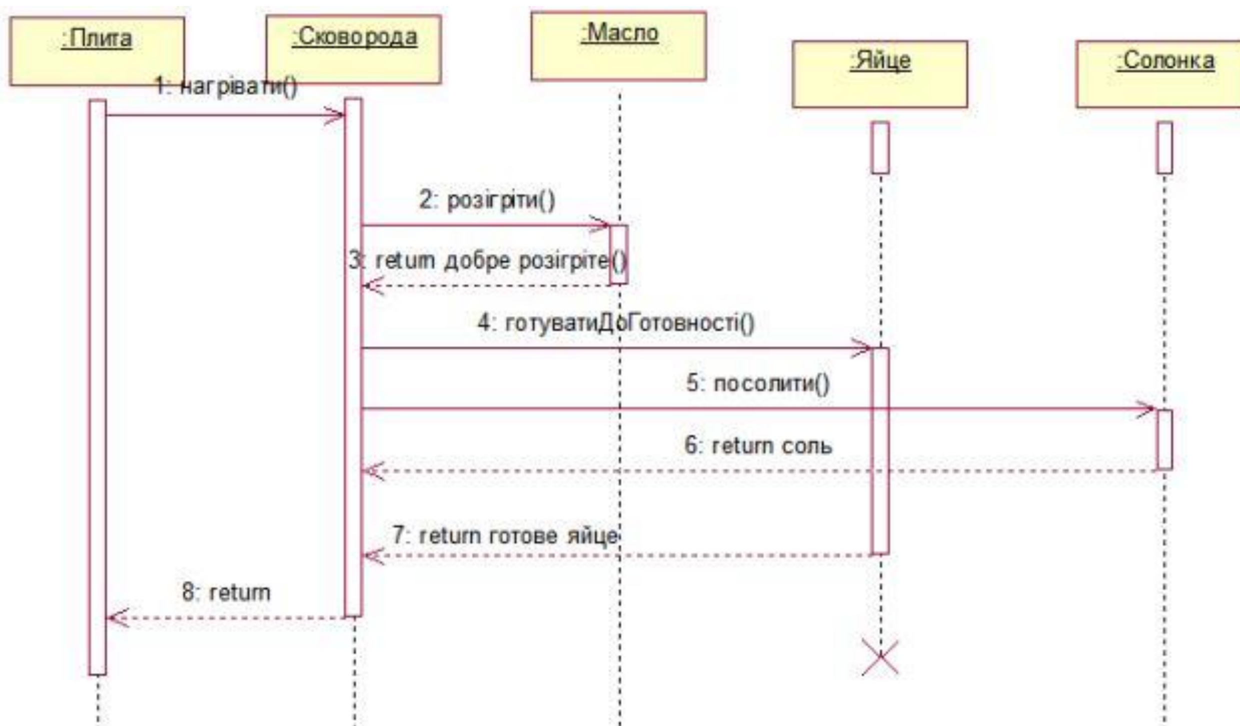


Рисунок 6.17. Діаграма послідовностей приготування яєчні

Коли ми дивимося на таку діаграму послідовності, одразу розуміємо:

– треба не забути увімкнути плиту перед смаженням;

– треба добре розігріти сковороду;

– першим треба покласти масло, а потім яйця, а не навпаки;

– сіль можна додати асинхронно в процесі готування.

Розберемо кожний елемент діаграми окремо:

1. *Об'єкт, Учасник (Object, Participant).*

Позначається прямокутником, в якому вказується інформація про учасника дії. Це, зазвичай, назва об'єкту та його клас, розділені двокрапкою.

2. *Лінія життя (Life Line).*

Лінія, яка йде від учасника, позначає відведений об'єкту час життя. Позначається пунктирною лінією.

3. *Активація, Фрагмент виконання (Activation Bar, Execution Occurances).*

Позначається вузьким прямокутником, який розташований на лінії життя. Вказує початок і завершення дії, в якій бере участь об'єкт.

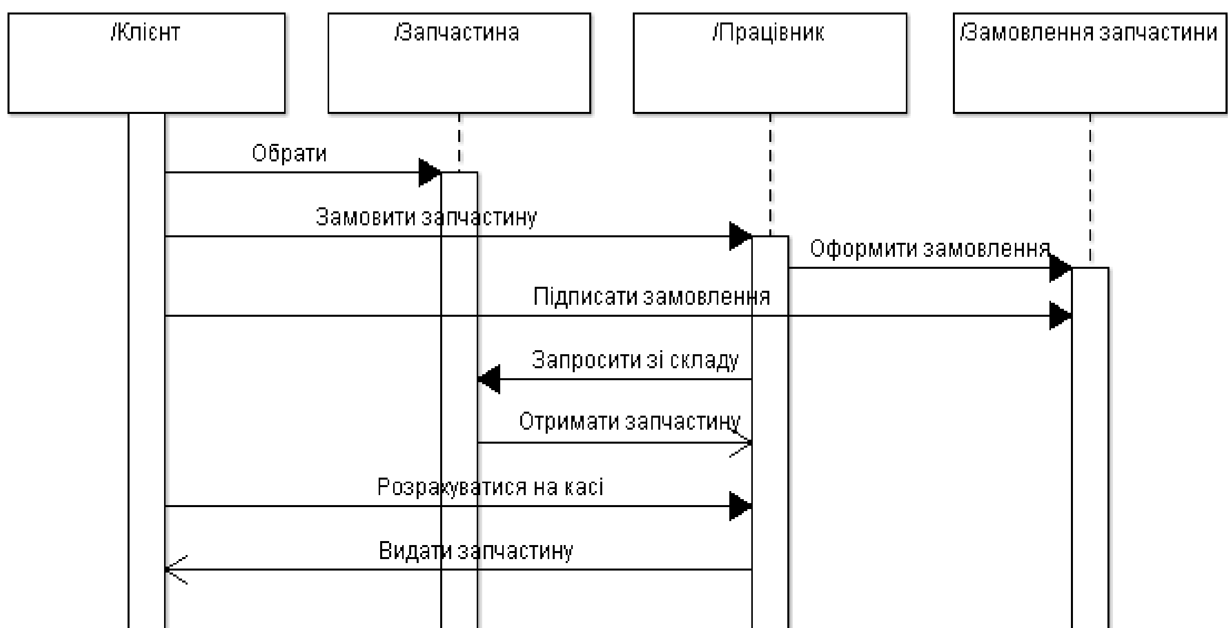
4. *Повідомлення, Стимул (Message, Stimulus)*

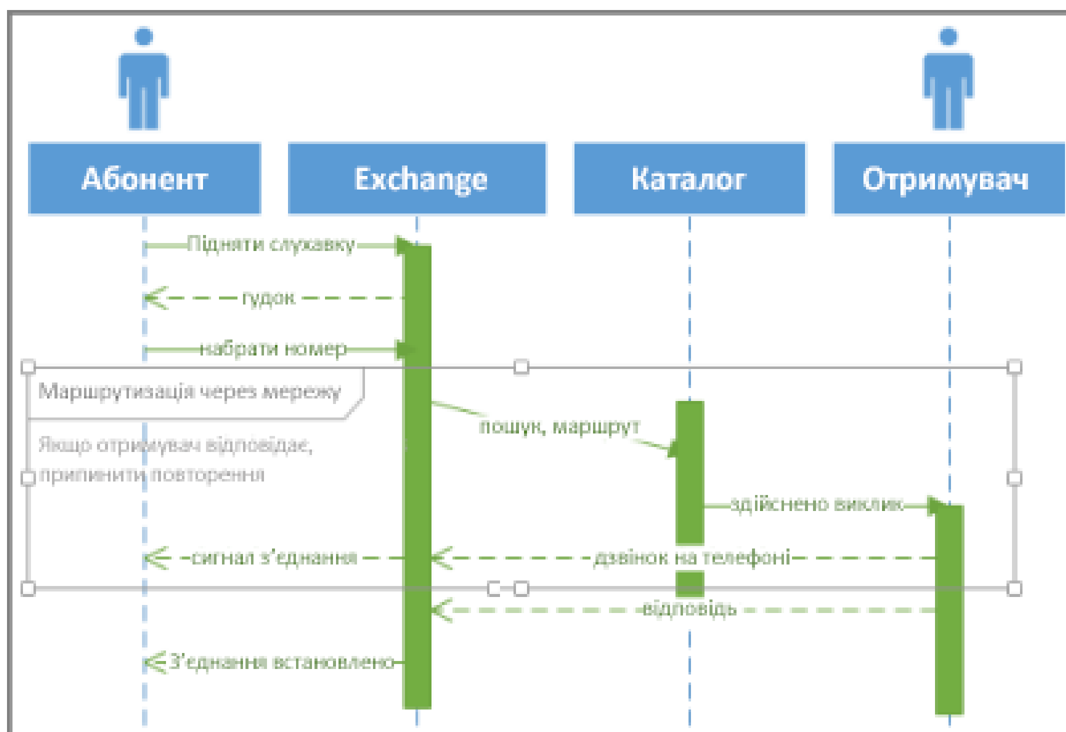
Стрілка від однієї лінії життя до іншої. Показує взаємодію об'єктів.

5. *Знищення об'єкта.*

Позначається діагональним хрестом на лінії життя. Позначає кінець життя об'єкта.

Далі наведено ще декілька прикладів діаграм послідовностей.





Контрольні запитання для самоконтролю

1. Для чого призначена діаграма послідовності (sequence diagram)?
2. Навести її основні елементи.
3. Дати визначення об'єктів діаграми послідовності.
4. Як вставити на діаграму послідовності актора?
5. Чи може бути в об'єкта декілька ліній життя?
6. Чи може діаграма послідовностей утримувати об'єкт з лінією життя, але без фокуса управління?
7. Дати визначення повідомлень діаграми послідовності, назвати основні типи повідомлень.
8. Чим асинхронне повідомлення відрізняється від синхронного?
9. Як створити відповідь на повідомлення?
10. Для чого використовуються прямі, зворотні та рефлексивні повідомлення?
11. Як на діаграмі послідовності використовуються актори?
12. Що таке лінія життя об'єкта? Чим визначається початковий і кінцевий момент періоду життя кожного об'єкта?

Лабораторне завдання

Лабораторне завдання

1. Дати відповіді на контрольні питання.

2. Створити діаграму послідовностей відповідно до індивідуального завдання (див. завдання до лабораторної роботи 5).

Для кожного варіанта використання на Usecase Diagram створити Sequence. На кожній діаграмі взаємодії має бути головний актор (за наявності) та не менше 5 об’єктів. Кожна діаграма взаємодії повинна містити не менше 10 повідомлень, якими обмінюються об’єкти в процесі виконання сценарію. Загальна сума різних об’єктів у проєкті має налічувати 12-15 об’єктів. Об’єкти та повідомлення на діаграмах повинні мати зрозумілі назви. При побудові діаграм використовувати прямі, рефлексивні та зворотні типи повідомлень, а також символи знищення об’єктів.

3. Оформити протокол лабораторної роботи.

Лабораторна робота 7
Створення діаграм діяльності (Activity Diagrams)

Мета роботи: навчитися створювати діаграми діяльності.

Теоретичні відомості

1 Основні елементи нотації діаграм діяльності (Activity Diagram)

Діаграми діяльності (діаграма активності) створюються на різних етапах життєвого циклу системи для моделювання послідовності бізнес-процесів або потоку дій, реалізованих методами класів.

Зазначені послідовності можуть являти собою альтернативні галузі процесу оброблення даних або галузі, які можуть виконуватися паралельно. Діаграми діяльності є аналогом блок-схеми будь-якого алгоритму. Вони, як і діаграми станів та переходів, зображуються у вигляді орієнтованого графу, вершинами якого є дії, а ребрами – переходи між діями.

Діаграми діяльності доцільно використовувати для аналізу:

– змісту сценаріїв застосування проєктованої системи;

– взаємодії потоків робіт різних сценаріїв;

– виконання сценаріїв у багатопроцесорних обчислювальних середовищах.

Ці діаграми широко використовуються при описі поведінки, що має значну кількість паралельних процесів. Кожний стан на діаграмі діяльності відповідає виконанню деякої елементарної операції, а перехід у наступний стан виконується тільки після завершення цієї операції. Тому діаграму діяльності можна вважати окремим випадком діаграми станів.

Основним напрямком використання діаграми діяльності є візуалізація особливостей реалізації операцій класів, коли необхідно надати алгоритми їх виконання.

Розглянемо основні елементи нотації діаграми діяльності. На ній ілюструються діяльності, переходи між ними, елементи вибору і синхронізації.

Діяльністю називається виконання певної поведінки в потоці керування системи. Діяльність (activity) є окремим випадком стану (state) без назви, який має одну вхідну подію (OnEntry action). В UML діяльність зображується у вигляді прямокутника з округленими кутами і текстовим описом усередині. Діяльність (певна дія) позначає деякий крок (етап) процесу, наприклад: «Розрахувати заробітну платню», «Перевірити результати запиту, «Перекласти слово» тощо. Так, у прецеденті Замовлення товарів онлайн одним з таких кроків може бути набирання номеру (рис. 7.1).

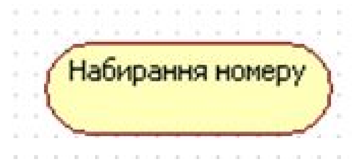


Рисунок 7.1. Діяльність

Перехід показує, як потік керування переходить від однієї діяльності до іншої. Зазвичай перехід здійснюється по завершенню певної діяльності (рис. 7.2).

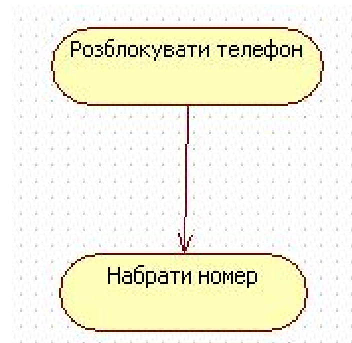


Рисунок 7.2. Перехід між діями

У наведеному прикладі (рис. 7.2) користувач, щоб зателефонувати може Розблокувати телефон та Набрати номер. Це дві різні діяльності, перехід до набору номера можливий тільки після розблокування телефону.

Події (events) на переходах діаграми діяльності не задаються, оскільки вважається, що перехід від однієї дії до іншої здійснюється безумовно.

Гранична умова (guard condition) використовується лише для визначення дії, до якої переходить керування у випадку неоднозначності). Тобто, якщо з даної вершини на діаграмі діяльності можна перейти до декількох інших вершин для всіх переходів, необхідно визначити граничну умову.

Характеристика дії (action) для переходу також немає сенсу, оскільки всі дії на цій діаграмі подані вершинами графу.

Для діаграми діяльності характерними є такі **спеціальні стани**:

1. *Початковий стан* – аналогічний до діаграми станів та переходів;

2. *Кінцевий стан* – аналогічний до діаграми станів та переходів;

3. *Стан прийняття рішення* – стан, в якому здійснюється прийняття рішення про перенаправлення потоку керування до одного зі станів, пов’язаних із даним станом.

4. *Стан синхронізації* – стан, в якому здійснюється розділення загального потоку керування на декілька гілок (чи навпаки, декілька гілок поєднуються в єдиний потік).

Два стани на діаграмі діяльності – початковий і кінцевий – визначають тривалість потоку. Початковий стан обов'язково має бути зазначено на діаграмі, він визначає початок потоку. Кінцевих станів може бути кілька або жодного, а початковий має бути лише один. Початковий стан зображується жирною крапкою, а кінцевий – жирною крапкою у колі (рис. 7.3). Саме кінцевий стан визначає точку завершення потоку. Саму діаграму діяльності прийнято розташовува-

ти так, щоб дії слідували згори донизу. При цьому початковий стан зображують у верхній частині діаграми, а кінцевий – внизу.

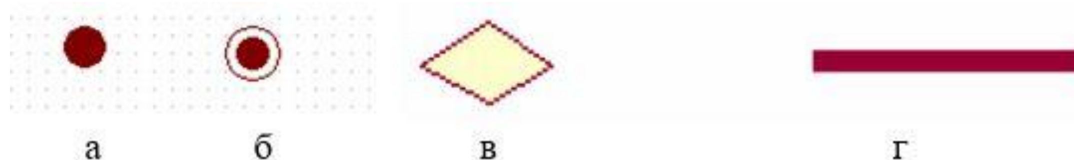


Рисунок 7.3. Позначення станів:
а) початковий; б) кінцевий; в) прийняття рішення; г) синхронізації

При моделюванні керуючих потоків системи часто буває потрібно показати місця їх поділу на основі умовного вибору. Вибір рішення на діаграмі показується ромбом, розміщеним на переході. Обмежувальні умови, від яких залежить вибір напрямку переходу, зазвичай розміщують над ромбом. У нотації UML умови записуються у квадратних дужках: [умова] (рис. 7.4).

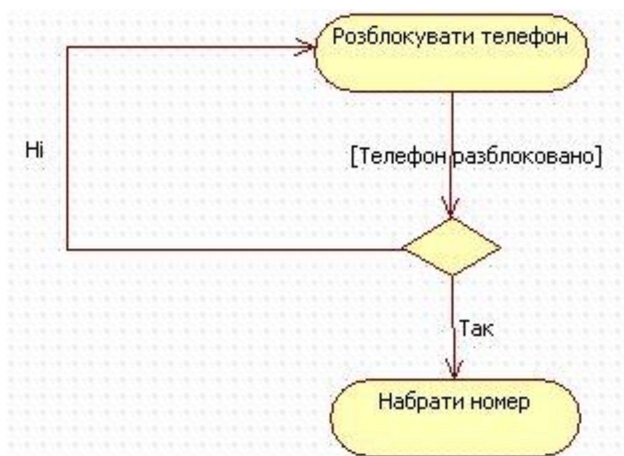


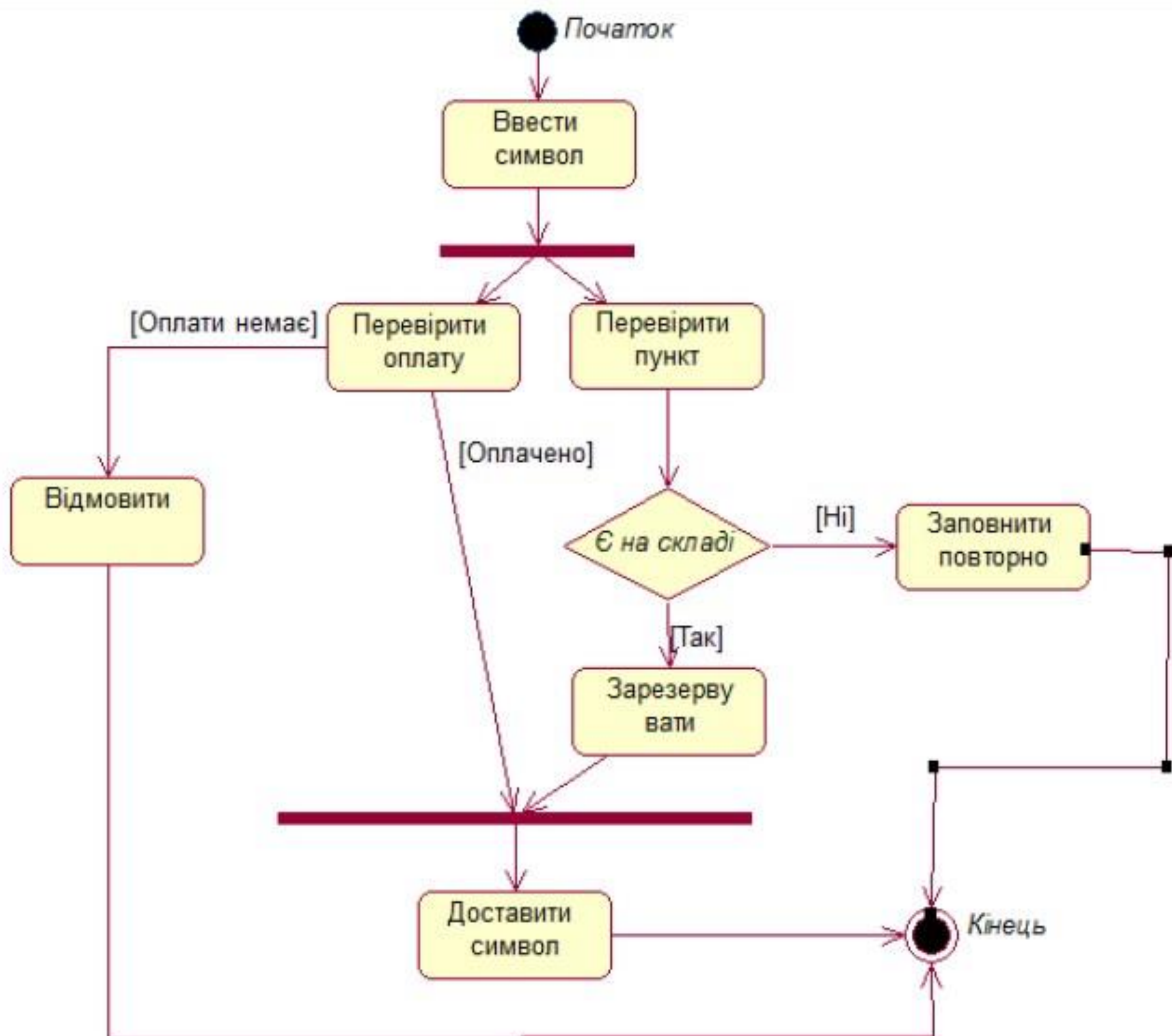
Рисунок 7.4. Умова переходу між діяльностями

Синхронізація – це спосіб показати, що дві або більше гілок потоку виконуються паралельно. Діяльності, поміщені між двома жирними лініями на діаграмі діяльності, виконуються синхронно (одночасно).

Наприклад, після оплати замовлення покупцем система присвоює замовленню унікальний номер та направляє підтвердження замовлення на електронну пошту покупця. Ці дві діяльності можна виконати синхронно, як це показано на діаграмі на рис. 7.5.

Рисунок 7.5. Лінії синхронізації

Переходи. Якщо зі стану дії виходить єдиний перехід, то він може бути ніяк не позначений. Якщо ж таких переходів декілька, то виконуватися може тільки один з них. У цьому разі для кожного з таких переходів має бути явно записана умова у прямих дужках. Умова істинності має виконуватися тільки для одного з них. Подібний випадок зустрічається тоді, коли послідовна діяльність повинна розділитися на альтернативні гілки, залежно від значення деякого проміжного результату. Така ситуація називається розгалуженням, а для її позначення використовується спеціальний символ.



Секції (доріжки, swimlane[15]) ділять діаграму діяльності на кілька ділянок. Це потрібно для того, щоб показати, хто відповідає за виконання діяльності і в якому порядку. Наприклад, якщо діяльність розміщена на секції з ім'ям Користувач, то цей актор і виконує її (рис. 7.6).

[15] Swimlane дослівно перекладається як доріжка плавального басейну (за аналогією з графічним відображенням). Як swimlanes на діаграмі можуть виступати фізичні особи, групи осіб, відділи підприємства чи навіть окремі організації.

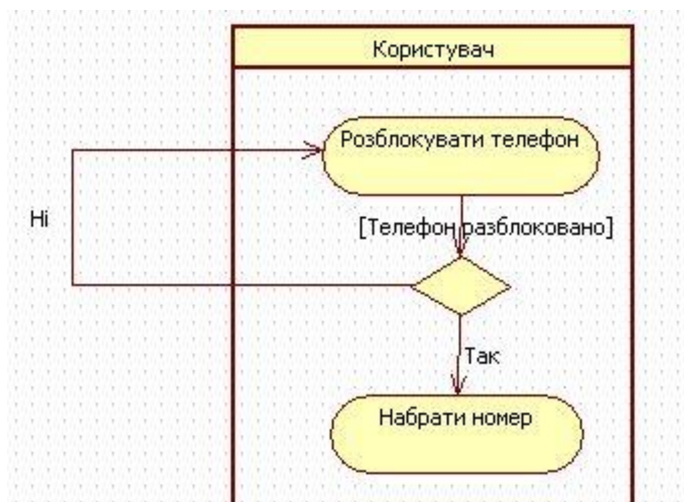


Рисунок 7.6. Секція

Використання секцій на діаграмі дозволяє при моделюванні бізнес-процесів асоціювати виконання кожної дії з конкретним підрозділом компанії. При цьому підрозділ несе відповідальність за реалізацією окремих дій, а сам бізнес-процес подається у вигляді переходів дій від одного підрозділу до іншого.

Назва підрозділів явно вказується у верхній частині доріжки. Перетинати лінію доріжки можуть тільки переходи, які позначають вхід або вихід потоку керування відносно певного підрозділу. Порядок проходження доріжок не несе жодної семантичної інформації і визначається міркуваннями зручності.

Об'єкти. Узагальнено дії на діаграмі діяльності виконуються над тими чи іншими об'єктами. Ці об'єкти або ініціюють виконання дій, або визначають деякий їхній результат. Дії специфікують виклики, які передаються від одного об'єкта графу діяльності до іншого.

Для графічного подання об'єктів використовується прямокутник класу, з тією відмінністю, що ім'я об'єкта підкреслюється. Далі після імені може вказуватися характеристика стану об'єкта у прямих дужках.

2 Створення діаграми діяльності StarUML

Щоб побудувати діаграму діяльності для певного прецеденту в StarUML, потрібно клацнути правою кнопкою миші по цьому прецеденту, у контекстному меню вибрати пункт Add Diagram, потім зі списку вибрати Activity Diagram (рис. 7.7).

Поле для створення діаграми діяльності з'явиться у вікні програми, зміниться панель інструментів ліворуч, і нова діаграма з'явиться у навігаторі моделі.

Як приклад побудуємо діаграму діяльності для додаткового прецеденту з оформлення замовлення актором Покупець керування (рис. 7.8). Оформлення замовлення включає вказівку своїх особистих контактних даних, електронної пошти і оплату замовлення. Оформлення розпочинається з кошика покупця, коли він вибирає опцію «Оформити замовлення».

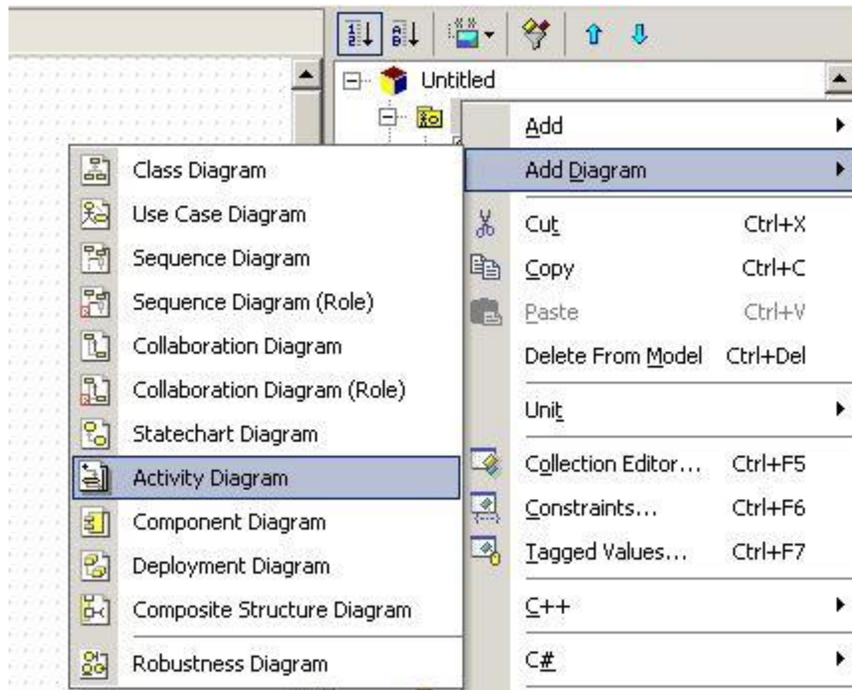


Рисунок 7.7. Додавання діаграми діяльності

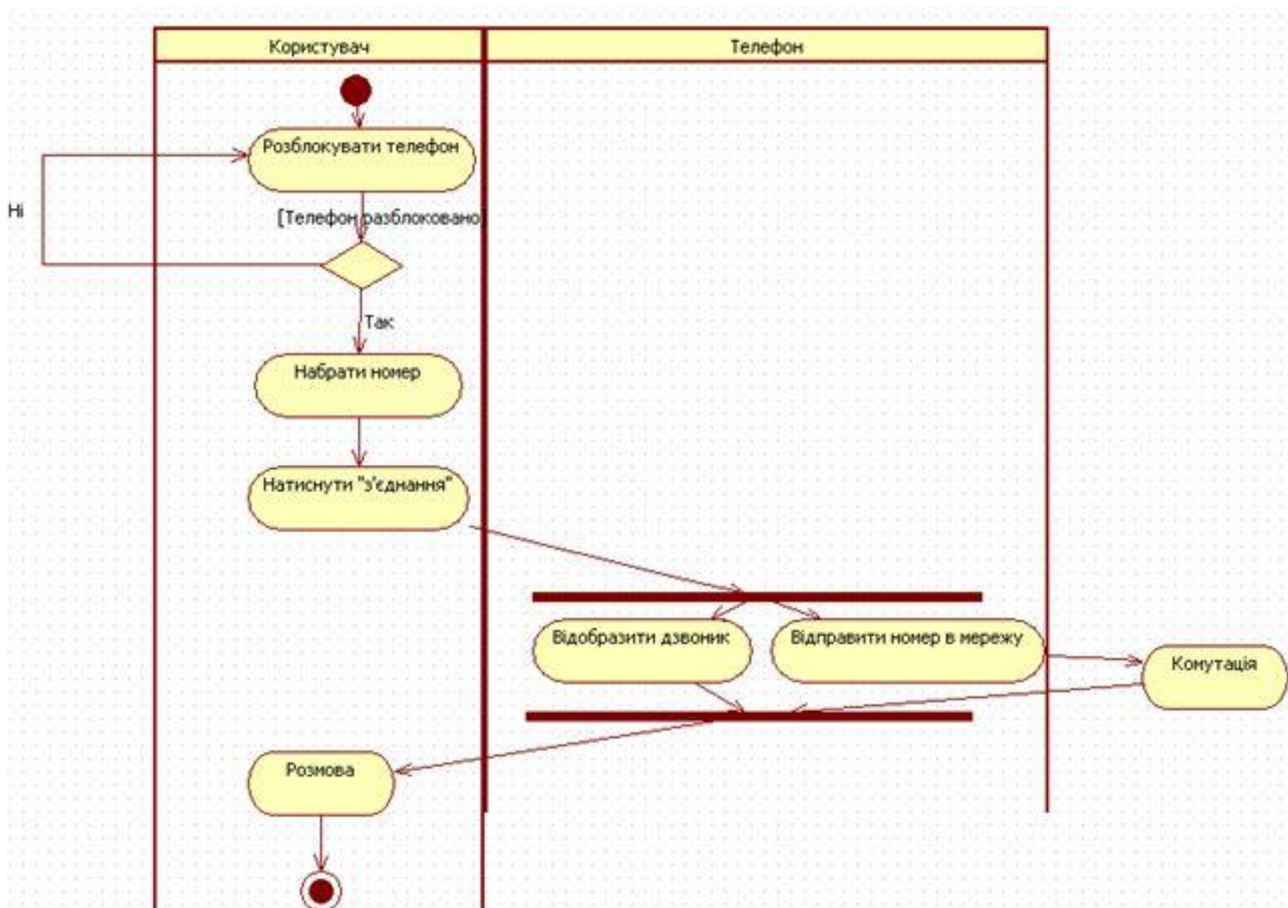


Рисунок 7.8. Діаграма діяльності прецеденту Оформити замовлення

Наведемо приклад розробки та створення діаграми діяльності «Перекласти слово» для електронного словника (застосування спеціальних станів). На рис. 7.9 показано повний вигляд діаграми діяльності. На початковому етапі перекла-

ду обирається словник, користувач вводить слово та система робить його переклад, далі з використанням стану прийняття рішення визначається чи обрані додаткові опції перекладу. У разі, якщо опцій не вибрано, система переходить до показу результату перекладу. В іншому випадку, потік керування розподіляється на дві гілки, кожна з яких виконує певну дію (отримати транскрипцію слова та список синонімів відповідно). Після закінчення виконання обох операцій дві гілки поєднуються в єдиний потік і здійснюється показ результату. Потім визначається чи потрібен друк отриманої інформації, і у разі потреби вона друкується.

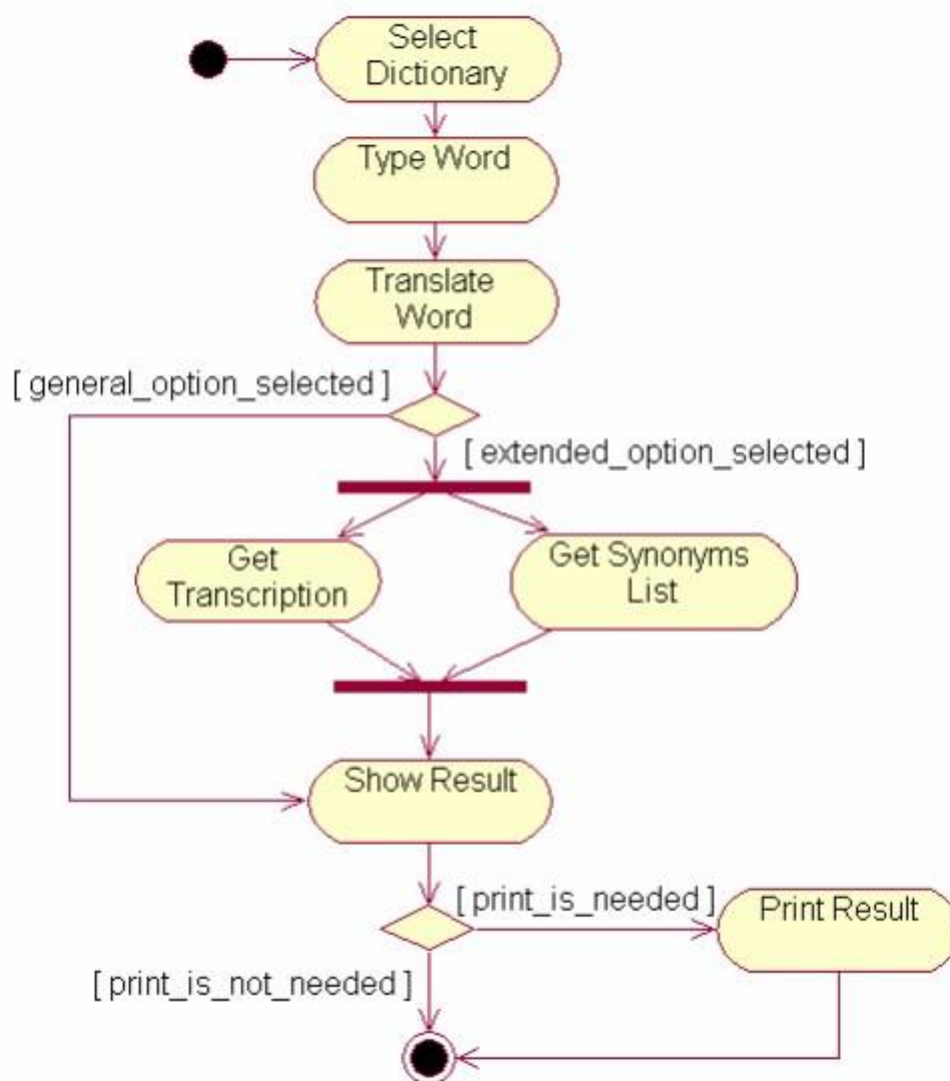


Рисунок 7.9. Діаграма діяльності для перекладу слова електронним словником

Розглянемо ще один приклад застосування діаграми діяльності для відображення послідовності дій при моделюванні бізнес-процесів (рис. 7.10). На ній використовуються доріжки для спрощеного варіанта бізнес-процесу «Розробка програмного забезпечення».

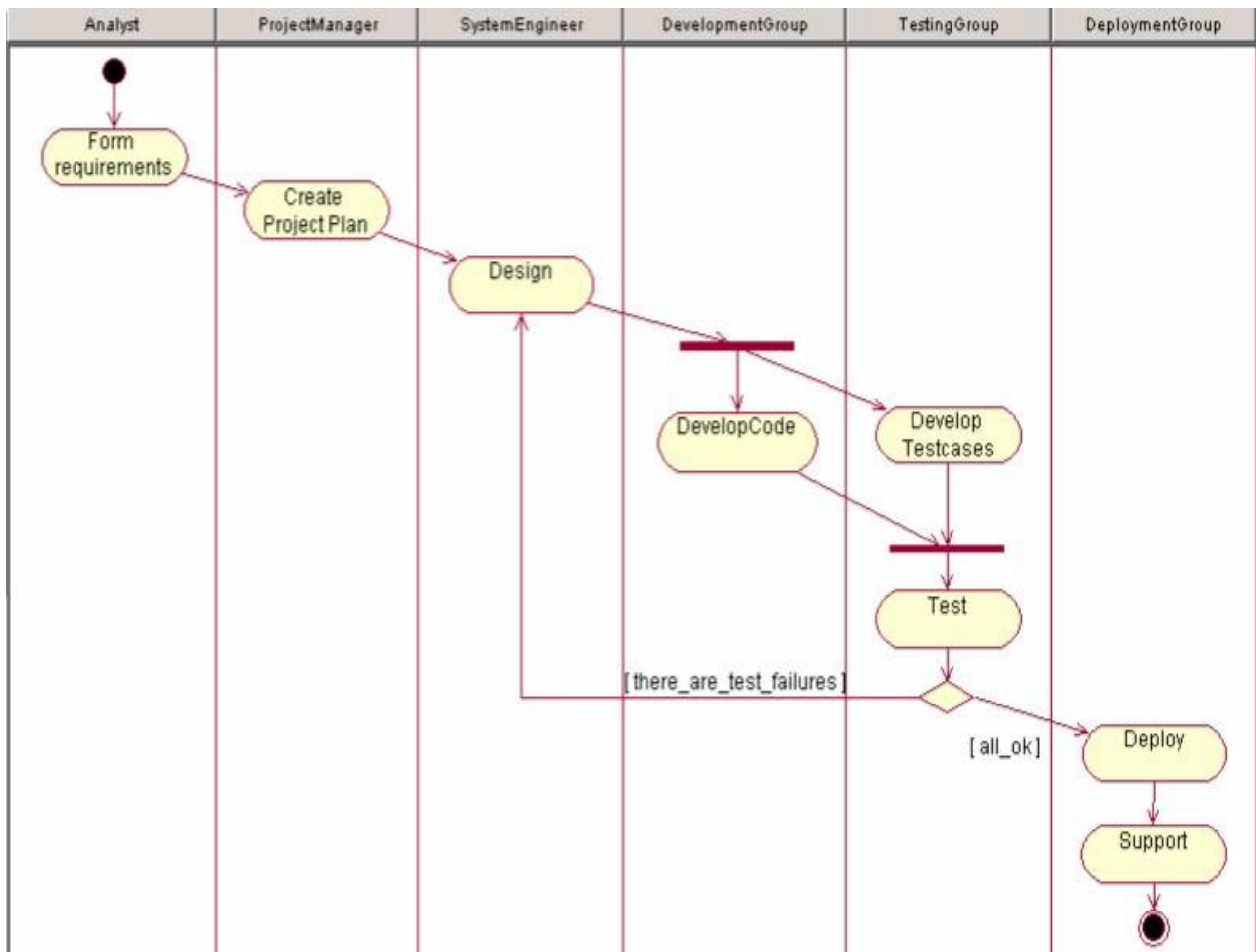


Рисунок 7.10. Використання діаграми діяльності для відображення бізнес-процесів

В окремих доріжках цієї діаграми розміщена діяльність таких осіб (груп осіб):

1) аналітик, який розробляє вимоги до проєкту;
2) керівник проєкту, який складає план виконання робіт;
3) системний інженер, який проєктує систему;
4) група розробників, які створюють програмний код;
5) група тестувальників, які формують варіанти тестування та тестують створену систему;
6) група впровадження, яка поставляє систему кінцевому користувачу та здійснює підтримку.

На рис. 7.11 наведена діаграма діяльності, яка відображає найпростіший процес прийняття на роботу. Наведений процес містить чотири діяльності:

– Interview – збір інформації;
– Analysis – аналіз зібраної інформації і прийняття рішення;
– Fill Forms – заповнення документів;
– Refuse – відмова у наймані.

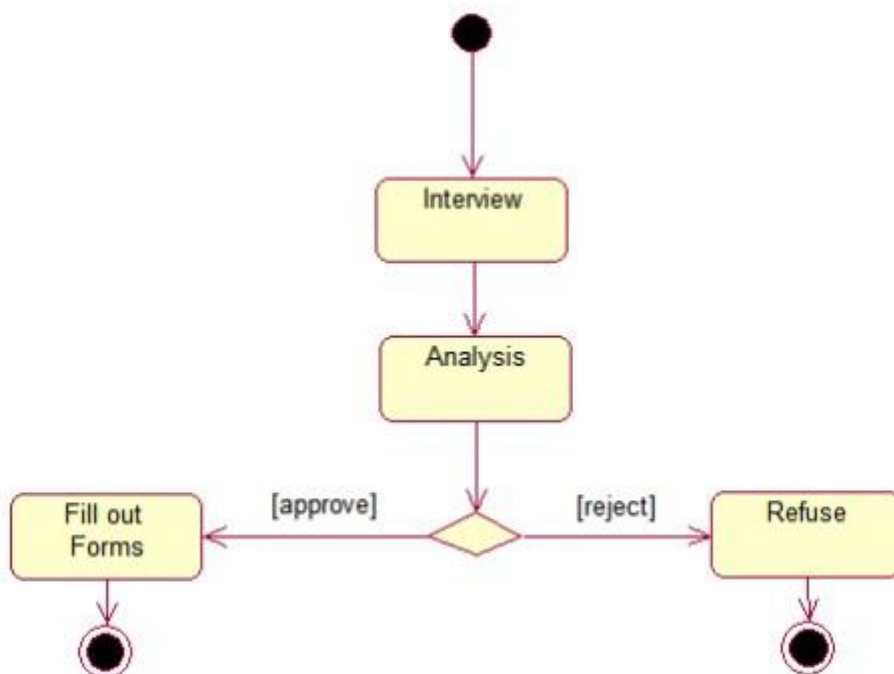


Рисунок 7.11. Діаграма діяльності, яка відображає найпростіший процес прийняття на роботу

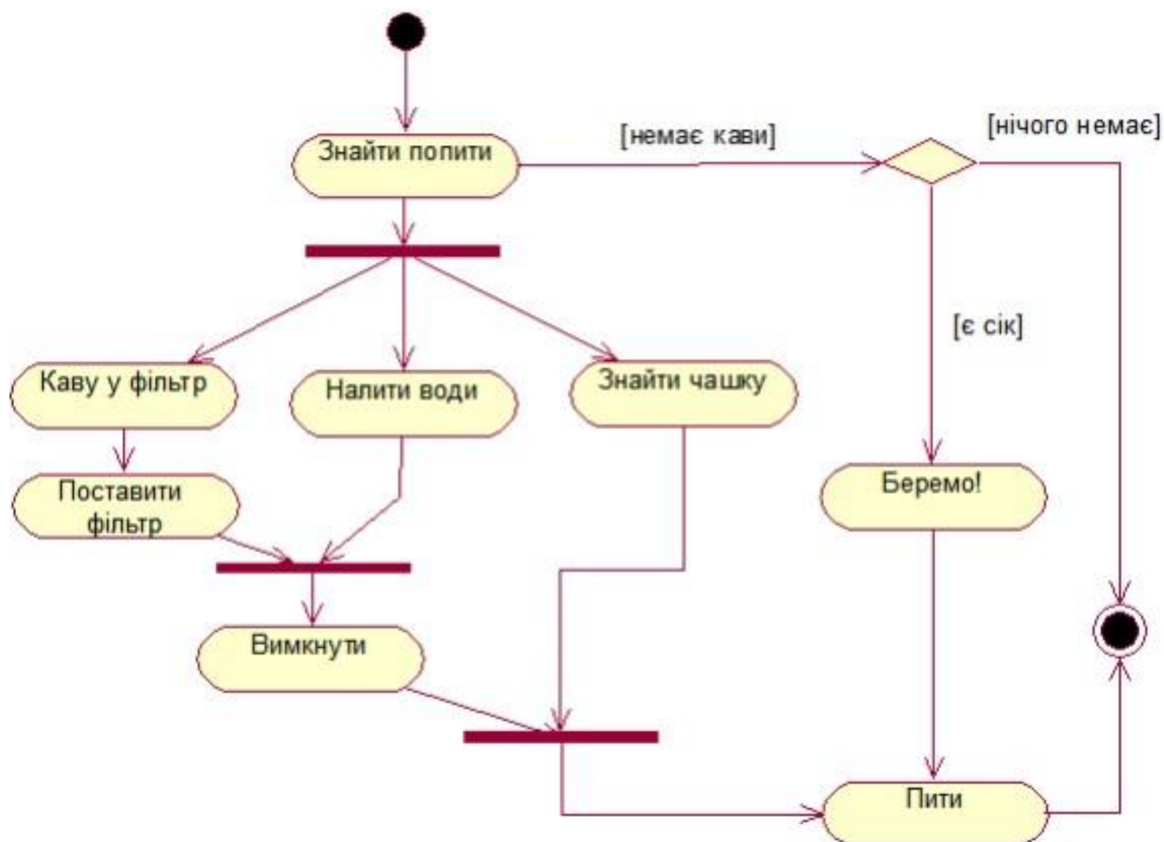


Рисунок 7.12. Діаграма діяльності, переходи

Контрольні запитання для самоконтролю

1. Чим діаграми діяльності відрізняються від блок-схем? Які переваги це обіцяє розробникам?
2. У яких випадках використовується діаграма діяльності?
3. Чим кінцевий стан потоку відрізняється від кінцевого стану діяльності?
4. Як позначаються стани дій на діаграмі діяльності?
5. Для чого використовуються стани дій на діаграмі діяльності?
6. Як позначаються переходи на діаграмі діяльності?
7. Для чого використовуються переходи на діаграмі діяльності?
8. Для чого використовуються доріжки на діаграмі діяльності?

Лабораторне завдання

1. Дати відповіді на контрольні питання.

2. Створити діаграми діяльності відповідно до індивідуальних завдань лабораторної роботи 5. Кожна діаграма повинна містити не менше 6 діяльностей. При побудові кожної діаграми використовувати стани прийняття рішення та синхронізації.

3. Оформити протокол лабораторної роботи.

Лабораторна робота 8
Створення діаграми класів (Class Diagrams)

Мета роботи: навчитися створювати діаграми класів.

Теоретичні відомості

1 Основні елементи діаграм класів

Діаграма класів (Class diagram) подає внутрішню логічну структуру моделі системи в термінології класів об'єктно-орієнтованого програмування.

Для кожної системи будується не одна, а кілька діаграм класів: можливо, що для кожного прецеденту або сценарію своя. На одних показують підмножини класів, об'єднані в пакети, і стосунки між ними, на інших – відображають ті ж підмножини, але з атрибутами та операціями класів. Для подання системи розробляється стільки діаграм класів, скільки буде потрібно.

Дамо деякі визначення і опишемо основні елементи нотації діаграм класів.

Об'єкт – це деяка сутність реального світу або концептуальна (абстрактна) сутність. Клас у мові UML використовується для позначення множини об'єктів, які мають однакову структуру, поведінку і стосунків із об'єктами інших класів. Графічно клас зображується у вигляді прямокутника, який додатково може бути розділений горизонтальними лініями на розділи або секції. У цих розділах можуть зазначатися ім'я класу, атрибути (змінні) та операції (методи).

Наприклад, об'єктами можуть бути будинок №10 по вулиці Садовій, студент групи Тарас Петров, ваш комп'ютер або щось абстрактне – іспит з математики, торгове замовлення номер 1234, ваш банківський рахунок тощо.

Об'єкт має чітко визначені межі та значення для системи і характеризується станом, поведінкою та індивідуальністю.

Стан об'єкта зазвичай змінюється з часом і характеризується набором властивостей, які називаються атрибутами. Наприклад, покупець визначається ім'ям, адресою, телефоном, датою народження.

Поведінка визначає, як об'єкт реагує на запити інших об'єктів, і що може робити сам об'єкт. Поведінка характеризується операціями об'єкта. Наприклад, покупець може додавати товари у кошик, переглядати каталог, видаляти товари з кошика.

Індивідуальність означає, що кожен об'єкт унікальний, навіть якщо його стан ідентичний стану іншого об'єкта. Наприклад, об'єкти Василь Іванов і Ганна Петренко унікальні, хоча кожен з них є покупцем магазину і має однакові поведінку і стан.

Звичайно в системі існують численні об’єкти, які мають однакову поведінку, набувають однаковий стан. Наприклад, працівники фірми, яких може бути декілька десятків, та дані про яких містяться в базі даних, мають однакові атрибути (прізвище, ім’я, по батькові, дату народження, посада тощо) з різними значеннями цих атрибутів, а також можуть мати схожу поведінку – подати заяву на відпустку або переведення в інший підрозділ. Для групування об’єктів використовуються класи.

Клас (class) – це опис групи об’єктів із загальними властивостями (атрибутами), поведінкою (операціями), стосунками з іншими об’єктами і семантикою. Кожен клас є шаблоном для створення об’єкта. А кожен об’єкт – це екземпляр класу. Важливо пам’ятати, що кожен об’єкт може бути екземпляром тільки одного класу!

У нотації UML класи і об’єкти зображують у вигляді прямокутників (рис. 8.1). Прямокутник класу завжди ділиться на три секції (розділи), ім’я класу записують у першу секцію, кожне слово в назві класу прийнято писати з великої літери. У другій і третій секціях можуть зазначатися атрибути і операції (методи) класу відповідно, ці секції можуть бути порожніми.

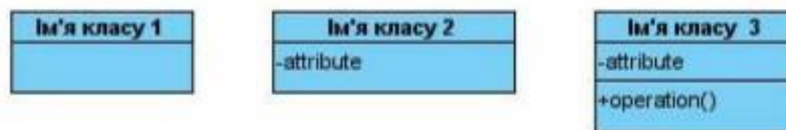


Рисунок 8.1. Варіанти графічного зображення класу на діаграмі класів

Назви класів вибираються відповідно до предметної області. Це має бути іменник або словосполучення в однині, що найбільш точно характеризує предмет. Клас повинен описувати тільки одну сутність. Назва об’єкта підкреслюється.

Ім’я класу може бути простим або складеним. Складене ім’я класу складається з імені класу і з імені пакета, якому належить клас, розділених двокрапкою. Ім’я класу має бути унікальним у межах пакета.

Складове ім’я об’єкта також складається з імені об’єкта та імені класу, розділених двокрапкою. Об’єкт може бути анонімним, якщо невідоме його справжнє ім’я. Тоді на діаграмі об’єкт зображується з ім’ям, яке складається з двокрапки та імені класу, якому належить об’єкт. Якщо об’єкт є екземпляром класу поки ще невідомого, то зображується ім’я об’єкта після якого йде двокрапка. Такий об’єкт називається «сиротою» (рис. 8.2).

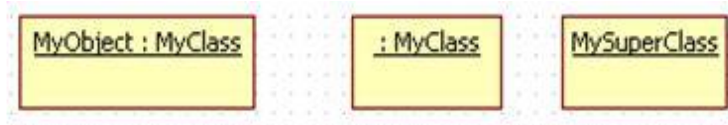


Рисунок 8.2. Найменування об’єктів

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

<символ видимості><ім’я атрибута>[кратність]: <тип атрибута> = <початкове значення>{рядок властивості}

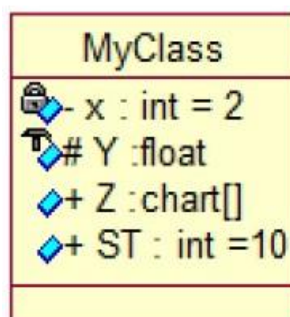
Відповідно до нотацій UML перед атрибутами та методами може зазначатись один з таких символів видимості:

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

«#» – атрибут із захищеною зоною видимості (protected), він недоступний для інших класів, за винятком операцій самого класу та нащадків (підкласів) даного класу;

«-» – закритий атрибут (private), недоступний для всіх класів без винятку;

«~» – пакетний (package), недоступний для класів за межами пакета, в якому визначений клас-власник даного атрибута.



Методи класу записуються в третій зверху секції прямокутника. Кожному методу класу відповідає окремий рядок, який складається з символу видимості операції, імені операції, типу значення, що повертається операцією і, можливо, рядка властивості операції:

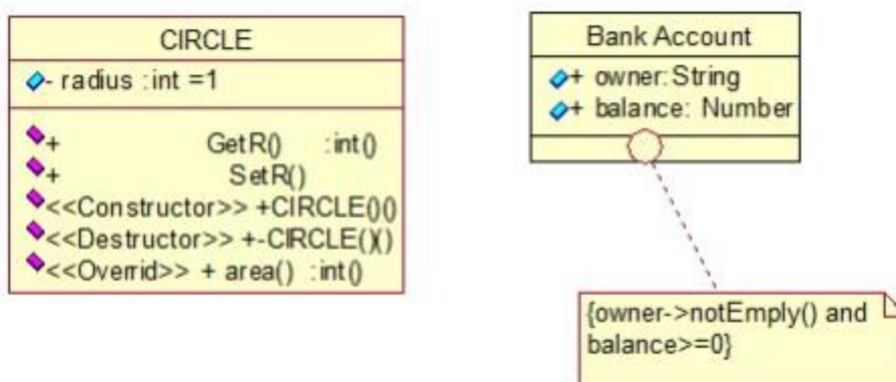


Рисунок 8.3. Методи та примітки класу

2 Виявлення класів

Виявлення класів можна почати з вивчення потоку подій. Іменники в описі цього потоку дадуть зрозуміти, що може бути класом. У загальному випадку іменник може виявитися дійовою особою, класом, атрибутом класу або виразом, що не є ні дійовою особою, ні класом, ні атрибутом класу.

Якщо в ході проєктування системи Ви вже побудували діаграми взаємодії, перед тим, як приступати до побудови діаграм класів, то шукайте на цих діаграмах схожі об'єкти.

Деякі можливі класи будуть виявлені при розгляді трьох стереотипів: сутність (entity), межа (boundary) і контроль (control). Ми вже зустрічалися зі стереотипами стосунків, коли говорили про стосунки на діаграмах прецедентів. Той самий принцип створення нового типу на базі вже існуючого застосуємо і для класів.

Стереотип – це механізм, який дозволяє класифікувати класи. Він використовується для створення нового типу елемента, в даному випадку нового типу класу.

Стереотипи допомагають краще зрозуміти відповідальності кожного класу моделі, категорізувати виконувані ними функції. В UML для цього застосовують три основні стандартні види стереотипів класів: класи-сутності, граничні класи і керуючі класи.

Клас-сутність (entity class) містить інформацію, що зберігається постійно і не повинна знищуватись зі знищенням об'єктів даного класу або припиненням роботи модельованої системи. Часто будучи абстракціями предметної області, вони мають найбільше значення для користувача, тому в їх назвах застосовуються терміни предметної області. Якщо існує проєкт бази даних, то можна звернутися до вивчення назв таблиць, багато з них стануть класами-сутностями. Позначаються класи-сутності стереотипом <<entity>> (рис. 8.4).

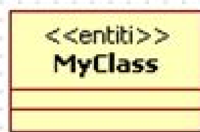


Рисунок 8.4. Позначення класів-сутностей

Граничними класами називаються класи, розташовані на межі системи з усім іншим оточенням, і тим самим вони забезпечують взаємодію між навколишнім середовищем і внутрішніми елементами системи.

Для визначення граничних класів необхідно дослідити діаграми варіантів використання. Для кожної взаємодії між актором і прецедентом потрібно створити хоча б один граничний клас. Якщо дві діючі особи ініціюють один прецедент, то вони можуть застосовувати один спільний граничний клас для взаємодії з системою. Позначаються граничні класи ім'ям стереотипу <<boundary>> (рис. 8.5).

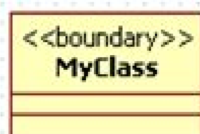


Рисунок 8.5. Позначення граничних класів

Керуючий клас відповідає за координацію дій інших класів. Вони служать для моделювання послідовної поведінки одного або декількох прецедентів і ко-

ординації подій, що реалізують закладену в них поведінку. Позначаються керуючі класи ім’ям стереотипу <<control>> (рис. 8.6).

Рисунок 8.6. Позначення керуючих класів

Керуючі класи можна уявити, як «виконуючі» прецеденти, тому у кожного варіанта використання зазвичай є один керуючий клас, який контролює послідовність подій цього прецеденту. Вони зазвичай залежать від застосунка. Керуючий клас делегує відповідальності інших класів. Сам він може отримувати мало повідомлень, але відсилати безліч. Його називають класом-менеджером. Він запускає альтернативні потоки і знає, як вчинити у випадку помилки. На початковому етапі проєктування керуючі класи створюються для кожної пари актор-прецедент, надалі вони можуть об’єднуватися, розділятися або виключатися.

3 Документування класів

Після того, як клас створений, інформацію про нього необхідно документувати. Зауважимо, що документація призначена для опису призначення класу, а не його структури.

Наприклад, якщо в моделі є клас Студент, то гарним описом для нього буде: «Студент – це людина, що навчається у внз. Клас містить інформацію, необхідну для виконання вишем своїх обов’язків щодо студента (проведення занять, проведення атестацій, нарахування стипендії тощо).»

Поганим описом буде опис структури класу, яка може бути і так описана за допомогою атрибутів. Наприклад, поганий опис класу Студент: «Ім’я, телефон, адреса, група, середній бал».

У StarUML документування класів виконується так, як і документування прецедентів. Потрібно виділити клас, який ви хочете описати, відкрити вікно документування Documentation на інспекторі моделі і ввести опис класу.

4 Стосунки між класами

Крім внутрішньої будови або структури класів на відповідній діаграмі вказуються стосунки між класами. При цьому сукупність типів таких стосунків фіксована в мові UML і зумовлена семантикою цих типів стосунків.

Як вже зазначалося, базовими стосунками в мові UML є:

– залежності;

– асоціації;

– узагальнення.

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

Існують ключові слова, які позначають деякі спеціальні види стосунків залежності:

– «access» – для позначення доступності відкритих атрибутів і операцій класу-джерела для класів-клієнтів;

– «bind» – клас-клієнт може використовувати деякий шаблон для своєї подальшої параметризації;

– «derive» – атрибути класу-клієнта можуть бути обчислені по атрибутах класу-джерела;

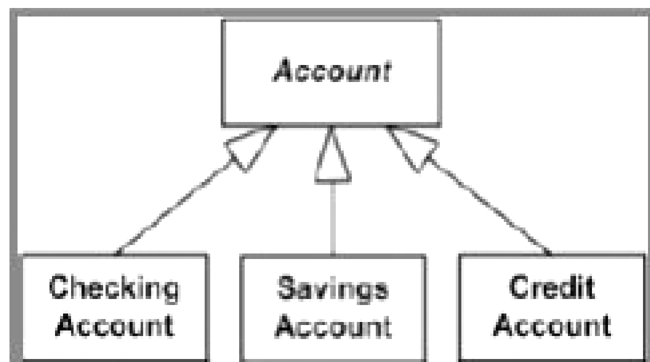
– «import» – відкриті атрибути й операції класу-джерела стають частиною класу-клієнта, так, ніби вони були оголошені безпосередньо в ньому;

– «refine» – вказує, що клас-клієнт є уточненням класу-джерела в силу причин історичного характеру, коли з’являється додаткова інформація в ході роботи над проєктом.

Стосунок асоціації відповідає наявності деяких стосунків між класами. Даний стосунок позначається суцільною лінією з додатковими спеціальним текстовим описом, який характеризує окремі властивості конкретної асоціації. В якості опису може використовуватись ім’я, асоціації, а також імена і кратність класів-ролей асоціації.

Стосунок узагальнення є звичайним таксономічним стосунком між більш загальним елементом (предком) і більш конкретним або спеціальним елементом (нащадком). Даний стосунок може використовуватися для подання взаємозв’язків між пакетами, класами, варіантами використання та іншими елементами мови UML.

На діаграмах стосунок узагальнення позначається суцільною лінією з трикутною стрілкою на одному з кінців. Стрілка вказує на загальний клас (клас-предок або супер-клас), а її відсутність – на спеціальний клас (клас-нащадок або підклас).



5 Побудова діаграми класів в StarUML

Діаграми класів є логічним поданням системи Logical View. На діаграмі Main подання Logical View зазвичай розміщують головну діаграму пакетів, а діаграми класів розміщують на інших аркушах цього подання. Для створення нової діаграми класів треба виконати такі кроки: клацнути правою кнопкою миші на папці подання Logical View у навігаторі моделі, у контекстному меню вибрати пункт Add Diagram, зі списку вибрати діаграму класів Class Diagram (рис. 8.7).

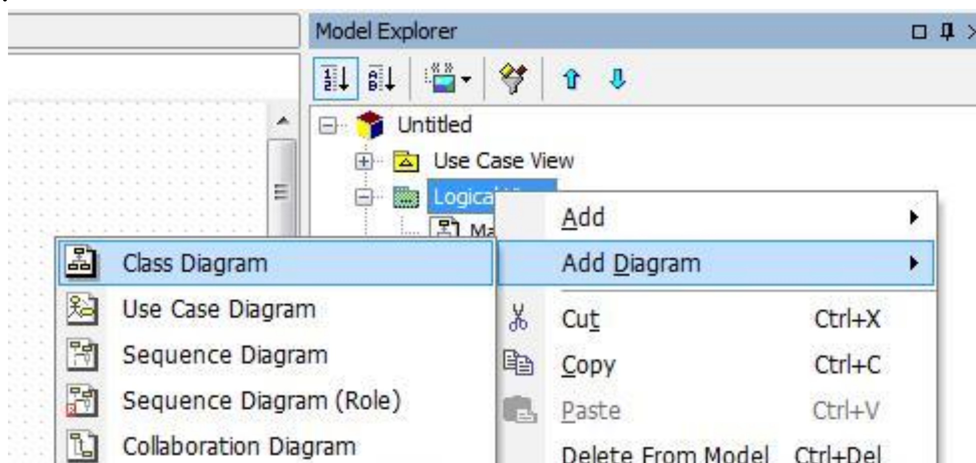


Рисунок 8.7. Додавання діаграми класів

Буде створена нова діаграма класів зі стандартним ім’ям ClassDiagram1, яке можна змінити в редакторі властивостей діаграми (рис. 8.8).

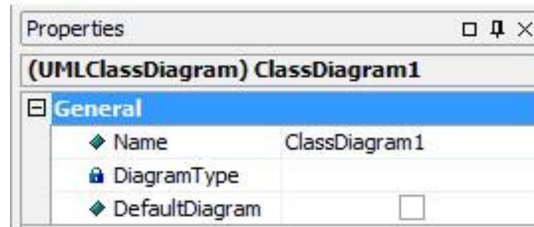


Рисунок 8.8. Редактор властивостей діаграми класів: змінення імені діаграми

Приклади діаграм класів

Для наочності вище наведеного матеріалу розглянемо діаграму класів, яка ілюструє відому казку «Курка Ряба». У казці Дід (клас Дід) та Баба (клас Баба) є людьми (класи Дід та Баба наслідують клас Людина) і володіють (стосунок асоціації) Куркою (клас Курка) на ім'я Ряба. З іншого боку, Курка Ряба є птахом (є нащадком класу Тварина) і може нести яйця (клас Яйце, стосунок «відкладати»). Зауважимо, що яйце у казці фігурує як золоте, так і просте, що показано на діаграмі за допомогою класу <<enumeration>> Вид. Ще одною дійовою особою казки є Миша (клас Миша), яка є Твариною та володіє Хвостом (клас Хвіст) як інструментом (клас Інструмент) для розбивання яйця (стосунок асоціації «розбиває»).

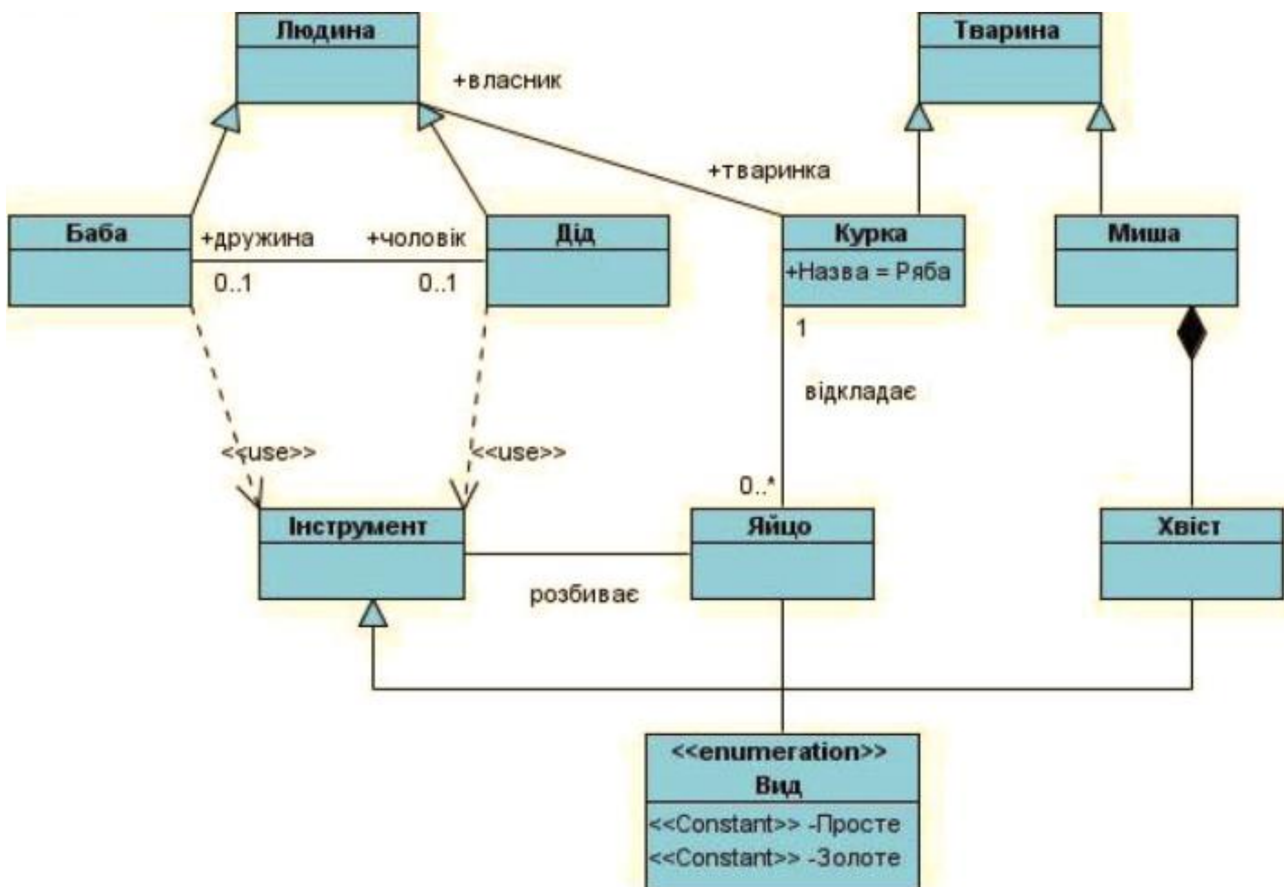


Рисунок 8.9. Діаграма класів для ілюстрації казки «Курка Ряба»

Контрольні запитання для самоконтролю

1. Для чого призначена діаграма класів? Назвати її основні елементи.
2. Дати визначення класу. Які є типи класів?
3. Що таке атрибути класу? Які є типи атрибутів?
4. Що таке операції класу? Які є типи операцій?
5. Зобразити та пояснити графічне позначення класу в UML.
6. Яке призначення стосунків між класами?
7. Назвати та охарактеризувати базові стосунки в UML.
8. Навести графічне зображення стосунків в UML.
9. Пояснити використання позначень спеціальних видів стосунків залежності.
10. Що таке стереотип? Які основні стандартні види стереотипів є в UML?

Лабораторне завдання

1. Ознайомитись із теоретичними відомостями та дати відповіді на контрольні питання.

2. Створити одну-дві діаграми класів, що деталізуватимуть окремі підсистеми зазначених предметних галузей, відповідно до індивідуального завдання (табл. 8.1). Для цього виділити основні класи об'єктів у системі, вказати для класів основні атрибути, операції, вид і напрямок асоціацій.

3. Оформити протокол лабораторної роботи, в якому описати елементи моделі.

Таблиця 8.1

Індивідуальні варіанти для створення діаграм класів

Вар.	Завдання
1	Корпоративна комп'ютерна мережа складається з декількох локальних сегментів, об'єднаних шлюзами. В кожному сегменті функціонує розподілена система управління базами даних, котра має такі атрибути: назва, кількість робочих станцій, кількість ланок, тип мережного протоколу, список операційних систем серверів. Один сегмент може бути з’єднаний з декількома шлюзами. Один шлюз може об'єднувати декілька сегментів. Шлюз має номер, назву, список IP-адрес, кількість з’єднань із сегментами. Сегмент має номер, назву, кількість вузлів, IP-адресу. Кожний сегмент використовує деякий протокол доступу до мережного середовища. Протоколи мають тип, назву, метод доступу
2	Акції акціонерного товариства (АТ) зберігаються у акціонерів, котрі купують акції у правління АТ. Правління має голову, телефон, факс, кількість членів, термін переобрання зборами акціонерів. Збори мають дату, кількість акціонерів, перелік рішень. Одне АТ може зберігати свої акції у декількох зберігачів, а один зберігач може зберігати акції декількох АТ. АТ має номер, назву, адресу, телефон. Зберігач має ім’я, адресу, телефон, ліцензію. Акція має номер, категорію, дату емісії, номінал

Продовження табл. 8.1

Вар.	Завдання
3	На користування автомобілем бухгалтерією пункту прокату видається документ. Пункт прокату має ліцензію, назву, адресу, номер телефону. Бухгалтерія має головного бухгалтера, телефон. Авто має марку, дату випуску, реєстраційний номер, дату реєстрації. Користувач має ПІБ, адресу, телефон, посвідчення водія. У документі на користування вказано реєстраційний номер автомобіля, дані користувача, дату видачі. На один автомобіль може бути видано декілька документів на користування, один користувач може мати документи на декілька автомобілів
4	Вищий навчальний заклад ліцензований для підготовки спеціалістів та магістрів з декількох спеціальностей, причому набором студентів займається приймальна комісія, яка має голову, адресу, телефон, факс, термін дії. Студент має ПІБ, вік, стать і отримує студентський квиток, залікову книжку, номер групи, факультет. Підготовкою фахівців з даної спеціальності займається випускова кафедра. При цьому кафедра може готувати фахівців з декількох спеціальностей і водночас з конкретної спеціальності може вести підготовку декілька кафедр. Вищий навчальний заклад має назву, адресу, телефон, рівень акредитації, ліцензію. Факультет має назву, декана, кількість кафедр, телефон деканату. Спеціальність характеризується номером, назвою, номером напряму, номером ліцензії, кількістю студентів. Кафедра має назву, номер, факультет, телефон, ПІБ завідувача
5	Студенти-дипломники виконують дипломні проєкти на випусковій кафедрі з конкретної спеціальності, котра має номер, назву, анотацію. Дипломник має керівників з основної частини проєкту, економічного обґрунтування та охорони праці. Кожний керівник може керувати декількома дипломниками. Випускова кафедра має назву, номер спеціальності, завідуючого, телефон. Дипломник має такі атрибути: ПІБ, номер групи, номер залікової книжки, тема диплому. Керівник працює на кафедрі і має вчену ступінь і посаду. Дипломний проєкт характеризується назвою, кількістю сторінок, кількістю додатків, датою захисту. Дипломні проєкти захищаються у Державній екзаменаційній комісії, котра має певну кількість членів, голову, телефон, терміни функціонування, кількість закріплених дипломників
6	Корпоративна мережа складається з декількох сегментів локальних мереж, кожний з яких функціонує згідно з певним протоколом. Протокол має назву, тип, метод доступу до мережі, правила використання. В кожному сегменті може бути декілька вузлів. Деякі вузли можуть належати декільком сегментам (шлюзи, мости, маршрутизатори). Мережа має такі атрибути: назва, тип, IP-адреса. Сегмент мережі має назву, діапазон IP-адрес, кількість вузлів, кількість серверів. Вузли (або станції) мають номер, IP-адресу, локальне ім'я, тип. Сервер характеризується іменем, типом, технічними даними, назвою операційної системи, IP-адресою

Продовження табл. 8.1

Вар.	Завдання
7	Товари знаходяться на складі. Товари з однаковою назвою можуть бути на різних складах. Покупець замовляє у постачальників певні товари, на що оформляється замовлення. Замовлення містить номер, дату, ПІБ замовника, ПІБ постачальника, список товарів, де для кожного товару вказано назву, кількість або вагу, вартість. Постачальник отримує товари у працівника складу і доставляє їх покупцеві, який сплачує вартість товарів згідно з рахунком, де зазначено номер, дату, ПІБ покупця, ПІБ постачальника та список товарів із зазначенням їх вартості. Працівник складу має табельний номер, посаду, телефон, факс. Товари мають артикул (номер), назву, кількість, вагу, вартість одиниці. Склади мають номер, адресу, телефон, площу (квадратні метри), об'єм (кубічні метри). Покупець і постачальник мають ПІБ, адресу, телефон, банківські рахунки
8	Автотранспортне підприємство (АТП) виконує пасажирські та вантажні перевезення і має назву, номер ліцензії, адресу, телефон, банківський рахунок. АТП має декілька відділів (дирекція, бухгалтерія, відділ пасажирських перевезень, відділ вантажоперевезень, технічний відділ, автопарк), в яких працюють співробітники на певних посадах. Відділ має назву, номер, телефон, кількість персоналу, начальника. Співробітники мають ПІБ, табельний номер, посаду, оклад, стаж роботи. Автотранспорт виконує перевезення згідно з рейсами, котрі характеризуються номером, пунктом відправлення, пунктом прибуття, датою і часом відправлення й прибуття. Автотранспорт керується водієм і має реєстраційний номер, тип, місткість, рік випуску. Один транспортний засіб може виконувати багато рейсів, причому деякі рейси можуть бути виконані різними транспортними засобами
9	У кожному місті маються житлово-комунальні контори (ЖКК), котрі здійснюють обслуговування жителів будинків, постачаючи газ, електроенергію та воду. Кожна ЖКК має номер, адресу, телефон, директора. В ЖКК мається декілька відділів (дирекція, бухгалтерія, інспектори з прийому громадян, водії та робітники різних спеціальностей), причому кожний співробітник має табельний номер, посаду, спеціальність, оклад, адресу проживання. Жителі будинків сплачують вартість комунальних послуг і характеризуються ПІБ, ідентифікаційним кодом, адресою, службовим, домашнім та мобільним телефонами, списком номерів квитанцій про сплату рахунків. На балансі ЖКК, як правило, знаходиться декілька будинків, котрі характеризуються, адресою, кількістю поверхів, кількістю квартир, кількістю мешканців, наявністю котельної, телефоном двірника. На виконання робіт жителі будинку подають заявку, в якій вказано номер, дату, адресу й телефони жителя та перелік робіт, котрі замовляються. На виконання робіт майстер оформляє накладну, в котрій зазначено: номер, дату, адресу, ПІБ замовника, наявність пільг, ПІБ майстра, список матеріалів та список виконаних робіт із зазначенням вартості, сумарну вартість.

Закінчення табл. 8.1

Вар.	Завдання
10	Приватні підприємства, що розробляють програмні продукти (ПП), мають назву, номер державної ліцензії, адресу, телефон, банківський рахунок, директора. У підприємстві мається декілька відділів, де працюють співробітники (дирекція, бухгалтерія, відділ розробки ПП, відділ тестування ПП, відділ супроводження ПП та ін.). Відділ має назву, начальника, кількість співробітників, список кімнат із зазначенням телефонів, список комп'ютерів та оргтехніки. Програмний продукт поставляється замовникові і має назву, список розробників, вартість, документацію, дистрибутив, правила використання. Замовниками можуть бути як фізичні, так і юридичні особи. Фізична особа має ПІБ, адресу, телефон, ідентифікаційний код. Юридична особа має зареєстровану назву, єдиний код платника податків, телефон, адресу, банківський рахунок. Кожний замовник може замовити декілька ПП, на кожний з яких він отримує ліцензію, в якій вказано назву продукту, дату продажу, вартість, кількість серверів та робочих станцій, які він може підключити, терміни апгрейдів

Самостійна робота 5
Планування програмного проєкту

Теоретичні відомості

1 Програмне забезпечення для керування проєктами

Системи керування проєктами – комплексні програмні засоби для контролю виконання проєктів, автоматизації складових керування наявними ресурсами (часовими, фінансовими, людськими) й організації взаємодії між учасниками проєктів.

Нормативне наповнення настанов щодо програм керування проєктами наведено в ДСТУ ISO/IEC/IEEE 16326:2015 «Розроблення систем та програмного забезпечення. Процеси життєвого циклу. Керування проєктами» (ISO/IEC/IEEE 16326:2009, IDT)[16].

Проєкти складаються з послідовності етапів і при цьому вимагають складної взаємодії різних підрозділів. Виконання таких завдань може складатися з десятків більш дрібних завдань. Над ними працюють десятки, сотні, а інколи навіть тисячі співробітників організації.

При виконанні проєктів важливість мають три складові[17]:

– якість кінцевого продукту відповідно до зазначених вимог;
– терміни виконання робіт;
– бюджет, в який необхідно вкластися.

Метою керування проєктом є пошук балансу між обсягом робіт, ресурсами (фінансами, витратами на оплату роботу виконавців, матеріалами, енерговитратами, арендою приміщень тощо), часом, якістю та ризиками. Завданнями керування проєктом є: визначення вимог до проєкту, складання календарного плану спільної роботи над проєктом, розподілу ресурсів, керування витратами, документування й адміністрування проєкту, керування ризиками, коригування характеристик, планів і підходів відповідно до думки і очікувань різних учасників тощо.

Керування проєктом здебільшого є завданням менеджера проєкту, що вимагає від цього фахівця досвіду, знань, інтуїції. Звичайно жодна комп'ютерна система повною мірою не здатна повністю замінити менеджера проєкту. Проте така система може стати для нього гарним помічником і забезпечити єдиний інформаційний простір для виконання рутинних завдань (наприклад, побудови

[16] ДСТУ ISO/IEC/IEEE 16326:2015 Розроблення систем та програмного забезпечення. Процеси життєвого циклу. Керування проєктами (ISO/IEC/IEEE 16326:2009, IDT). URL: http://online.budstandart.com/ru/catalog/doc-page?id_doc=67052

[17] Як працює система керування проєктами? URL: https://e-contentum.com.ua/projects-features

діаграми Ґанта), щоб у менеджера залишалося більше часу на вирішення нетривіальних завдань.

Програмне забезпечення (ПЗ) для керування проєктами систематизує дані про:

- поетапну послідовність розробки та виконання проєкту;
- список завдань та виконавців, а також про розподіл ресурсів;
- бюджет проєкту;
- терміни виконання етапів і його складових;
- комунікації між учасниками проєкту;
- попередження про можливі ризики у проєкті;
- робоче навантаження тощо.

2 ProjectLibre

Програма ProjectLibre розроблена для керування проєктами і є безкоштовним аналогом Microsoft Project. ProjectLibre є кросплатформним ПЗ, сумісним з операційними системами Microsoft Windows, Linux, Mac OS X. Програма виграла нагороду InfoWorld "Кращий з відкритим кодом". Вона підтримує багато мов, в тому числі українську. Є хмарна версія для керування кількома проєктами.

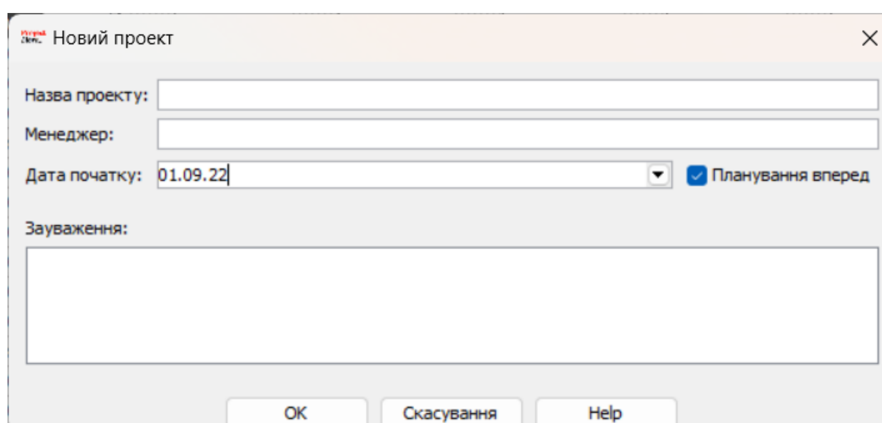
ProjectLibre підтримує сумісність з форматами файлів Microsoft Project.

2.1 Створення нового проєкту

Створити новий проєкт на основі одного з уподобаних шаблонів https://www.canva.com/templates/ можна командою *Файл / Новий*.

У діалоговім вікні, що відкриється, треба внести дані:

- назва проєкту;
- менеджер (керівник проєкту);
- дата початку (або дату закінчення) проєкту;
- тип планування (при плануванні вперед варто вказати дату початку, а якщо ні – дату закінчення проєкту);
- зауваження або коментарі (часто сюди вписується мета).



Дату початку проєкту можна вписати вручну або вибрати з календарика натисканням стрілки ▼ праворуч від поля введення.

Після натискання кнопку *OK* буде створений новий файл проєкту, при цьому проєкт автоматично перейде на час початку.

Детальнішу інформацію про проєкт можна задати, якщо скористатись командою *Файл / Інформація*.

У цьому діалоговому вікні на вкладці *Statistics* можна ознайомитися з поточною статистикою по відкритому проєкту: вартість і тривалість проєкту, поточний відсоток виконання тощо.

Зберегти проєкт слід командою *Файл / Зберегти як*, зазначивши ім'я файлу.

2.2 Календар проєкту

Календар (Calendar) – це інструмент формування розкладу етапів проєкту, що дозволяє встановлювати розподіл робочого і неробочого часу задач і ресурсів.

Командою *Файл / Календар* можна налаштувати робочий календар проєкту: задати час початку роботи, кількість робочих годин на день, тиждень та місяць, а також розклад змін при змінній роботі.

Тут можна визначити додаткові неробочі (святкові) дні та призначити час початку, перерви, час закінчення робочого дня.

До речі, в україномовній версії програми є помилки перекладу. Так поле «*Від кого*» цього діалогового вікна слід сприймати як «*З*», а «*Кому*» – як «*По*».

Change Working Calendar

Для: П'ятиденка

- По-умовчанню
- Неробочий час
- Не по-умовчанню

Від кого: Кому:

Квітень 2024

Н	П	В	С	Ч	П	С
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Травень 2024

Н	П	В	С	Ч	П	С
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Червень 2024

Н	П	В	С	Ч	П	С
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Липень 2024

Н	П	В	С	Ч	П	С
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Серпень 2024

Н	П	В	С	Ч	П	С
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

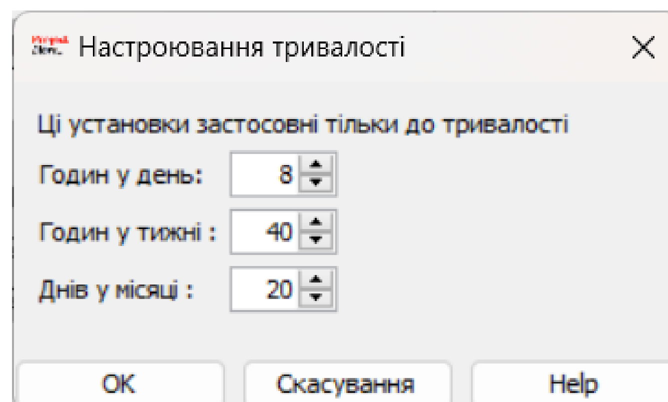
Вересень 2024

Н	П	В	С	Ч	П	С
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

Новий… Настроювання… OK Скасування Help

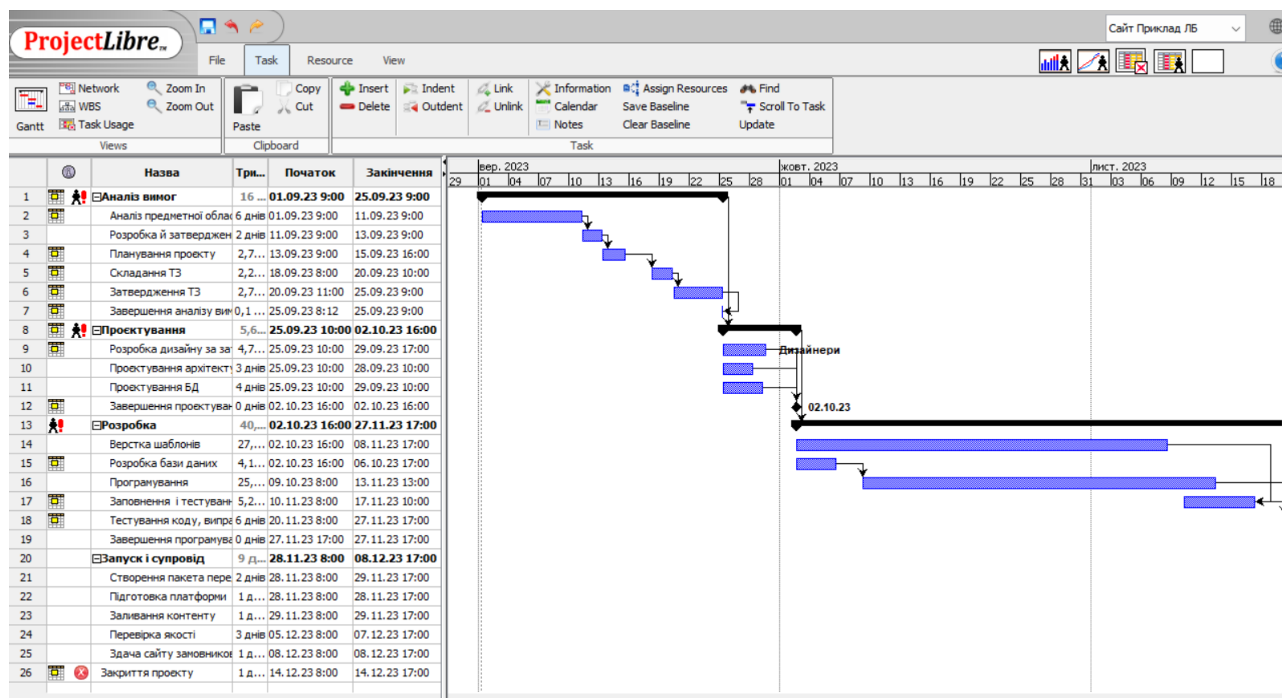
Як видно з цього рисунка, ProjectLibre надає три усталених календарі: стандартний календар (п'ятиденка), 24-годинний календар і календар з нічними змінами. Звичайно базовим є стандартний календар. За потреби можна створити свій власній календар, скориставшись кнопкою *Новий.*

Налаштувати тривалість робочого тижня та робочого дня можна після натискання кнопки *Настроювання*.



2.3 Задачі

На головній вкладці *Task (Задача)* розміщено інструментарій для роботи зі списком задач (етапів проєкту) і налаштування таймлайну проєкту у вигляді діаграми Ґанта. Вигляд діаграми Ґанта автоматично синхронізується з таблицею задач.



Діаграма Ґанта – це графічне подання задач проєкту. Кожна смуга діаграми є графічним поданням тривалості запланованої задачі. Діаграма Ґанта є зручним інструментом, який дозволяє користувачеві оцінити дані проєкту. На діаграмі можуть бути відзначені сукупні задачі, відсотки завершення, покажчики залежності робіт, віхи проєкту, мітка теперішнього моменту часу тощо.

Планування задач проєкту тією чи іншою мірою здійснюється в період усього строку виконання проєкту. На самому початку життєвого циклу проєкту звичайно розробляється неофіційний попередній план про те, що буде необхідно виконати для реалізації проєкту.

Рішення про вибір проєкту значною мірою ґрунтується на оцінках попереднього плану. Формальне й детальне планування проєкту починається після ухвалення рішення про його реалізацію. Визначаються ключові точки проєкту, формулюються задачі та їхня взаємозалежність.

ProjectLibre має набір засобів для розробки плану проєкту: таблицю для введення задач і діаграму Ґанта (команда *Task / Views / Gantt*), графічне подання ієрархічної структури робіт *Task Views / WBS*), мережний графік (команда *Task / Views / Network*).

Задача (*Task*) у плані проєкту задає певну діяльність, яка необхідна для досягнення конкретних результатів (кінцевих продуктів нижнього рівня, deliverables). На виконання задачі потрібен час. Момент завершення задачі означає факт одержання кінцевого продукту, тобто результату виконання задачі.

Виділяють три основні типи задач:

– *Зведені або складені задачі* (*Summary task*), що охоплюють декілька підлеглих задач;

– *Підзадачі* (*Subtask*) – невеликі задачі, що є частиною якоїсь зведеної задачі;

– *Віхи* (*Milestone*) – це задачі з нульовою тривалістю, які позначають проміжні цілі проєкту. Віха – ця подія або дата в ході виконання проєкту. Вона використовується для відображення стану завершеності тих чи інших задач. У контексті проєкту менеджери використовують віхи для того, щоб позначити важливі проміжні результати, які повинні бути досягнуті в процесі реалізації проєкту. Послідовність віх проєкту називається *планом по віхах*. Важливою відмінністю віхи від задачі є те, що вона не має тривалості.

Організація задач проєкту охоплює такі дії:

– *Визначення задач* – розбивка очікуваних результатів проєкту на більш дрібні задачі, які легше піддаються обліку й керуванню;

– *Визначення послідовності задач* – встановлення залежності між задачами й визначення обмежень у властивостях задач;

– *Оцінка тривалості задач і проєкту в цілому* – оцінка часу, необхідного для завершення всіх задач проєкту.

– *Розрахунки вартості задач* – оцінка вартості реалізації проєкту.

Для створення задачі в таблиці задач треба вибрати комірку *Назву* (*Name*) і вказати назву задачі. Дали варто задати тривалість задачі або оцінку тривалості задачі (позначається знаком питання).

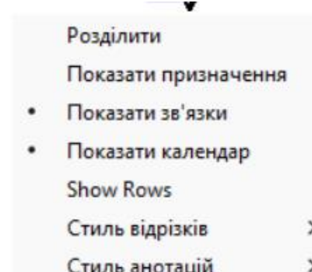
При подвійному клацанні мишею по задачі або при виборі меню *Задача / Інформація* (*Task / Information*) відкриється діалогове вікно *Інформація про задачу.* Тут можна додати більш докладний опис задачі, а також вказати коментарі та посилання на відповідні документи.

Усталено тривалість кожної задачі встановлюється 1 день. Однак на практиці тривалість переважно буде іншою.

Для створення віхи треба вказати її назву й встановити тривалість задачі значенням 0. Віхи в ProjectLibre на діаграмі Ґанта подані у вигляді ромбів, а не смужок.

Перелік стовпців таблиці *Задачі* можна змінювати за потреби, натиснувши

праву кнопку мишки на заголовку й вибравши в контекстному меню пункт *Додати колонку* або *Сховати колонку*.



Для того щоб розділити задачу на частини (Split task) так, щоб її частини можна було виконувати в різний час, або якщо буде потреба призупинити задачу на час (відпустка або хвороба виконавця), треба натиснути праву кнопку миші на задачі в діаграмі Ґанта й у контекстному меню вибрати пункт *Розділити* (*Split*).

Усталено в ProjectLibre використовується шкала тижнів і днів для відображення діаграми Ґанта. Щоб побачити укрупнену картину проєкту, варто перемкнутися на шкалу місяців, років, кварталів тощо. Для цього є кнопки зміни масштабу *Приблизити* (*Zoom In*) і *Віддалити* (*Zoom Out*) у режимі перегляду задач.

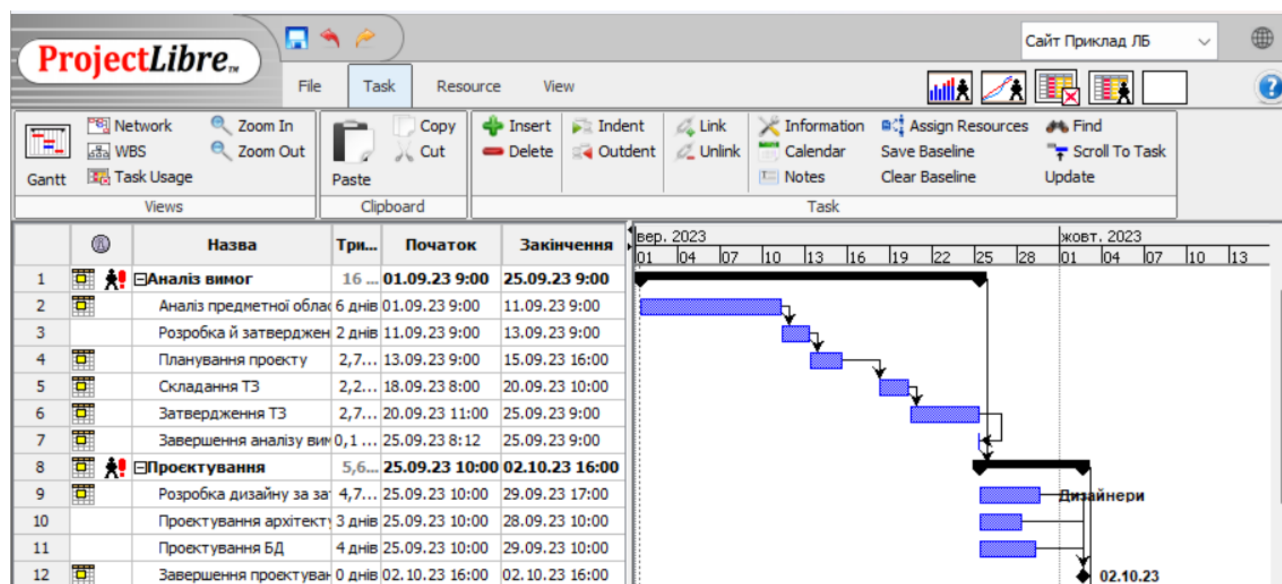
Критична задача (Critical task) – це задача, що не має резерву часу. Будь-який зсув дати завершення такої задачі на пізніший строк спричинить зсув наступної задачі або запізнювання проєкту в цілому.

Критичний шлях (Critical path) – це послідовність взаємозалежних критичних задач, що з'єднує початкову й кінцеву дати проєкту. Іншими словами це найбільш довгий маршрут у мережній діаграмі проєкту. Критичний шлях визначає мінімально можливу тривалість проєкту при заданій структурі. Будь-які затримки на критичному шляху затримують закінчення проєкту.

В ProjectLibre на діаграмі Ґанта критичні задачі відображаються червоним кольором, а не критичні – синім. Для некритичних задач також можна відобразити **часовий резерв** (Total Slack) у контекстному меню на діаграмі Ґанта.

2.4 Структурування задач

Після внесення зведених задач у таблицю необхідно внести в порожні рядки підзадачі і структурувати їх командами *Задача / Відступ* (*Task / Indent*) і *Задача / Виступ* (*Task / Outdent*). Відповідні підзадачі розташовані з відступом вправо щодо зведеної задачі.

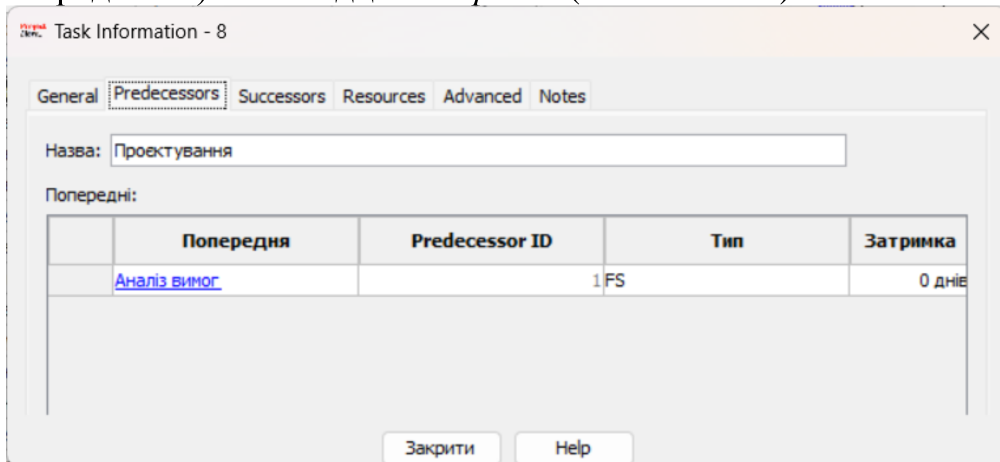


Будь-яка дія над зведеною задачею – видалення, переміщення або копіювання – застосовується також до всіх вкладених підзадач.

На діаграмі Ґанта в розділі графіка робіт тривалість кожного завдання показана у вигляді кольорових смужок різної довжини.

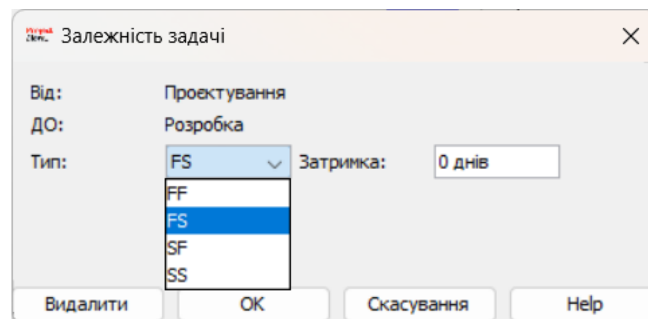
Іноді, навіть після проороблення розкладу, багато непередбачених змін можуть вплинути на виконання проміжних задач. Це може привести до порушення виконання інших (залежних) задач.

Пов’язати задачі можна командою *Задача / Зв'язок* (*Task / Link*) або через діалогове вікно *Інформація про задачу* (*Task Information*), вказавши ID попередника (попередників) на вкладці *Попередні* (*Predecessors*).



Зведені задачі можуть бути пов'язані з іншими завданнями або з підзадачами між зведеними групами. Підзадачі також можуть бути зв'язані між собою. Зв'язки бувають чотирьох типів:

Зв'язок	Код	Опис	Вигляд на діаграмі Ґанта
Закінчення-Початок (Finish-Start)	**FS**	Попередня задача закінчується і починається наступна задача	
Початок-Закінчення (Start-Finish)	**SF**	Початок попередньої задачі визначає закінчення наступної задачі	
Закінчення-Закінчення (Finish-Finish)	**FF**	Обидві задачі закінчуються одночасно	
Початок-Початок (Start-Start)	**SS**	Задача починається одночасно з попередньою	



Видалити можна командою *Задача / Видалити зв'язок*. Не варто намагатися вилучити зв'язок шляхом натисканням клавіші Del для позиції в стовпці попередньої задачі, оскільки при цьому віддаляється вся задача.

Іноді задачі можуть залежати від інших факторів, які викликають затримку або перекриття за часом, що приводить до потреби вказівки затримки або раннього початку в описуваному зв'язку. Затримка і ранній початок можуть бути зазначені в одиницях часу або у відсотках від тривалості попередньої задачі:

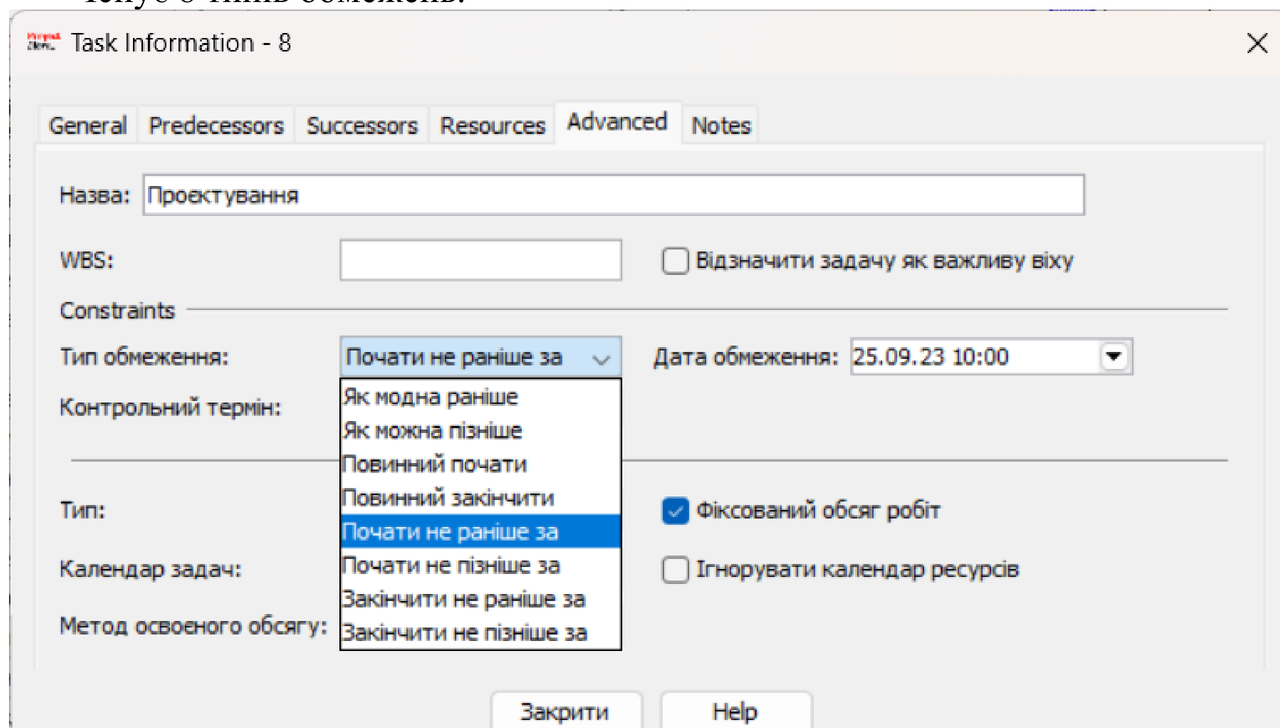
1. відкрийте діалогове вікно *Інформація про завдання* (*Task information*) за допомогою подвійного клацання на задачі;
2. вибрати вкладку *Попередні*;
3. вибрати стовпець *Затримка*;
4. вказати у полі тривалість у днях або відсотках;
5. натиснути кнопку *Закрити*.

Для того, щоб розділити задачу на частини так, щоб їх можна було виконувати в різний час, треба натиснути праву кнопку миші на задачі в діаграмі Ґанта й у контекстному меню вибрати пункт *Розділити*.

Деякі задачі повинні бути завершені до певної дати. Також може виникнути потреба зазначення проміжних віх. Тобто при плануванні можливі певні обмеження.

Задавати обмеження можна в діалоговому вікні *Інформація про завдання* на вкладці *Додатково* (*Advanced*), де вказують *Тип обмеження*, натиснувши кнопку ▼ і вибравши тип обмеження зі списку. Крім того, вказують значення *Дата обмеження* і натискають кнопку *ОК*.

Існує 8 типів обмежень.



Обмеження бувають двох категорій – гнучкі й негнучкі. Негнучкі обмеження суттєво обмежують можливість планування, у той час як гнучкі обмеження дозволяють розрахувати розклад і зробити відповідні зміни, залежно від

типу зазначеного обмеження. Негнучкі обмеження можуть викликати конфлікти між наступними й попередніми завданнями, що може потребувати прибрати такі обмеження.

Щоб не ставити негнучке обмеження, варто задати контрольні строки, що ніяк не впливатиме на розклад завдання. Якщо строк пройшов, а завдання не було виконано, Projectlibre покаже це в стовпці *Індикатор* (хрестик у червоному кружку у списку завдань при виведенні діаграми Ґанта).

2.5 Ресурси

Ресурси бувають двох типів – робота й матеріали.

Ресурси типу ***Робота*** виконують задачу шляхом витрати часу (виділення часу на його виконання). Звичайно це люди та/або устаткування, які призначені виконувати роботу в рамках проєкту.

Ресурси типу ***Матеріал*** – це запаси видаткових матеріалів і компонентів, які потрібні для виконання проєкту. Матеріальні ресурси можна відслідковувати й призначати для задачі.

Доступний для виконання роботи набір ресурсів називають *пулом ресурсів*. Після визначення необхідної кількості ресурсів потрібно встановити час і доступність кожного ресурсу. Для трудових ресурсів кількість часу, яку вони можуть працювати, вимірюють в годинах, днях, місяцях або роках, а кількість матеріальних ресурсів зазначають у відповідних одиницях виміру.

Відпочинок і неробочий час можуть бути визначені й включені для кожного ресурсу. У такий спосіб можна довідатися про перевантажений ресурс, тобто коли ресурс призначено для виконання декількох задач водночас або коли ресурсу призначається більше роботи, ніж він може виконати за певний час.

Створення списку для пулу ресурсів проводиться у вигляді списку ресурсів:

	Назва	Тип	Ініціали	Максимальн...	Стандартна ставка	Ставка понаднормових	Нараховувати	Основний календар
1	Проектувальники	Робота	Петренко	100%	500 грн./година	700 грн./година	Пропорційно	П'ятиденка
2	Аналітики	Робота	Абрамов	100%	400 грн./година	700 грн./година	Пропорційно	П'ятиденка
3	Дизайнери	Робота	Денисова	100%	550 грн./година	700 грн./година	Пропорційно	П'ятиденка
4	Програмісти	Робота	Пасічник	100%	1000 грн./година	1200 грн./година	Пропорційно	П'ятиденка
5	Тестувальники	Робота	Тарасенко	100%	450 грн./година	700 грн./година	Пропорційно	П'ятиденка
6	Планшети	Матеріал	П		20000 грн.		Пропорційно	
7	Мобільні телефони	Матеріал					Пропорційно	
8	ПК	Матеріал					Пропорційно	
9	Хостинг	Матеріал					Пропорційно	

Іноді може знадобитися змінити розподіл роботи над задачею. ProjectLibre має вісім визначених профілів:

– плоский – контур з рівномірним розподілом роботи. Задається усталено;

– завантаження наприкінці – пік активності виникає наприкінці задачі;

– завантаження на початку – пік активності виникає на початку задачі;

– подвійний пік – у задачі існують два піки активності;

– ранній пік – пік активності на початковому етапі, але з рампою до піка активності;

– пізній пік – пік активності наприкінці проєкту, але з рампою;

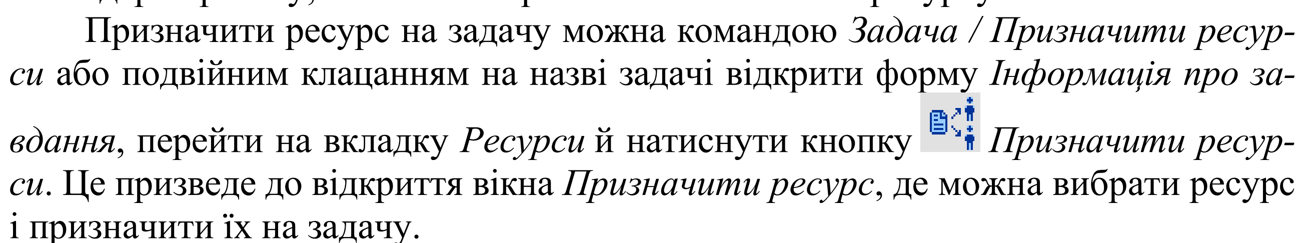
– "дзвін" – одиночний пік активності в середині проєкту;

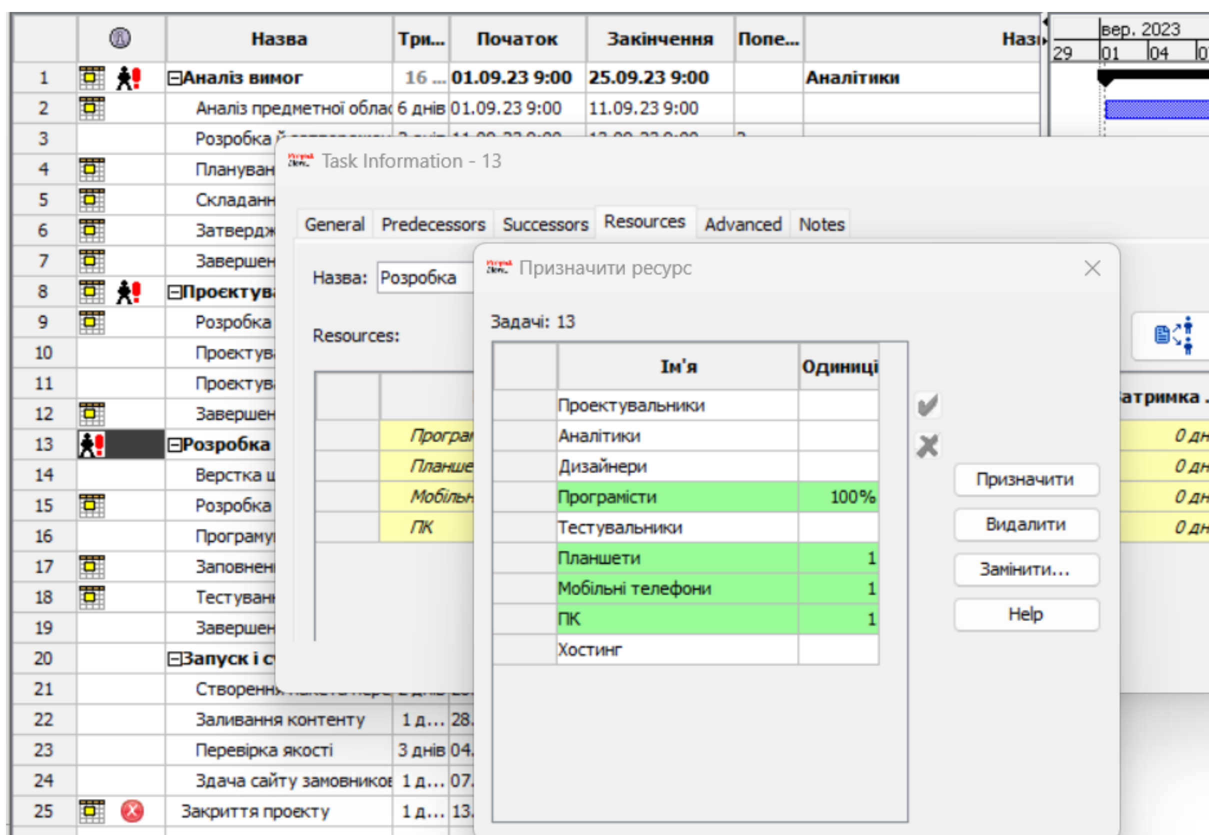
– "черепаха" – плоске завантаження без виражених піків, з рампою на початку й наприкінці.

Змінити робочий контур для призначення можна у поданні *Використання задач* (*Task Usage*).

Ресурси типу *Матеріал* не містять величини *Максимальне використання*.

При клацанні в таблиці ресурсів по назві ресурсу буде виведене вікно *Інформація про ресурс*. На вкладці *Загальне* цього вікна можна вказувати різні базові календарі, наприклад, індивідуальний календар, відмінний від загального календаря проєкту, і змінювати робочий час кожного ресурсу.

Призначити ресурс на задачу можна командою *Задача / Призначити ресурси* або подвійним клацанням на назві задачі відкрити форму *Інформація про завдання*, перейти на вкладку *Ресурси* й натиснути кнопку  *Призначити ресурси*. Це призведе до відкриття вікна *Призначити ресурс*, де можна вибрати ресурс і призначити їх на задачу.



В такий спосіб можна призначати ресурси й змінювати інформацію призначення відповідно до вимог.

При призначенні ресурсів на завдання вартість розраховується через перемноження кількості годин роботи на погодинну ставку робітників (зазначені відсотки) або кількість необхідних одиниць (одиниці виміру) матеріалів:

*Робота (Work) = Тривалість (Duration) * Одиниці (Units)*

*(W = D * U)*

Наприклад, для виконання певної роботи, скажім, розробки деякої програми навчання, встановлена тривалість в 5 днів і призначено 3 ресурси із завантаженням 100%. Припускаючи, що в дні усталено 8 робочих годин, кожний із цих ресурсів повинен буде працювати по 8 годин усі 5 днів.

Тепер припустимо, що на проєкт призначено ще два ресурси з доступністю 50%, які, відповідно, будуть працювати по 4 години на день. У результаті робота буде виконана раніше, ніж у зазначені 5 днів і кількість роботи (часу), витрачена першими трьома ресурсами, зменшиться.

Окремі витрати по ресурсах підсумуються.

У таблиці ресурсів повинні бути заповнені значення стандартної вартості ресурсів для звичайних і понаднормових робочих годин, або вартість одиниць видаткових матеріалів і компонентів. Сума фіксованих витрат на використання ресурсів і часових витрат становлять загальну величину витрат.

Ресурси – це звичайно люди, а отже, вони вносять непередбачуваність – можуть змінитися дати й строки їх доступності для залучення до робіт, їм може бути призначено занадто багато роботи або вони можуть звільнитися в середині робіт над проєктом. У призначеннях і розподілі робіт варто передбачати простір для маневру. Projectlibre дозволяє організувати ресурси для різних дат початку або закінчення роботи, робити перерви в призначеннях, дозволяючи ресурсам працювати над іншими завданнями, а також за потреби призначати понаднормові роботи – іншими словами, можлива зміна контурів призначень. Ці можливості забезпечують види *Використання задач* і *Використання ресурсів*.

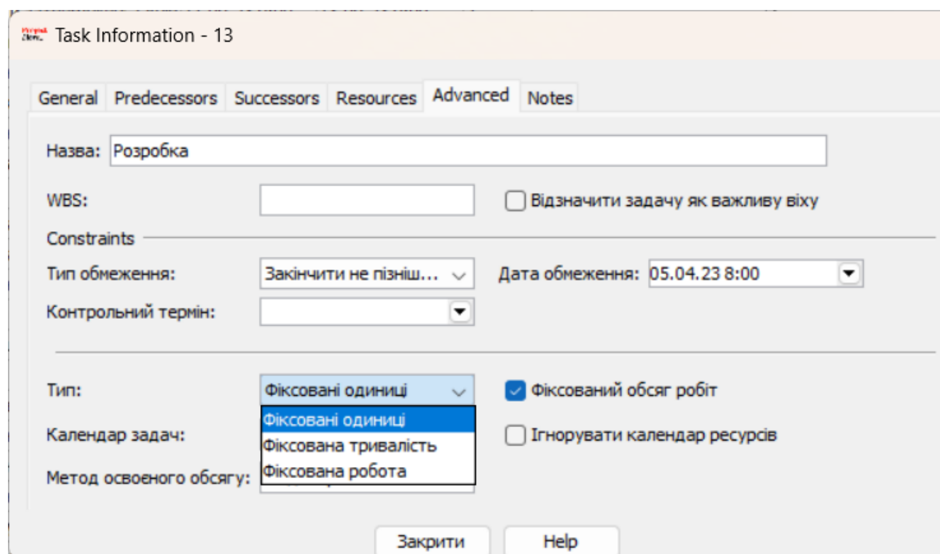
В Projectlibre можна керувати розрахунками ресурсів за допомогою встановлення типу завдання:

- фіксовані одиниці (*Fixed Units*);
- фіксована робота (*Fixed Work*);
- фіксована тривалість (*Fixed Duration*).

Кожний тип базується на тому, що одна зі змінних фіксується. Наприклад, для задачі типу *Fixed Units* буде перерахована робота або тривалість, залежно від того, що необхідно, але завантаження ресурсів (одиниці) залишиться незмінним. Подібним чином для задач із типом *Fixed Duration* при зміненні завантаження ресурсів (одиниць) буде зроблений перерахунок роботи. При керування проєктом керівник проєкту має проаналізувати кожну задачу і вибрати для неї доречний тип. Усталено тип задачі заданий як *Fixed Units*.

Змінення типу задачі виконується і вікні *Інформація про завдання* або подвійним клацанням на назві задачі або за допомогою меню *Задача / Інформація*.

Далі у вікні, що відкриється, треба вибрати вкладку *Додатково* і зі списку *Тип* вибрати необхідний тип задачі:



2.6 Витрати

Нарахування витрат може бути за потреби задано до початку роботи (метод *На початок*) або, якщо оплата проводиться тільки після закінчення роботи, використовується метод *По закінченню*. Для задач, які виконані частково, оцінка витрат розраховується пропорційно виконанню, тобто при виконанні робіт на 20% розрахункові витрати становитимуть 20% від оцінки повних витрат. Тут метод нарахування витрат відіграє важливу роль. Однак це не впливає на загальну суму витрат по проєкту.

	Назва	Тип	Ініціали	Максимальн...	Стандартна ставка	Ставка понаднормових	Нараховува
1	Проектувальники	Робота	Петренко	100%	500 грн./година	700 грн./година	Пропорційно
2	Аналітики	Робота	Абрамов	100%	400 грн./година	700 грн./година	Пропорційно
3	Дизайнери	Робота	Денисова	100%	550 грн./година	700 грн./година	Пропорційно
4	Програмісти	Робота	Пасічник	100%	1000 грн./година	1200 грн./година	Пропорційно
5	Тестувальники	Робота	Тарасенко	100%	450 грн./година	700 грн./година	Пропорційно
6	План						Пропорційно
7	Мобі						Пропорційно
8	ПК						Пропорційно
9	Хості						Пропорційно

Resource Information

General | Costs | Resource Availability | Tasks | Notes

Назва: Тестувальники

Таблиця норм витрат

A B C D E

	Дата дій	Стандартна ставка	Ставка понаднормових	Витр
	01.01.70 0:00	450 грн./година	700 грн./година	

Закрити Help

Для внесення витрат треба відкрити діалогове вікно *Інформація про ресурс* і вибрати вкладку *Вартість*. Фактична дата витрат може збігатися з датою початку проєкту. Якщо це не так, слід вказати цю дату або вибрати її з відривного календаря.

У вікні *Інформація про ресурс* на вкладці *Вартість* можна вказати до 5 різних тарифних ставок для вибраного ресурсу і вказати такі ставки:

- стандартна ставка;
- ставка за понаднормові години роботи;
- витрати за кожне використання ресурсу.

Стандартна ставка для ресурсів типу *Робота* усталено вказується значенням в грн/годину, але можна за потреби вказати ставку в грн/рік або грн/місяць. Для ресурсів типу *Матеріали* використовуються відповідні одиниці виміру, зазначені в *Одиницях виміру*.

Ставку за понаднормові години роботи треба вказувати, навіть якщо вона дорівнює стандартній ставці.

Витрати за використання ресурсу є додатковою вартістю, наприклад, доставки, встановлення і налаштування тощо.

Якщо ставка може змінюватися, треба клацнути на другій вкладці B, вказати відповідні дати й внести для них нові значення ставок: стандартна, понаднормова і за використання. Тут можна вказати абсолютні величини або збільшення чи то зменшення у % від ставок, перерахованих на першій вкладці.

Завантаження ресурсу є перевищеним, коли загальна сума його погодинних робіт (роботи, розподілені по певних періодах у проєкті) перевищує максимально можливу кількість одиниць ресурсу. Якщо ресурс уже запланований до роботи над іншим завданням, завантаження ресурсу може бути перевищено.

Перевантажені ресурси можна знайти, відкривши додаткову панель із виведенням *Гістограми завантаження ресурсів*[18].

При виявленні перевищення завантаження можна:

- затримати виконання завдання;
- розділити завдання;
- призначити додаткові ресурси або призначити інші ресурси

Варто перевірити значення змінної *Залишилася наявність*, щоб довідатися доступні інтервали часу для ресурсу. Projectlibre пропонує функцію вирівнювання, яка може вирішити проблеми розподілу ресурсів шляхом поділу або затримки завдань. Вирівнювання проводять з використанням таких факторів:

- доступний час;
- пріоритети, залежності й обмеження задач;
- ідентифікатори задач;
- дати розкладу.

2.7 Підготовка звітів по проєкту

У ProjectLibre не так багато варіантів для формування звітів, якщо порівнювати з комерційними продуктами. Однак подання інформації у звітах проводиться з достатньою якістю.

---

[18] Step by Step Setting up a project in ProjectLibre. URL: https://www.udemy.com/course/setting-up-a-project-in-projectlibre/

Дата звіту про стан (або контрольна дата, Status date) – це дата, на яку слід одержати інформацію про стан проєкту. Усталено датою звіту про стан вважається поточна дата. Користувач може вибирати контрольну дату довільно, але вона має бути розташована на осі часу «лівіше» поточної дати.

Фільтрація дозволяє в будь-який момент накладати певні умови на спосіб виведення інформації (у тому числі і для звітів). Можна тимчасово приховати якусь інформацію, змінити порядок сортування даних, об'єднати їх у групи (наприклад, по виконанню, критичності, по залучених ресурсах). Для цього використовується пункт команда *Перегляд / Фільтри* (*View / Filters*).

Для формування звітів по проєкту є команда *Перегляд / Звіт* (*View / Other views / Report*).

Тип звіту – це категорія звіту, що визначає, якого роду інформація про проєкт буде розміщена в друкований документ. В ProjectLibre є 4 типи звітів:

- загальні відомості про проєкт (Project details);
- інформація про ресурси (Resource information);
- інформація про задачі (Task information);
- призначення (Who Does What).

Контрольні запитання

1. Яке призначення систем керування проєктами?
2. Що таке діаграма Ґанта?
3. Що таке критична задача й критичний шлях? Що таке часовий резерв задачі?
4. Які типи залежностей між задачами існують і як вони впливають на розрахунки календарного плану проєкту?
5. Як використовувати запізнювання й випередження при формуванні залежностей між задачами?
6. Для чого необхідні календарі?
7. Що таке віхи й навіщо вони потрібні?
8. Для чого структурувати задачі проєкту?
9. Що таке обмеження задач і які вони бувають?
10. Що таке крайній строк виконання задачі і як його можна використовувати?
11. Як можна врахувати перерви у виконанні задач?

Завдання

Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними принципами керування проєктами в цілому й зокрема із системою ProjectLibre.

Дати відповіді на контрольні запитання.

Лабораторна робота 9
Календарне планування робіт проєкту

Мета роботи: набути практичних навичок роботи планування робіт із застосуванням спеціалізованого програмного забезпечення.

Теоретичні відомості

Склад робіт з розробки програмного продукту

Розглянемо короткий опис етапів розробки у технологічній послідовності з визначенням зв'язків між ними.

Аналіз вимог. На цьому етапі спочатку здійснюється аналіз предметної області: виявлення потреб майбутніх користувачів; визначення технічних вимог до структури сайту, змісту розділів і підрозділів; з'ясування вимог до дизайну; визначення вимог до структури й функцій програмних модулів; визначення вимог до архітектури, системи керування контентом сайту тощо.

За результатами аналізу складається концепт проєкту, наприклад, сайту, який обговорюється із замовником і командою розробників проєкту. Після затвердження концепту розробники приступають до детального аналізу і формування вимог до розроблюваної системи.

Паралельно з розробкою технічного завдання (ТЗ) здійснюється планування проєкту, де визначають склад і строки виконання робіт, ресурси і бюджет проєкту. Результати цього наводяться в ТЗ, яке погоджується і затверджується замовником.

Проєктування. Дизайн сайту повинен відповідати встановленим цілям і бути функціональним. Для створення ескізів дизайнер аналізує концепт і ТЗ. На цьому етапі також проводиться проєктування архітектури сайту і проєктування бази даних згідно з ТЗ.

Програмування й верстка. Верстка шаблонів сторінок здійснюється відповідно до керівництва зі стилю оформлення.

Паралельно з версткою починається розробка бази даних, після чого виконується реалізація спроєктованої системи.

Заповнення сторінок контентом і тестування на реальних даних повинні початися за 10 днів до завершення програмування. Робота з верстки до цього моменту вже має бути закінченою.

Доробка коду й виправлення помилок здійснюється одразу по завершенню програмування й наповнення.

Запуск і супровід. До передачі замовникові й запуску сайту треба сформувати комплект документації («пакет передачі»), що містить всі вихідні файли,

зображення, шаблони, інструкції та ін., необхідні особі, яка буде супроводжувати сайт після його впровадження.

Водночас із формуванням пакета передачі проводиться підготовка платформи: розміщення сайту на хостинговому сервері, налаштування й підтримка DNS. Потім здійснюється перенесення сайту (заливання контенту) на сервер замовника.

Після успішного розміщення на сервері сайт тестується ще раз для усунення помилок, пов'язаних з особливостями розміщення.

Після завершення всіх робіт у встановлений термін з необхідним рівнем якості замовник підписує акт про впровадження, здійснює взаєморозрахунок з виконавцем згідно з договором. Керівник здійснює закриття проєкту.

Контрольні запитання для самоконтролю

1. Назвати технологічні етапи розробки програмного продукту.
2. На якому етапі із замовником з'ясовуються вимоги до структури, дизайну, функціоналу програмного продукту?
3. На якому етапі затверджується концепт розроблюваної системи?
4. Що містить ТЗ?
5. Що відбувається на етапі проєктування програмного продукту?
6. На якому етапі починається розробка бази даних для проєкту?
7. На якому етапі відбувається верстка шаблонів сторінок створюваного сайту?
8. Коли виконується заповнення сторінок створюваного сайту контентом і тестування на реальних даних?
9. В чому полягає запуск і супровід програмного продукту?
10. Коли замовлений сайт переносять на сервер замовника?
11. На яких етапах виконується тестування створюваного сайту (програмного продукту)?

Лабораторне завдання

1. Використовуючи теоретичні відомості цієї роботи та рекомендовану літературу за відповідною тематикою, ознайомитись з основними принципами планування робіт проєкту та керування проєктами в цілому й зокрема із системою ProjectLibre.

2. Відкрити на комп'ютері програму ProjectLibre. За відсутності безкоштовно встановити її з офіційного сайту https://sourceforge.net/projects/projectlibre/ (реєстрації не потребує). Ознайомитись з інтерфейсом програми та командами.

3. Створити в ProjectLibre новий проєкт з ім'ям відповідно до індивідуального варіанта табл. 9.1.

Таблиця 9.1

Індивідуальні варіанти проєктів для розробки

Номер варіанта	**Назви проєктів**
1	Розробка вебсайту магазину мобільних телефонів
2	Розробка вебсайту деканату
3	Розробка вебсайту наукового журналу
4	Розробка вебсайту школи
5	Розробка вебсайту поліклініки
6	Розробка вебсайту бібліотеки
7	Розробка вебсайту магазину одягу
8	Розробка вебсайту турагенції
9	Розробка вебсайту спортклубу
10	Розробка вебсайту готелю
11	Розробка вебсайту для переказу коштів
12	Розробка вебсайту поштової служби
13	Розробка вебсайту аптеки
14	Розробка вебсайту університету
15	Розробка вебсайту прокату автомобілів
16	Розробка вебсайту хостелу
17	Розробка вебсайту мережі продовольчих крамниць
18	Розробка вебсайту ресторану
19	Розробка вебсайту медичного центру
20	Розробка вебсайту піцерії
21	Розробка вебсайту стоматологічної клініки
22	Розробка вебсайту кінотеатру
23	Розробка вебсайту оптики
24	Розробка вебсайту мережі кафе
25	Розробка вебсайту ательє
26	Розробка вебсайту ІТ-компанії
27	Розробка вебсайту мережі кав’ярень
28	Розробка вебсайту дитячого садка
29	Розробка вебсайту церкви
30	Розробка вебсайту офтальмологічного центру

4. Задати інформацію про проєкт:

- *Менеджер* – своє прізвище;
- *Початок* – дату початку семестру. Вибрати спосіб планування – *вперед від дати початку проєкту*;

- *Звітна дата* – через чотири місяці від початку над проєктом;
- *Примітки* – додаткові дані про проєкт, наприклад, поточну дату створення проєкту в ProjectLibre, автора цього проєкту (лабораторної роботи), прізвище викладача тощо.

Project Information

General | Statistics | Notes

Назва: Сайт Приклад ЛБ
Менеджер: Трофименко О.Г.
Початок: 01.09.22 8:00
Поточна дата:
Закінчення: 27.12.23 17:00
Звітна дата: 31.01.23
Планування вперед
Основний календар: П'ятиденка
Пріоритет: 500
Статус проєкту: Планування
Тип проєкту: Розробка продукту
Тип витрат: Немає
Розподіл:
Група:
Чистий приведений дохід: 0
Вигода: 0
Ризик: 0.0

Закрити | Help

5. Перейти до діаграми Ґанта і командою *Задача / Вставити* додати у проєкт список задач та їхні тривалості у днях з угрупуваннями відповідно до тематики проєкту. Приклад приблизного переліку задач наведено у табл. 9.2. Угрупування підзадач виконувати командою *Задача / Відступ* для виділеного набору задач.

Таблиця 9.2

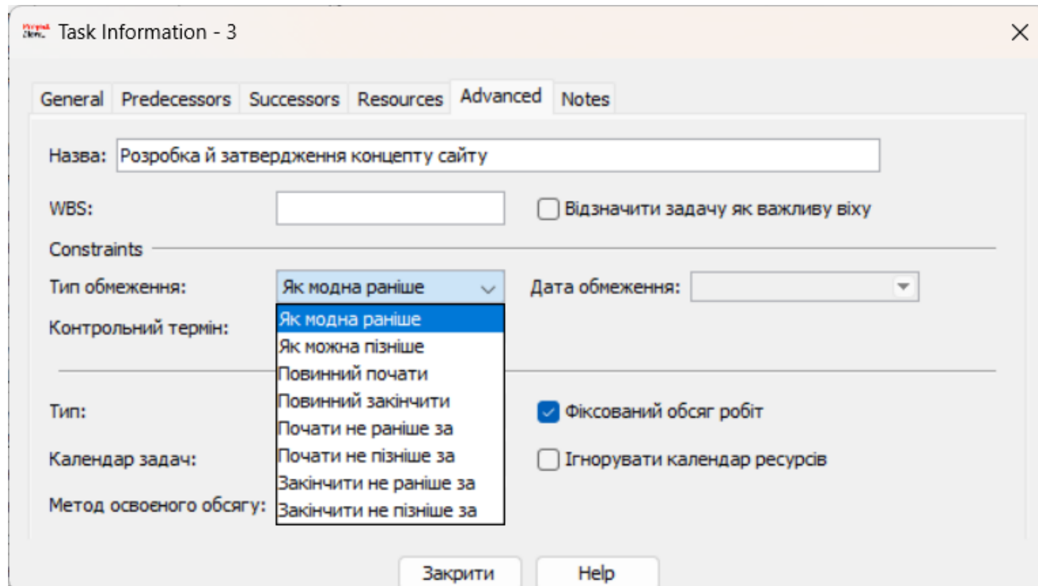
Приклад можливого складу і тривалості робіт проєкту

Назва роботи	**Тривалість**, дні
Аналіз вимог	
Аналіз предметної області, інтерв'ювання клієнта	6
Розробка й затвердження концепту сайту	2
Планування проєкту	3
Складання ТЗ	5
Затвердження ТЗ	2
Завершення аналізу вимог	0
Проєктування	
Розробка дизайну за затвердженим концептом	7
Проєктування архітектури сайту	3
Проєктування БД	3
Завершення проєктування	0

Закінчення табл. 9.2

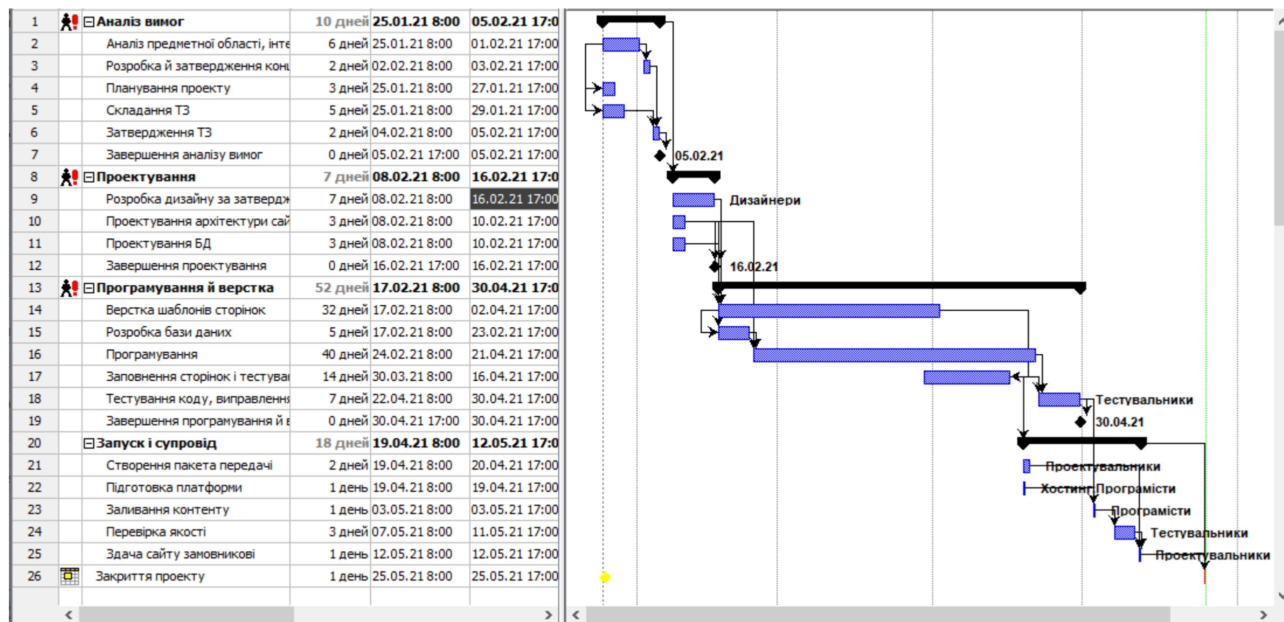
Назва роботи	**Тривалість**, дні
Програмування й верстка	
Верстка шаблонів сторінок	32
Розробка бази даних	5
Програмування	40
Заповнення сторінок і тестування на реальних даних	14
Тестування коду, виправлення помилок	7
Завершення програмування й верстки	0
Запуск і супровід	
Створення пакета передачі	2
Підготовка платформи	1
Заливання контенту	1
Перевірка якості	3
Здача сайту замовникові	1
Закриття проєкту	**1**

6. Налаштувати календар (команда *Файл / Календар*), в якому святкові дні відзначити як неробочі.
7. Повернутись до діаграми Ґанта і задати коректні типи обмежень для всіх задач. Для цього двічі клацнути задачу і вибрати доречний тип обмеження на вкладці *Додатково* діалогового вікна.



8. Перевірити в полі *Початок* дату початку проєктного завдання так, щоб початок проєкту збігся з початком семестру.
9. Визначити по черзі зв'язки між усіма задачами так, щоб, у цьому була логіка відповідно до реальних етапів розробки сайту. Зв’язки можна створити простим проведенням миші від одної задачі до іншої.
10. Визначити загальну тривалість проєкту. Скоригувати зв’язки між задачами так, щоб загальна тривалість усього проєкту не перевищувала узгодже-

ний із замовником строк у чотири місяці. Для цього треба внести у план проєкту необхідні затримки й випередження виконання робіт.



1	Аналіз вимог	10 дней	25.01.21 8:00	05.02.21 17:0
2	Аналіз предметної області, інте	6 дней	25.01.21 8:00	01.02.21 17:00
3	Розробка й затвердження конц	2 дней	02.02.21 8:00	03.02.21 17:00
4	Планування проекту	3 дней	25.01.21 8:00	27.01.21 17:00
5	Складання ТЗ	5 дней	25.01.21 8:00	29.01.21 17:00
6	Затвердження ТЗ	2 дней	04.02.21 8:00	05.02.21 17:00
7	Завершення аналізу вимог	0 дней	05.02.21 17:00	05.02.21 17:00
8	Проектування	7 дней	08.02.21 8:00	16.02.21 17:0
9	Розробка дизайну за затвердж	7 дней	08.02.21 8:00	16.02.21 17:00
10	Проектування архітектури сай	3 дней	08.02.21 8:00	10.02.21 17:00
11	Проектування БД	3 дней	08.02.21 8:00	10.02.21 17:00
12	Завершення проектування	0 дней	16.02.21 17:00	16.02.21 17:00
13	Програмування й верстка	52 дней	17.02.21 8:00	30.04.21 17:0
14	Верстка шаблонів сторінок	32 дней	17.02.21 8:00	02.04.21 17:00
15	Розробка бази даних	5 дней	17.02.21 8:00	23.02.21 17:00
16	Програмування	40 дней	24.02.21 8:00	21.04.21 17:00
17	Заповнення сторінок і тестува	14 дней	30.03.21 8:00	16.04.21 17:00
18	Тестування коду, виправлення	7 дней	22.04.21 8:00	30.04.21 17:00
19	Завершення програмування й в	0 дней	30.04.21 17:00	30.04.21 17:00
20	Запуск і супровід	18 дней	19.04.21 8:00	12.05.21 17:0
21	Створення пакета передачі	2 дней	19.04.21 8:00	20.04.21 17:00
22	Підготовка платформи	1 день	19.04.21 8:00	19.04.21 17:00
23	Заливання контенту	1 день	03.05.21 8:00	03.05.21 17:00
24	Перевірка якості	3 дней	07.05.21 8:00	11.05.21 17:00
25	Здача сайту замовникові	1 день	12.05.21 8:00	12.05.21 17:00
26	Закриття проекту	1 день	25.05.21 8:00	25.05.21 17:00

11. Зберегти скріншот з ім’ям *лб_Прізвище_Ґант*. Для того, щоб коректно зафіксувати на скріншоті все зображення схеми (діаграми), варто скористатись кнопками *Віддалити* та *Приблизити*.

12. Підготувати звіт з лабораторної роботи.

Лабораторна робота 10
Планування ресурсів проєкту

Мета роботи: набути практичних навичок роботи розподілу часових і матеріальних ресурсів робіт під час планування робіт проєкту.

Теоретичні відомості

Ресурси (Resource) бувають двох типів – робота й матеріали.

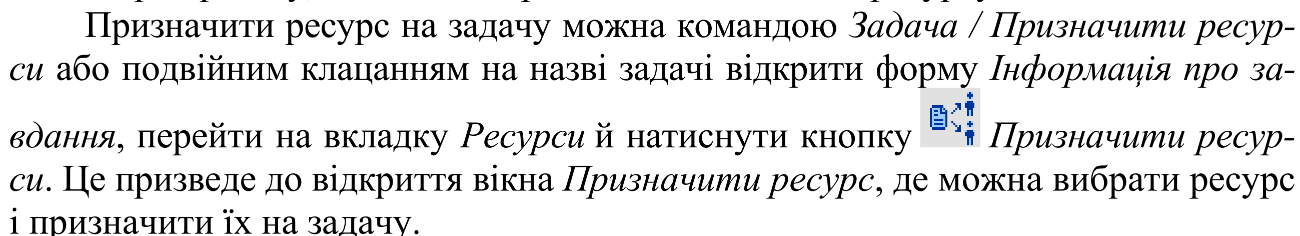
Ресурси типу **Робота** (Work) виконують задачу шляхом витрати часу (виділення часу на його виконання). Звичайно це люди та/або устаткування, які призначені виконувати роботу в рамках проєкту.

Ресурси типу **Матеріал** (Material) – це запаси видаткових матеріалів і компонентів, які потрібні для виконання проєкту. Матеріальні ресурси можна відслідковувати й призначати для задачі.

Для доступу до таблиці ресурсів треба виконати команду *Ресурс / Ресурси* або *Перегляд / Ресурси.* Далі почергово в комірки колонки *Назва* вписати назви ресурсів. Назва ресурсу не може містити символів косої риси «/», квадратних дужок «[» і «]», а також ком «**,**». Для кожного з ресурсів вибирають тип ресурсу – *Робота* або *Матеріал*.

Для матеріалів задають одиниці виміру. У стовпці *Максимальне використання* (*одиниці*) показують максимальне завантаження, яке можливе для ресурсу при виконанні будь-яких задач у певний період часу. Усталено використовується формат відсотків. Наприклад, якщо зазначено 100%, це означає, що цей ресурс працюватиме всі вісім годин у звичайний робочий день.

При клацанні в таблиці ресурсів по назві ресурсу буде виведене вікно *Інформація про ресурс.* На вкладці *Загальне* цього вікна можна вказувати різні базові календарі, наприклад, індивідуальний календар, відмінний від загального календаря проєкту, і змінювати робочий час кожного ресурсу.

Призначити ресурс на задачу можна командою *Задача / Призначити ресурси* або подвійним клацанням на назві задачі відкрити форму *Інформація про завдання*, перейти на вкладку *Ресурси* й натиснути кнопку  *Призначити ресурси*. Це призведе до відкриття вікна *Призначити ресурс*, де можна вибрати ресурс і призначити їх на задачу.

Контрольні запитання для самоконтролю

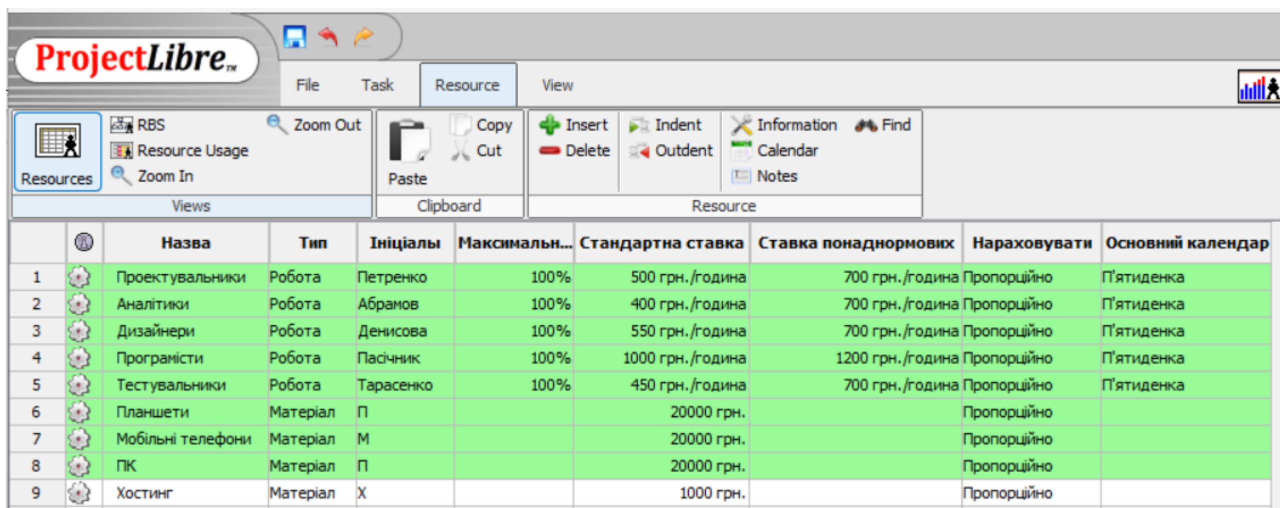
1. Що таке ресурси і яких типів вони бувають?
2. Для чого необхідні календарі ресурсів?
3. Як визначається вартість проєкту?

4. Як визначаються витрати на задачу?
5. Що таке бюджет проєкту й що таке витрата бюджету?
6. Що таке фіксовані витрати на задачу?
7. Що таке метод нарахування при розрахунках витрат?
8. Що таке контур або профіль завантаження?
9. Як розподіляти завантаження ресурсів у рамках призначення за допомогою профілів?

Лабораторне завдання

1. Відкрити в ProjectLibre проєкт своєї попередньої лабораторної роботи.

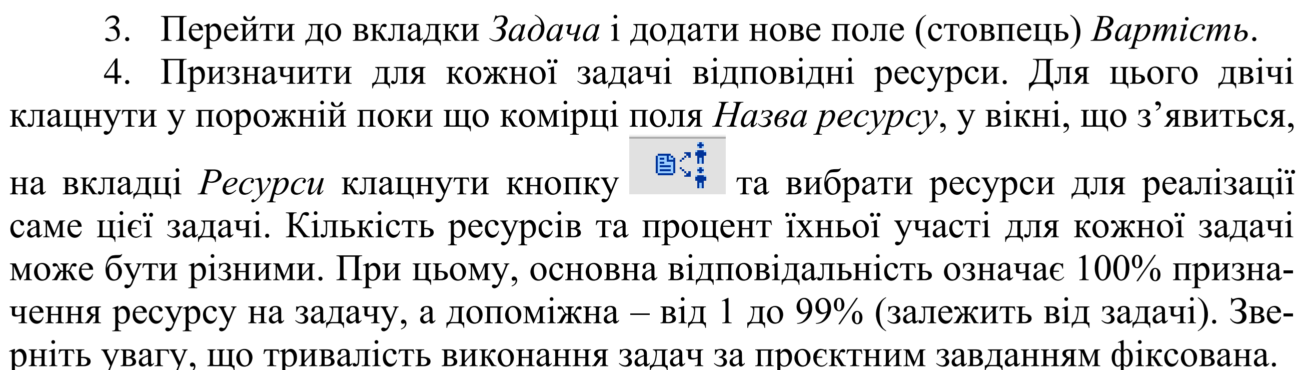
2. Виконати команду *Ресурс / Ресурси* і внести дані про робочі та матеріальні ресурси для розробки проєкту. Набір можливих ресурсів має відрізнятися від наведених далі на рисунку:

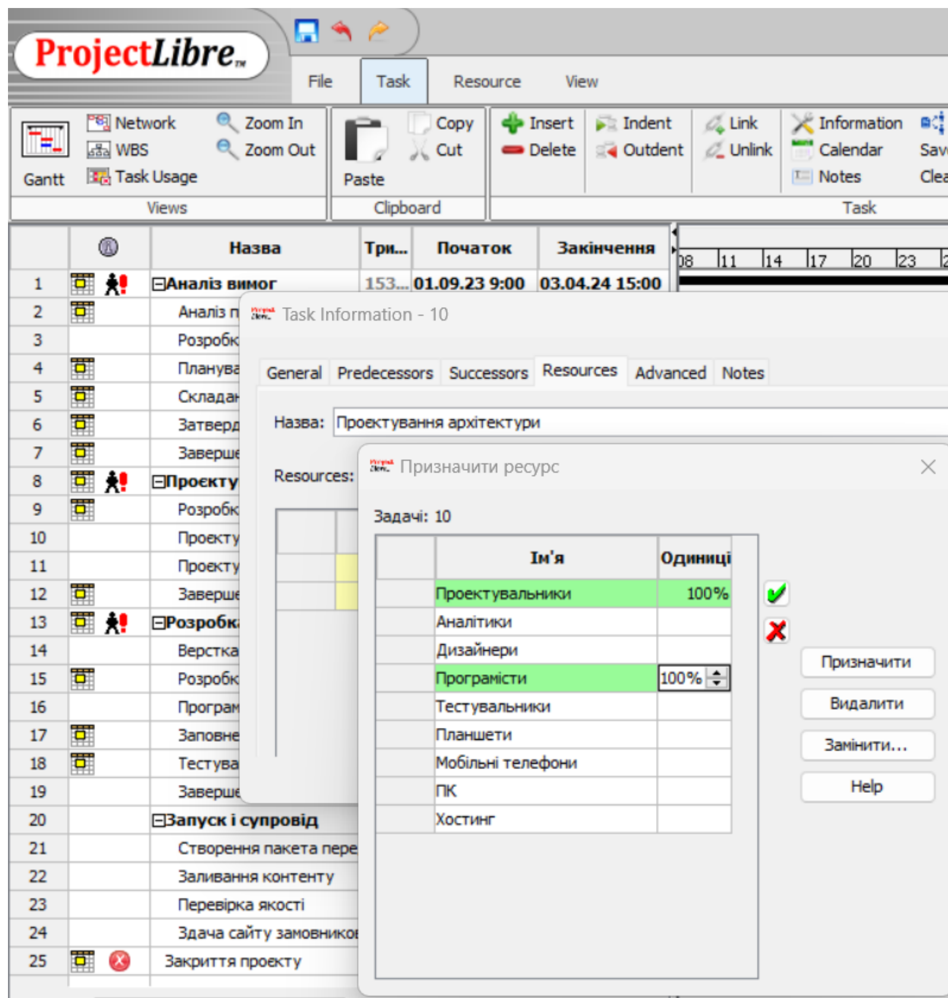


		Назва	Тип	Ініціали	Максимальн...	Стандартна ставка	Ставка понаднормових	Нараховувати	Основний календар
1		Проектувальники	Робота	Петренко	100%	500 грн./година	700 грн./година	Пропорційно	П'ятиденка
2		Аналітики	Робота	Абрамов	100%	400 грн./година	700 грн./година	Пропорційно	П'ятиденка
3		Дизайнери	Робота	Денисова	100%	550 грн./година	700 грн./година	Пропорційно	П'ятиденка
4		Програмісти	Робота	Пасічник	100%	1000 грн./година	1200 грн./година	Пропорційно	П'ятиденка
5		Тестувальники	Робота	Тарасенко	100%	450 грн./година	700 грн./година	Пропорційно	П'ятиденка
6		Планшети	Матеріал	П		20000 грн.		Пропорційно	
7		Мобільні телефони	Матеріал	М		20000 грн.		Пропорційно	
8		ПК	Матеріал	П		20000 грн.		Пропорційно	
9		Хостинг	Матеріал	Х		1000 грн.		Пропорційно	

Стандартні ставки для трудових ресурсів проєкту задати відповідно до проєктного завдання. Понаднормові ставки встановити в 1,5-2 рази більше стандартних.

3. Перейти до вкладки *Задача* і додати нове поле (стовпець) *Вартість*.

4. Призначити для кожної задачі відповідні ресурси. Для цього двічі клацнути у порожній поки що комірці поля *Назва ресурсу*, у вікні, що з'явиться, на вкладці *Ресурси* клацнути кнопку  та вибрати ресурси для реалізації саме цієї задачі. Кількість ресурсів та процент їхньої участі для кожної задачі може бути різними. При цьому, основна відповідальність означає 100% призначення ресурсу на задачу, а допоміжна – від 1 до 99% (залежить від задачі). Зверніть увагу, що тривалість виконання задач за проєктним завданням фіксована.



5. Побачити вартість етапів робіт і загальну вартість проєкту можна, якщо перейти до перегляду списку розроблених проєктів. Можна порівняти отриману вартість проєкту із затвердженим бюджетом. Розбіжність із затвердженим бюджетом у даній роботі є можливою.

6. Відповідними командами на вкладці *Задача* переглянути, проаналізувати розроблений план проєкту у різних режимах та зробити відповідні скріншоти:

- діаграма Ґанта (команда *Gantt*);
- графічне подання ієрархічної структури робіт (команда *WBS*[19]);
- мережевий графік (команда *Network*).

Для того, щоб коректно зафіксувати на скріншоті все зображення схеми (діаграми), варто скористатись кнопками *Віддалити* та *Приблизити*.

Зберегти скріншоти з іменами *лб_Прізвище_Ґант*, *лб_Прізвище_WBS*, *лб_Прізвище_Network.*

7. Підготувати такі звіти:

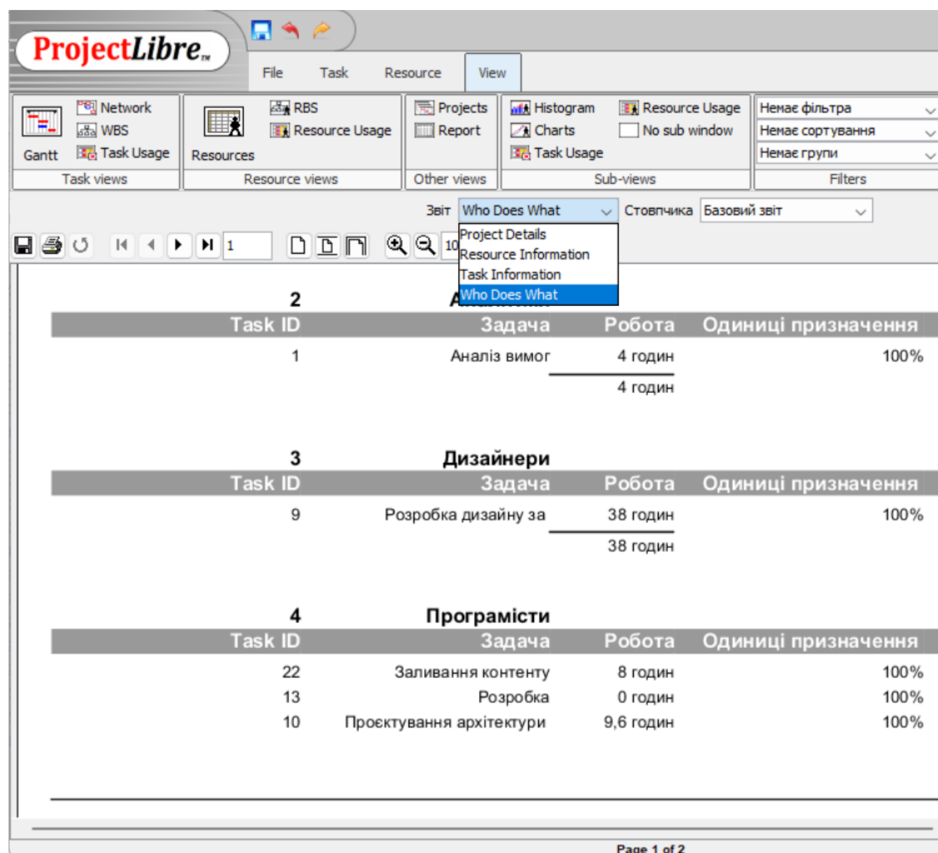
- загальні відомості про проєкт (*Project Details*);
- список ресурсів (*Resource Information*);
- задачі проєкту (*Task Information*);

[19] Work Breakdown Structure (WBS) – ієрархічна структура робіт показує декомпозицію робіт.

– хто що виконує, використання ресурсів (*Who Does What*).

– рух коштів по проєкту (*Task Information / Cost*).

Для цього варто скористатись командою *Перегляд / Звіт*. Зберегти кожен з чотирьох звітів у форматі PDF з відповідними іменами (*лб_Прізвище_1*, ..., *лб_Прізвище_5*), скориставшись кнопкою у вигляді дискети у верхньому лівому куті звіту.



8. Створити копію отриманого плану проєкту й провести серію експериментів для вивчення властивостей призначень:

а) визначити персональний календар для одного ресурсу проєкту;

б) нехай один з виконавців не працює над проєктом по п'ятницях в одному з місяців, а інший в ці дні працює половину робочого дня;

в) розподілити завантаження ресурсів під час виконання завдань за допомогою профілів, визначити різні профілі завантаження для деяких задач;

г) провести дослідження щодо впливу властивостей задач різних типів на параметри призначення;

д) реалізувати перерву у виконанні однієї із задач;

е) внести зміни ставки працівника з деякої певної дати до завершення проєкту, визначити різні норми витрат для ресурсу, призначеного на кілька різних задач.

Зробити звіти результатів цих експериментів і проаналізувати отримані результати.

9. Підготувати звіт з лабораторної роботи.

Лабораторна робота 11
Декомпозиція програм

Мета роботи: оволодіти методикою створення ментальних карт.

Теоретичні відомості

1. Навіщо потрібна декомпозиція програм

Як було зазначено раніше, базова концепція проєктування ПЗ – це методологія проєктування архітектури за допомогою різних методів (об'єктного, компонентного й ін.), процеси ЖЦ (стандарт ISO/IEC 12207) і техніки – декомпозиція, абстракція, інкапсуляція й ін.

На початкових стадіях проєктування предметна область декомпозується на окремі об'єкти (при об’єктно-орієнтованому проєктуванні) або на компоненти (при компонентному проєктуванні). Для подання архітектури програмного забезпечення вибираються відповідні артефакти (нотації, діаграми, блок-схеми і методи).

Отже ключовими питаннями проєктування є **декомпозиція програм** на функціональні компоненти для незалежного й одночасного їхнього виконання, розподіл компонентів у середовищі функціонування та їх взаємодія між собою, забезпечення якості і живучості системи тощо.

Декомпозиція, як метод поділу, дозволяє розглядати будь-яку велику систему як складну, що складається з окремих взаємопов'язаних підсистем, які також можуть складатися з менших підсистем.

Декомпозиція потрібна для:
- розуміння структури системи;
- виявлення характеристик та особливостей складових системи;

– виявлення взаємозв'язків між частинами системи.

Ментальні карти, інтелект-карти, карти думок або карти пам'яті, розуму (англ. Mind map) – спосіб візуальної структуризації, тобто зображення процесу загального системного мислення за допомогою схем. Вони використовуються для створення, візуалізації, структуризації і класифікації ідей, а також як засіб для навчання, організації, вирішення завдань, ухвалення рішень, при написанні статей тощо.

Ментальні карти реалізуються у вигляді діаграм, на яких зображені слова, ідеї, завдання або інші поняття, зв'язані гілками, що відходять від центрального поняття або ідеї. В основі цієї техніки лежить принцип «радіантного мислення», що відноситься до асоціативних розумових процесів, відправною точкою або точкою дотику яких є центральний об'єкт.

2. Що таке MindMap

Одним з найбільш ефективних методів планування, конспектування, підготовки презентацій і написання текстів є майндмеппінг (mind mapping), або створення ментальних карт (Mind Maps). Саме цей метод сьогодні широко застосовується студентами й викладачами у всьому світі, а також співробітниками найбільших компаній і корпорацій, таких як Walt Disney, IBM, Boeing, De Beers, Apple та ін. Чому ж майндмеппінг користується такою популярністю?

Ментальні карти – це графічне подання інформації в зручній для людського сприйняття формі: логічних і асоціативних деревоподібних схем. Ключова ідея, тема або проблематика – в центрі, і від неї розходяться гілки – підтеми, поняття, ідеї тощо. Пов’язані інформаційні блоки поєднуються однаковим кольором чи фоном. Оскільки ментальні карти відображають всю картину в цілому, це дозволяє встановити всі взаємозв'язки між об'єктами. Структура і логіка даних стають легкими для розуміння і запам'ятовування.

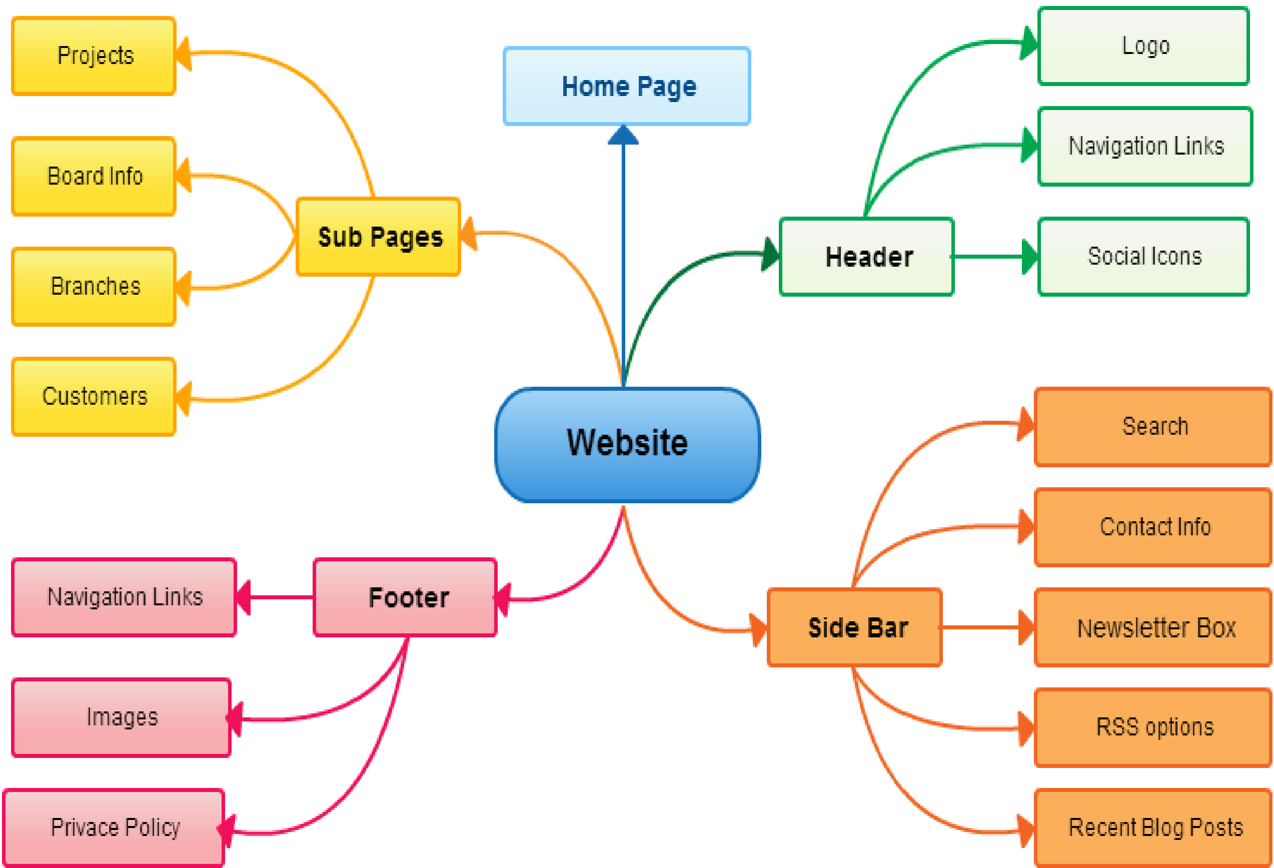
Ментальну карту можна створити як на комп'ютері, за допомогою спеціального програмного забезпечення або онлайн сервісів, так і намалювати від руки на аркуші. Зауважимо, що читається карта за годинниковою стрілкою, починаючи з правого верхнього кута.

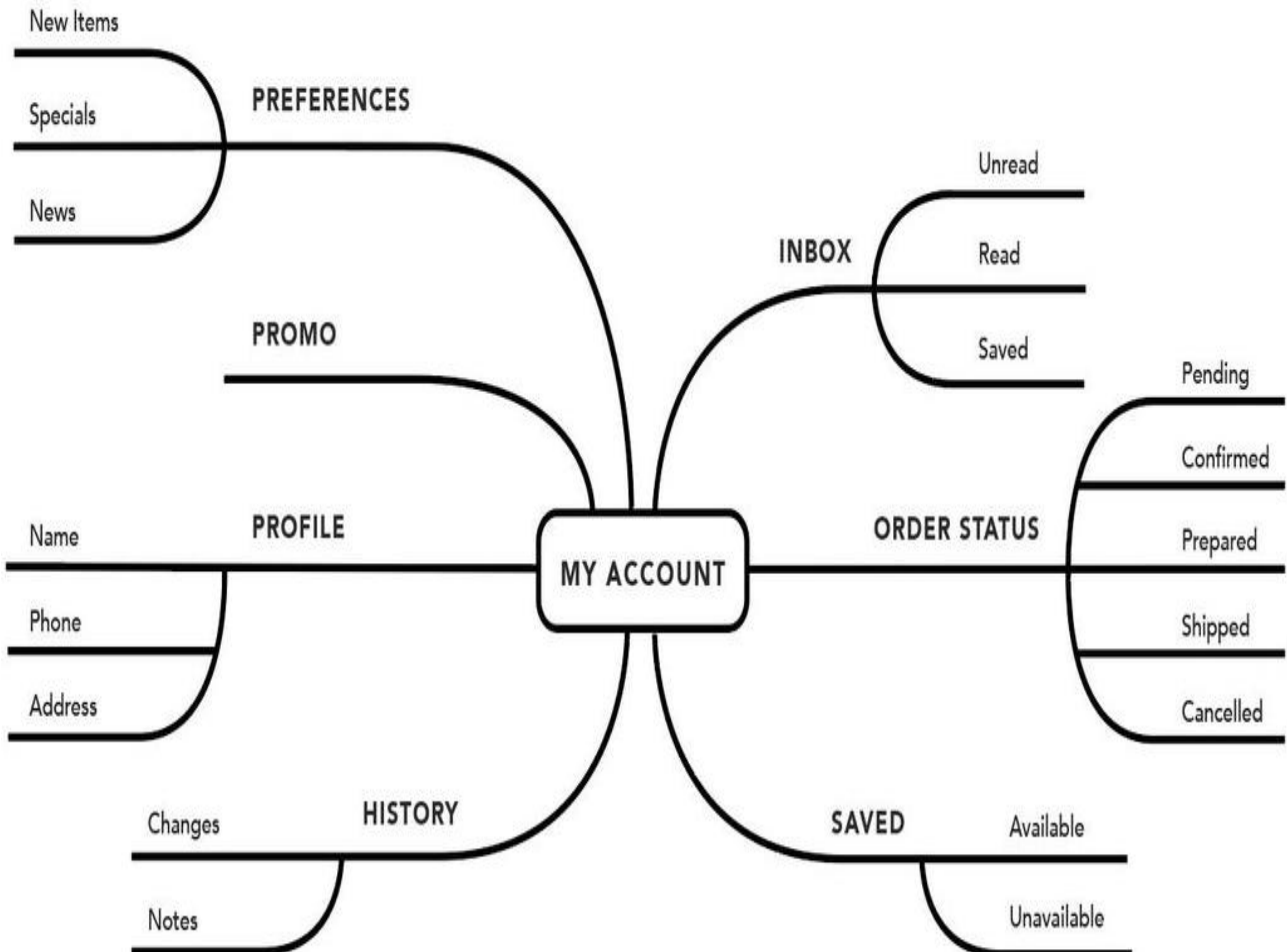
Отже, Mindmapping (малювання деревоподібних карток або ментальних карток) міцно увійшов у життя багатьох людей саме з появою відповідного ПЗ. Використовуючи Mindmap, роблять техзавдання (ТЗ), консультанти-аналітики роблять проєкти, дизайнери вигадують концепції, тренери роблять презентації, менеджери складають плани тощо.

Ба більше, **MindMap** (або альтернативна назва **інтелект-карта**, **ментальна карта**) є оптимальним інструментом для тестувальника ПЗ, оскільки він поєднує в собі плюси тест-кейсів і чек-листів. Технологія MindMap задіює ієрархічне мислення і є візуалізованою, що набагато краще сприймається, порівняно з інформацією у формі текстів, списків і таблиць. Візуалізоване та ієрархічно структуроване подання структури компонентів та об’єктів програми є набагато природнішим і простіше засвоюється свідомістю, завдяки асоціаціям.

Безперечна користь методики майндмеппінгу полягає в тому, що вона допомагає структурувати і детально розібрати будь-яку інформацію, ідею і завдання. Допоможе методика і в разі, коли потрібно швидко опрацювати великі об’єми інформації; провести мозковий штурм, вирішити професійні завдання і навіть особисті проблеми.

Ось приклади інтелект-карт:


Home Page
Projects
Board Info
Sub Pages
Branches
Customers
Website
Header
Logo
Navigation Links
Social Icons
Search
Contact Info
Side Bar
Newsletter Box
RSS options
Recent Blog Posts
Footer
Navigation Links
Images
Privace Policy


MY ACCOUNT
PREFERENCES
New Items
Specials
News
PROMO
PROFILE
Name
Phone
Address
HISTORY
Changes
Notes
INBOX
Unread
Read
Saved
ORDER STATUS
Pending
Confirmed
Prepared
Shipped
Cancelled
SAVED
Available
Unavailable

3. MindMap в тестуванні

Отже, MindMap або інтелект карта – це інструмент для візуального подання інформації, що допомагає ефективно її структурувати, простіше для розуміння людським мозком, ніж текст, і тому простіше для застосування у роботі.

Переваги MindMap для тестування:

– *наочність і швидкість зчитування інформації*: головною перевагою MindMap для тестувальника є наочне бачення тестованого продукту, його функцій і залежностей між собою. Видно всю "картину" крок за кроком;

– *зручна оцінка можливих взаємозв'язків*: правка коду однієї гілки наочно показує яких дочірніх гілок торкнуться зміни – їх потрібно перевіряти насамперед;

– *чудова альтернатива документації*: таку карту добре демонструвати новим співробітникам як альтернативу чи доповнення до документації. Тестувальнику, який знайомий з проєктом, простіше увійти в курс справи, не перечитуючи постановки та завдання. Команда онлайн може працювати над створенням набору пула категорій для своїх User Story на одній дошці;

– *повне охоплення картини тестування і зручна статистика*: розташування всієї необхідної інформації одному місці. Кожній гілці можна надавати кольори (наприклад: при перевірці, крок за кроком відзначати, що перевірено, і навпаки, якщо десь баг, то відразу при вході на дошку це буде видно). З MindMap можна сказати з упевненістю, які саме компоненти продукту вже перевірялися, а які ще потребують перевірки;

– *можливість за допомогою гіперпосилання пов'язати постановку завдання з інтелект-картою*: назву будь-якої категорії можна зробити у вигляді посилання. При натисканні на пункт плану відбувається перехід на потрібну сторінку. Наприклад, зручно зробити посилання на User Story у Bug Tracker, тоді можна одразу відкрити і завести баг, якщо щось знайшли на екрані конкретної US;

– *легко підтримувати*: з виходом нових функцій її нескладно доповнити і знову відстежити взаємозв'язки нових частин програми, можливо навіть виявити, де продукт можна зробити простіше і зрозуміліше користувачеві;

– *надійність тестування:* можна дуже довго тестувати програму, але так і не переконатися, що перевірено дійсно все. Постійно актуалізована MindMap, що охоплює весь функціонал, дозволить мінімізувати ризики пропуску помилок.

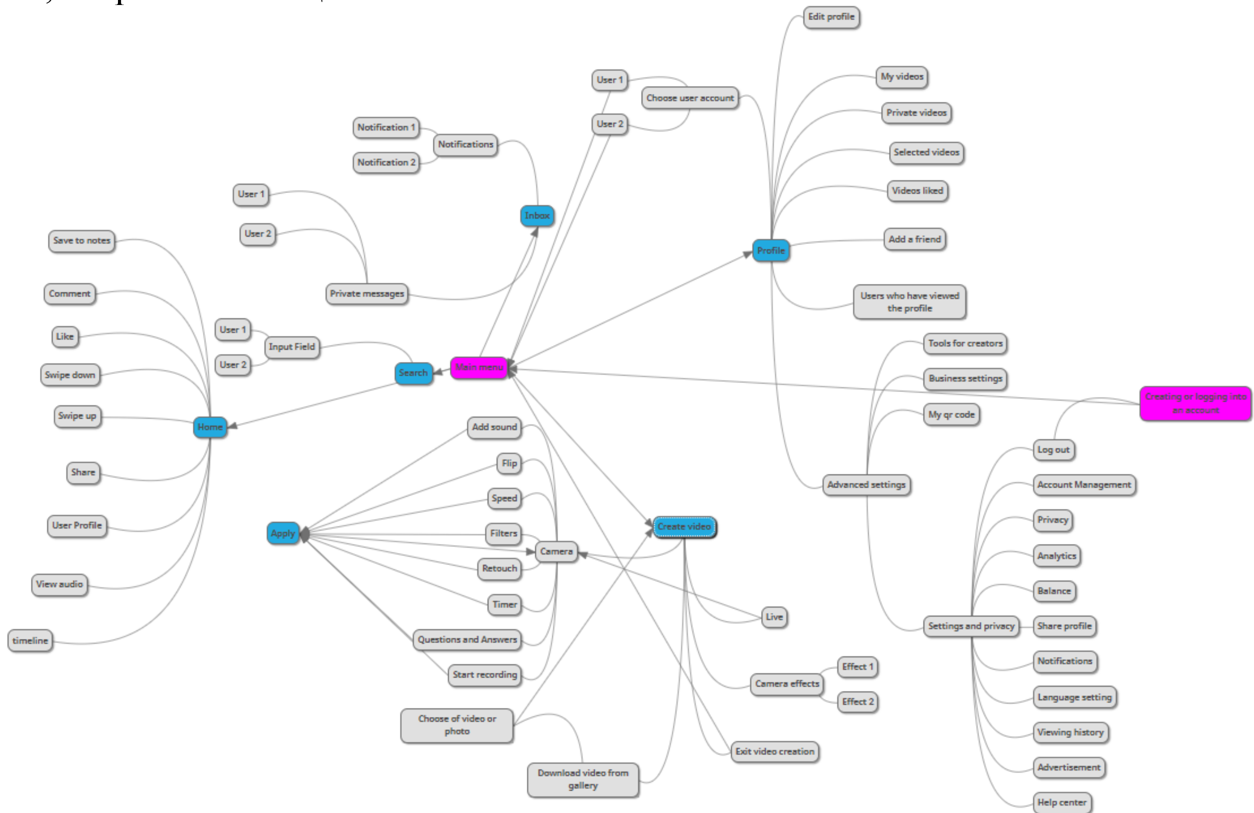
За допомогою MindMap можна зобразити:

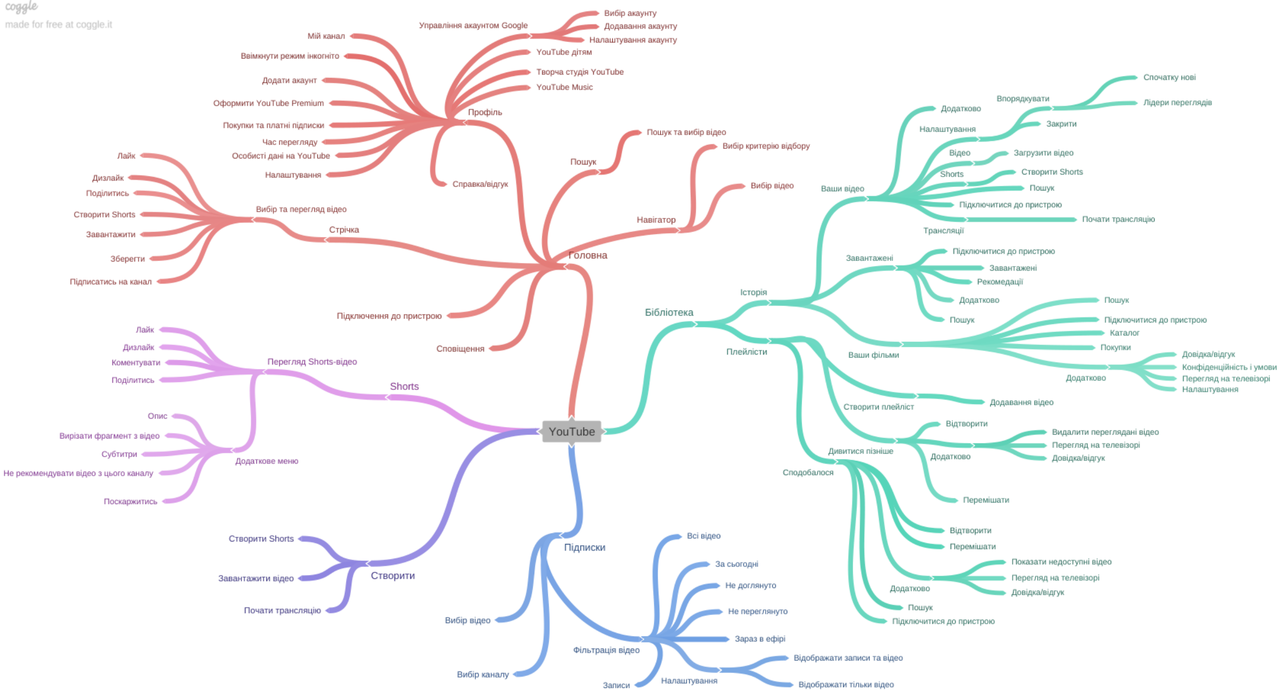
– функціонал програми на різних рівнях;
– пріоритет функціоналу;
– залежності у застосунку тощо.

Декомпозиція. Для визначення функцій та/або частин програми варто розділяти функції за видами сутності і за діями, які з ними можна зробити. Наприклад, для схеми онлайн-магазину *сутностями* будуть: «Товар», «Каталог», «Кошик», «Аккаунт». *Діями* для онлайн-магазину будуть: «Знайти товар», «Переглянути товар», «Придбати товар», «Поставити оцінку», «Створити акаунт», «Увійти до облікового запису»[20]. Отже, використовуючи правила декомпозиції,

[20] Mind Map у тестуванні. URL: https://habr.com/ru/post/515990/

розкладення на дії та сутності наочно інформує про те, що клієнт може зробити, що йому для цього потрібно і з чим він взаємодіятиме. Так можна описати застосунок до найдрібніших деталей, що дуже знадобиться у тестуванні. На прикладі гілки «Товар», відгалуженнями будуть: назва, ціна, розмір, кількість, опис, зображення тощо.





Взаємозв'язки на MindMap можна зобразити за допомогою стрілок, що йдуть від одного блоку до іншого. Прикладом може бути взаємозв'язок ціни одиниці товару в каталозі, у сортуванні результатів пошуку, на сторінці товару, у кошику та логіка суми всіх товарів до оплати. Це допоможе не забути протес-

тувати пов'язані між собою функції, побачити, на які ще компоненти могли вплинути баги або правки тієї чи іншої частини програми[21].

Інтелект-карта є основою написання тестових випадків. Розставивши пріоритети, можна почати писати тестові випадки, починаючи з високопріоритетних функцій. Завдяки взаємозв'язкам точно буде відомо, які ще частини програми міг торкнутися знайдений "баг" в ході перевірки. Вказавши на це у складанні "баг-репорту", тим самим можна покращити якість і швидкість розробки.

Завдання QA спеціаліста полягає у відстежуванні покриття тестами. Почавши писати тест-кейси на високопріоритетні перевірки, використовуючи взаємозв'язки, варто рухатися по карті і просто відзначати осередки з функціями, які були задіяні в тест-кейсах, іншим кольором або будь-яким зручним способом. Наприклад, можна ставити позначку "Thumbs Up" і по мірі написання тест-кейсів буде ясно видно, які функції вже покриті, а які ні.

Розберемо на прикладі. Припустимо, що в країні змінився закон про розрахунок ПДВ за представленою на сайті категорією товарів. Бізнес вирішує внести зміни у формування ціни, Product Owner вже створив відповідні завдання для розробників і вони вже повною мірою оновлюють код. А QA спеціаліст можете почати писати тестову документацію для покриття цих змін тестами. Тут і приходить на допомогу MindMap. Завдяки тому, що наочно видно, в яких частинах застосунка фігурує сутність “ціна”, через взаємозв'язки з іншими частинами до-

[21] Mind Map у тестуванні - або легкий спосіб тестувати складні програми. URL: https://habr.com/ru/post/515990/

датка та діями, які можна зробити з цією сутністю, ясно видно, що потрібно протестувати.

Отже, дослідження програми та її декомпозиція на сутності та дії з розставленням пріоритетів допоможе у написані тестових випадків, оскільки відомо що тестувати і як нічого не забути.

Звичайно, залежно від складності продукту, складання та підтримка такої картки може зайняти багато часу, але в майбутньому, це заощадить більше часу і зробить процес тестування простіше, зрозуміліше і від того приємніше.

Додатковий функціонал

Щоб описати пункт плану, можна використовувати систему **нотаток**. Додавання нотаток зручне для зберігання «паролів і явок», списку дій для перевірки потрібного пункту (не плутати з тест-кейсом).

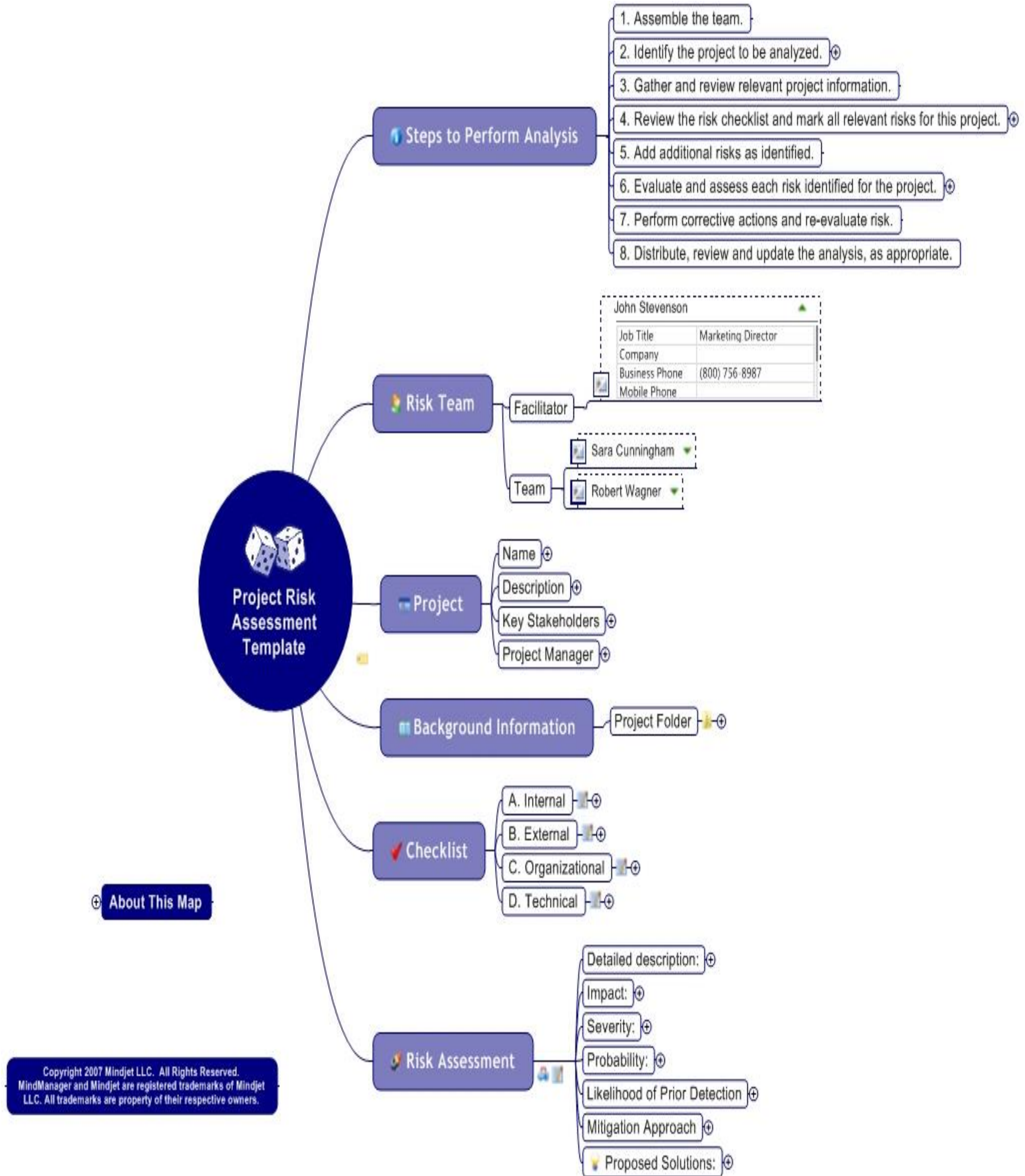
Функціонал **додавання фотографії** до інтелект-карти зручний у тих випадках, коли потрібно візуалізувати зміни в конкретній області. Наприклад, зміна логотипу з одного на інший (зображення нового логотипу доцільно прикріпити до пункту карти).

У тих випадках, коли сутність надто складна для опису в «нотатці» або з якихось причин її неможливо описати, зручно використовувати **гіперпосилання** та пов'язати постановку завдання з інтелект-картою. При натисканні на пункт плану відбувається перехід на потрібну сторінку.

Ведення інформації – це певний звіт про хід тестування для менеджера проєкту. Під час тесту в інтелект-карті можна вказувати провалені сценарії, відзначати час та прогрес виконання тестування. Інтелект-карту можна надати менеджеру проєкту, щоб донести до нього результати тестування. Можна вважати MindMap зворотним зв'язком і навіть якогось роду захистом тестувальника від свавілля ПМ та аналітика, оскільки, якщо щось працює поза цією інтелект-картою, то тестувальник не винен, позаяк. картка не передбачає такого сценарію.

Ведення версійності. Щоб ідентифікувати нову функціональність інтелект-карти, доцільно вказати версійність. Тобто додати пункт карти (по суті є функціоналом) та вказати версію ПЗ, куди увійшов цей функціонал. При перевірці нових фіч зробити фільтр за версією, і на карті будуть підсвічені тільки ті пункти, які стосуються вибраної версії.

Отже, спочатку, поки проєкт ще невеликий, аналітик спільно з фахівцем з тестування створюють першу просту інтелект-карту. Далі, перед випуском нового релізу з новим функціоналом, аналітик актуалізує інтелект-карту, проставляє на ній версію та передає у тестування. Тестувальник виконує свою роботу, зазначає виконані, невиконані, провалені пункти, відзначає час та прогрес виконання – тим самим отримує звіт про тестування, який може передати керівнику підрозділу чи проєкту. У новому релізі аналітик знову актуалізує інтелект-карту та передає копію в тестування і все повторюється за вказаною схемою. А якщо вивантажити MindMap в Excel, можна отримати зручний звіт про тестування, який не соромно показати замовнику.



Контрольні запитання для самоконтролю

1. Навіщо потрібна декомпозиція програм?
2. Навіщо потрібні інтелект-картки?
2. Чому інтелект-карти такі популярні?
3. В якому вигляді складають інтелект-картки??
4. Які переваги MindMap для тестування?
5. Які правила складання MindMap?

Лабораторне завдання

1. Використовуючи лекційний та методичний матеріал за відповідною тематикою, ознайомитись з основними принципами декомпозиції програм в цілому й зокрема засобами MindMap.

2. За допомогою MindMap (https://drive.mindmup.com/) або Coggle (http://coggle.it/) виконати декомпозицію якогось програмного мобільного застосунка, який є на вашому телефоні, або довільного сайту. Додатково узгодити вибір програми для композиції з викладачем для усунення дублювання варіантів. Декомпозицію виконати по екранах і функціональностях.

3. Дати відповіді на контрольні запитання.

4. Підготувати звіт з лабораторної роботи.

Самостійна робота 6
Класифікація видів та напрямків тестування

Теоретичні відомості

Тестування можна класифікувати за багатьма ознаками, а глибоке розуміння кожного різновиду в класифікації потребує певного досвіду[22].

1. По запуску коду виконання тестування ділиться на:

- статичне тестування;
- динамічне тестування.

Статичне тестування виконується без запуску ПЗ. Тестування проводять розробники шляхом аналізу програмного коду (code review) або скомпільованого коду. Аналіз може проводитись як вручну, так і за допомогою спеціальних інструментів. Мета аналізу – раннє виявлення помилок та потенційних проблем у продукті. В рамках такого тестування виявляються помилки, пов'язані з ефективністю та правильністю програмних конструкцій. Статичне тестування проводиться і під час перевірки вимог та дизайну за макетами.

Приклади помилок, які можна виявити за допомогою автоматичного статичного тестування:

- витоки ресурсів (витоки пам'яті, незвільнені файлові дескриптори тощо);
- можливість переповнення буфера (buffer overflows);
- ситуація часткової (неповної) обробки помилок.

Приклади статичного тестування:

- тестування вимог;
- статичне тестування коду (code review).

На відміну від статичного, **динамічне тестування** проводиться шляхом запуску продукту та перевірки його функціоналу. Воно здійснюється за допомогою ручного або автоматичного виконання заздалегідь підготовленого набору тестів.

2. За рівнем тестування у процесі розробки:

– **модульне (компонентне) тестування** (unit testing) дозволяє перевірити на коректність окремі модулі вихідного коду програми. Виконується розробниками;

– **інтеграційне тестування** (integrated testing) призначено для перевірки зв'язку між компонентами і взаємодії з різними частинами системи: операційною системою, устаткуванням чи зв'язки між різними системами;

---
[22] Ткач М. QA. KeepSolid.

– **системне тестування** (system testing) перевіряє як функціональні, так і нефункціональні вимоги у системі в цілому. При цьому виявляються дефекти неправильного використання ресурсів системи, не передбачені комбінації даних на рівні користувача, несумісність з оточенням, непередбачені сценарії використання, відсутні або неправильні функціональності, незручності використання тощо;

– **приймальний тест** (acceptance testing) перевіряє відповідність системи вимогам і проводиться з метою визначення відповідності системи приймальним критеріям і прийняття рішення замовником про прийом програми чи то ні.

Іноді ще додаються компонентно-інтеграційне та системно-інтеграційне тестування.

3. За доступом до коду та архітектури програми розрізняють такі методи тестування:

– метод білої скриньки;
– метод чорної скриньки;
– метод сірої скриньки.

Тестування «**чорної скриньки**» базується лише на тестуванні за функціональною специфікацією та вимогами, без урахування внутрішньої структури коду та без доступу до бази даних. Ми знаємо, який має бути результат за певного набору даних, що подаються на вхід. Інтерфейс перевіряють лише на рівні простого користувача. На даний момент така стратегія є найпопулярнішою в IT-компаніях.

Стратегія «**сірої скриньки**» частково задіює доступ, наприклад, до структури баз даних. Її використовує трохи менша кількість тестувальників.

Стратегію «**білої скриньки**» малопоширена, оскільки мало тестувальників здатні аналізувати чужий код і займатися написанням тестів, навіть не запускаючи програму, а використовуючи лише програмний код. Найчастіше це колишні розробники, які пішли в тестування, або тестувальники, що займаються автоматизацією та захоплюються програмуванням. Цей метод використовуватися на додаток до чорного та сірого. Модульне тестування, яке переважно проводять розробники продукту, є прикладом «white box» тестування.

4. За ступенем автоматизації:

– **ручне тестування** – це процес пошуку дефектів у роботі програми, коли тестувальник перевіряє працездатність всіх компонентів, входячи у роль користувача. Фахівці використовують заздалегідь заготовлені плани тестування та тести на основі вимог до ПЗ;

– **автоматизоване тестування** використовує програмні засоби для виконання тестів та перевірки результатів. Застосування автоматизованих тестів дозволяє скоротити час тестування та спростити сам процес. Проте не варто думати, що автоматизація – це просто, і всі тести потрібно зробити автоматизованими. Автоматизоване тестування проводиться різних рівнях. Часто розробники автоматизують тести для перевірки своїх модулів чи зв'язків між ними. Широкої популярності набули автоматизовані тести інтерфейсу користувача, які ему-

люють поведінку користувача, тобто пересувають покажчик миші по екрану, натискають кнопки, пишуть тексти тощо.

І при ручному, і при автоматизованому тестуванні потрібно складати тести та плани тестування. Тести з часом старіють, тому їх треба актуалізувати та підтримувати. Тести для ручного тестування оформляють у формі простого тексту, а автоматизовані – як програмний код. Якщо розробники трохи змінять інтерфейс користувача, то при ручному тестуванні ця зміна не буде критичною і тест, виконуваний людиною, буде успішно пройдено. А автоматизований тест зламається, позаяк. він дуже чутливий до змін.

5. За ступенем важливості тестованих функцій:

– **димове (smoke) тестування** – це мінімальний набір тестів на очевидні помилки. Його успішне проходження говорить про те, що ПЗ можна тестувати глибше і ґрунтовніше. Якщо програмне забезпечення не пройшло smoke-тестування – то перевіряти далі немає сенсу. Раніше термін застосовувався у тестуванні радіоелектроніки. Перше ввімкнення електронного пристрою, що прийшов із виробництва, відбувається на дуже короткий час (менше секунди). Потім інженер перевіряє компоненти щодо перегріву. Якщо перше ввімкнення не виявило перегріву, прилад вмикається знову на триваліший час. Перевірка повторюється. І так декілька разів. Вираз «smoke-test» використовується інженерами в жартівливому сенсі, тому що появи диму, а значить і псування частин пристрою, намагаються уникнути;

– **тестування критичного шляху** – це перевірка найбільш часто використовуваних користувачами функцій ПЗ;

– **розширене тестування** спрямоване на дослідження всієї заявленої у вимогах функціональності, навіть тієї, яка низько проранжована за ступенем важливості. При цьому тут також враховується, яка функціональність є більш важливою, а яка менш важливою. Але за наявності достатньої кількості часу та інших ресурсів тест-кейси цього рівня можуть перевірити найменш пріоритетні вимоги.

6. За принципами роботи з програмою:

– **позитивне тестування** перевіряє функції програмного забезпечення строго за вимогами та інструкціями, тільки з допустимими діями та коректними даними. Наприклад, позитивним тестом для перевірки функції квадратного кореня є введення числа "4" та отримання результату "+2" або "-2", або в полі "Вік" ввести значення "18";

– **негативне тестування** – це перевірка поведінки ПЗ під час введення некоректних даних. Розглядається те, що не передбачено вимогами чи типом даних. Наприклад, при обчисленні квадратного кореня ввести значення «FFA04» – число у шістнадцятковій системі числення. Як на нього відреагує калькулятор? Або перевірити, як програма відреагує на обчислення кореня із символів «!№@. Очікуваним результатом при такому тесті є або число в шістнадцятковому форматі, або помилка, що введене значення не є числом. Негативне тестування перевіряє, що програма не видає дивних помилок або не падає, коли на вхід подають щось незвичайне.

7. За ступенем формалізації:

– **тестування на основі тест-кейсів** – формалізований підхід, в якому тестування проводиться на основі заздалегідь підготовлених тест-кейсів, наборів тест-кейсів та іншої документації. Це найпоширеніший спосіб тестування, який дозволяє досягти максимальної повноти дослідження програми за рахунок суворої систематизації процесу, зручності застосування метрик і широкого набору вироблених за десятиліття та перевірених на практиці рекомендацій;

– **дослідницьке тестування** (exploratory testing) – частково формалізований підхід, в рамках якого тестувальник виконує роботу з програмою за вибраним сценарієм, який, у свою чергу, доопрацьовується в процесі виконання з метою повнішого дослідження програми. Ключовим фактором успіху при виконанні дослідницького тестування є робота за сценарієм, а не виконання розрізнених бездумних операцій. Існує навіть спеціальний сценарний підхід, який називають сесійним тестуванням. Як альтернативи сценаріям при виборі дій з програмою іноді можуть використовуватися чек-листи, і тоді цей вид тестування називають тестуванням **на основі чек-листів**;

– **вільне (інтуїтивне) тестування** (ad hoc testing) – повністю неформалізований підхід, в якому не передбачено використання ні тест-кейсів, ні чек-листів, ні сценаріїв – тестувальник повністю спирається на свій професіоналізм та інтуїцію (experience-based testing) спонтанного виконання з програмою дій, які, на його думку, можуть виявити помилку. Цей вид тестування використовується рідко і виключно як доповнення до повністю або частково формалізованого тестування у випадках, коли для дослідження деякого аспекту поведінки програми поки що немає тест-кейсів.

8. За метою тестування:

Вид тестування сфокусований на конкретну мету тестування, яка може бути перевіркою функції, виконуваної компонентом або системою в цілому. Мета тестування може бути спрямована як на перевірку елементів нефункціонального тестування (надійність, зручність використання), структури, архітектури компонентів або системи в цілому, так і на елементи залежно від змін у системі (перевірка виправлення (фіксу) конкретного дефекту (підтвердження або повторне тестування) (confirmation testing or retesting) або перевірка ненавмисних змін (regression testing). Залежно від таких цілей процес тестування повинен бути організований відповідним чином.

Можна визначити 4 види тестування:

– функціональне тестування;
– нефункціональне тестування;
– структурне тестування;
– тестування змін.

Функціональне тестування

Сьогодні складно недооцінити важливість функціонального тестування, адже саме воно спрямоване на тестування всіх функцій системи для підтвердження, що кожна функція програми працює відповідно до документації.

Елементи функціонального тестування:

– підготовка тестових даних по описаній документації;
– бізнес-вимоги як частина функціонального тестування;
– отримання результатів на основі специфікації;
– проходження тест-кейсів;
– аналіз фактичних та очікуваних результатів.

Функціональне тестування може бути проведене як у суворій відповідності до специфікації, так і на основі бізнес-процесу (тобто відповідно до знань системи).

Нефункціональне тестування

Якщо при функціональному тестуванні відповідають на питання «Чи працює система?», то нефункціональне відповідає питанням: «Як добре працює система?». Нефункціональне тестування спрямоване на перевірку тих аспектів ПЗ, які можуть бути описані в документації, але не належать до конкретних функцій.

Нефункціональне тестування складається з цілої армії підвидів:

1. *Тестування стабільності програми* (Stability/Reliability testing) – виявлення крешів системи під час використання;

2. *Юзабіліті тестування* (Usability testing) – дослідження для визначення зручності використання ПЗ;

3. *Тестування ефективності* (Efficiency testing) – перевірка необхідних обсягів коду та ресурсів QA, використовуваних програмою для виконання окремої функції;

4. *Тестування ремонтопридатності* (Maintainability testing) – цей підвид нефункціонального тестування визначає як легко підтримувати працездатність системи;

5. *Перевірка переносимості (портованості)* (Portability testing) – тестування доступності переносу окремого компонента або всього програмного забезпечення з одного оточення на інше (Windows 8.1 -> Windows 10, Windows -> MacOS);

6. *Приймальний тест* (Compliance/Acceptance testing) – перевірка продукту на відповідність критеріям готовності;

7. *Тестування документації* (Documentation testing) – перевірка всієї створеної в рамках тестування документації (від майстер тест-плану до тест-кейсів);

8. *Тестування витривалості системи* (Endurance testing) – тестування системи при високому навантаженні протягом тривалого часу з метою вивчення її поведінки;

9. *Навантажувальне тестування* (Load testing) зазвичай проводиться з метою визначення поведінки ПЗ під очікуваним рівнем навантаження;

10. *Тестування продуктивності* (Performance testing) – перевірка швидкості роботи ПЗ або його окремих функцій;

11. *Тестування сумісності* (Compatibility testing) – тестування системи під час роботи в різних оточеннях: "залізо", софт частина тощо;

12. *Тестування безпеки* (Security testing) проводиться для відповіді на запитання «Чи є програма безпечною/захищеною чи ні?»;

13. *Об'ємне тестування* (Volume testing) – тестування ПЗ з використанням баз даних певного розміру;

14. *Стрес тестування* (Stress testing) – це тестування в обмежених умовах, наприклад, перевірка поведінки системи (відсутність крешів) за умов нестачі ресурсів ПК (ОЗУ або місця на HDD/SSD дисках);

15. *Тестування швидкості відновлення* (Recovery testing) проводиться з метою визначення швидкості відновлення системи у разі софтверного крешу (крешу програмного забезпечення) або помилки "заліза";

16. *Тестування локалізації, інтернаціоналізація* (Localization testing) – перевірка ПЗ на відповідність мовним, культурним та/або релігійним нормам. Локалізація – перевірка коректного відображення всіх текстів, що є в ПЗ.

Тестування продуктивності ПЗ (навантажувальне, об'ємне, стрес-тестування) проводиться за допомогою спеціальних інструментів, що імітують роботу користувачів, наприклад, Jmeter.

Тестування змін

Після внесення змін (виправлення бага/дефекту) ПЗ має бути перевірено заново для підтвердження, що проблема дійсно вирішена. Нижче пераховані види тестування, які необхідно проводити після встановлення ПЗ:

1. *Димове (smoke) тестування* в області ПЗ розглядається як короткий цикл тестів, що виконуються для підтвердження того, що програма стартує і виконує основні функції. Висновок про працездатність основних функцій робиться на підставі результатів поверхневого тестування найбільш важливих модулів на предмет можливості виконання необхідних завдань. У разі відсутності дефектів димове тестування оголошується пройденим і програма передається для повного циклу тестування, інакше димове тестування оголошується проваленим, і програма йде на доопрацювання. Часто кейси для smoke тестування автоматизовані, і на них, в першу чергу, орієнтуються при прийомі білда на тестування, крім того результати переглядають перед безпосередньою передачею білда в реліз.

Аналогами димового тестування є *Acceptance Testing*, що виконується на функціональному рівні командою тестування, за результатами якого робиться висновок про те, приймається чи ні встановлена версія програмного забезпечення в тестування, експлуатацію або на постачання замовнику.

2. *Регресійне тестування* спрямоване на перевірку змін, зроблених у ПЗ (виправлення дефекту, злиття коду, міграція на іншу операційну систему, базу даних, вебсервер або сервер програми). Таке тестування проводиться, щоб підтвердити, що відповідна функціональність працює. Регресійними можуть бути як функціональні, так і нефункціональні тести. Здебільшого для регресійного тестування використовуються тест-кейси, написані на ранніх стадіях розробки і тестування. Це дає гарантію того, що зміни в новій версії програми не пошкодили наявну функціональність.

Сам по собі термін "регресійне тестування", залежно від контексту використання, може мати різний сенс:

- *регресія багів* (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена;

- *регресія старих багів* (Old bugs regression) – спроба довести, що нещодавня зміна коду або даних зламала виправлення старих помилок, тобто старі баги знову стали відтворюватися;
- *регресія побічного ефекту* (Side effect regression) – спроба довести, що нещодавня зміна коду або даних зламала інші частини програми.

3. *Тестування складання* визначає відповідність випущеної версії початковим критеріям якості. За своїми цілями є аналогом димового тестування, спрямованого передачу нової версії на подальше тестування чи в експлуатацію. Воно може проникати глибше – залежно від вимог якості випущеної версії;

4. *Sanity* – вузько спрямоване тестування, достатнє для доказу того, що конкретна функція працює відповідно до заявлених у специфікації вимог. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін, зроблених у ній чи оточенні. Зазвичай виконується вручну.

5. *Приймальне (acceptance) тестування* – вид тестування, що проводиться на етапі здачі готового продукту (або готової частини продукту) замовнику. Метою приймального тестування є визначення готовності продукту, що досягається шляхом проходу тестових сценаріїв та випадків, які побудовані на основі специфікації вимог до ПЗ. Результатом приймального тестування може стати:

- надсилання проєкту на доопрацювання.
- здача його замовнику як виконане завдання.

Це фінальний етап тестування продукту перед його випуском. При цьому, він не є ретельним, всеохоплюючим і повним – тестується перважно тільки основний функціонал. Приймальний тест проводиться або самим замовником, або групою тестувальників, які представляють інтереси замовника, або тестувальниками компанії-розробника. Це залежить від переваг компанії-замовника.

6. *Повторне тестування* – перевірка, під час якої виконуються тестові сценарії, що виявили помилки під час останнього запуску для підтвердження успішності виправлення цих помилок.

7. *Тестування документації* (вимог) важливо починати на ранніх етапах розробки, коли тільки починають з'являтися вимоги до продукту.

Контрольні запитання для самоконтролю

1. За допомогою якого виду тестування можна гарантовано виявити всі дефекти?

2. Який з методів тестування використовується для визначення можливостей ПЗ при масштабуванні, наприклад, при незначному збільшенні кількості одночасних користувачів?

3. Назвати нефункціональні види тестування.

4. Коли застосовується Smoke тестування?

5. Який вид тестування використовують для перевірки того, що після змін колишні функціональності працюють?

6. Назвати функціональні види тестування.

7. Який вид тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можна виміряти різними величинами?

8. На якому рівні тестування перевіряється відповідність системи до вимог?

Завдання

Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними видами та напрямками тестування.

Дати відповіді на контрольні запитання.

Лабораторна робота 12
Розробка специфікації вимог до програмного продукту

Мета роботи: навчитися розробляти специфікації вимог до ПЗ.

Теоретичні відомості

1. Складові специфікації вимог

Специфікація вимог (Software Requirements Specification, SRS) – це документ з описом, в якому міститься набір вимог до програмного продукту. Вимоги структуруються та описують логіку роботи продукту (функціональні вимоги), його зовнішній вигляд (користувацький інтерфейс), обмеження в розробці та нефункціональні вимоги. Для опису функціональних вимог часто використовуються користувацькі сценарії (use cases). У користувацьких сценаріях подані варіанти того, як користувач може взаємодіяти з ПЗ. Нефункціональні вимоги описують обмеження пов'язані з дизайном продукту та його реалізацією (продуктивність, безпека, надійність, сумісність, проєктні обмеження, стандарти якості тощо).

Специфікація вимог має у структуру, рекомендовану стандартом IEEE 830. Структура може дещо різнитися у різних проєктах, але в цілому виглядає так:

Вступ (Introduction):
- призначення, мета (Purpose);
- термінологія (Document conventions);
- переважна аудиторія та послідовність сприйняття;
- масштаб проєкту;
- посилання на джерела (References);

Загальний опис (Overall Description):
- бачення, перспективи продукту (Product features);
- функціональність продукту;
- класи та характеристики користувачів;
- середовище функціонування продукту (Operating environment);
- рамки, обмеження, правила і стандарти (Design and implementation constraints);
- документація для користувачів (User documentation);
- припущення та залежності;

Функціональність системи, системні характеристики (Functional requirements, System Features):
- функціональний блок, «фіча» продукту (їх може бути декілька);

- опис та пріоритет (Description and priority);
- причинно-наслідкові зв’язки, алгоритми (рух процесів, workflows) (Stimulus / Response sequences);
- функціональні вимоги (Functional requirements);

Вимоги до зовнішніх інтерфейсів (Interface Requirements):

- інтерфейси користувача (UX);
- програмні інтерфейси;
- інтерфейси обладнання;
- інтерфейси зв’язку та комунікацій;

Нефункціональні вимоги (Nonfunctional Requirements):

- вимоги до продуктивності (Performance requirements);
- вимоги до збереження (даних);
- критерії якості програмного забезпечення (Software quality attributes);
- вимоги до безпеки системи (Security requirements);

Інші вимоги (Other Requirements):

- Додаток А: Глосарій (Appendix A: Glossary);
- Додаток Б: Моделі процесів і предметної області (Appendix B: Analysis Models);
- Додаток В: Список пропозицій (Appendix C: Issues list).

Незважаючи на значну кількість пунктів у специфікації, ядро вимог до ПЗ міститься у функціональних вимогах. Найчастіше, їх подають у вигляді сценаріїв користувача (Use cases).

Кожен сценарій використання зосереджується на описі того, як досягти мети або вирішення завдання. Для більшості програмних проєктів це означає, що потрібними є численні сценарії використання, щоб визначити необхідний набір властивостей нової системи. Ступінь формальності програмного проєкту та його стадії впливатиме на необхідний рівень деталізації, для кожного сценарію використання.

Є кілька вимог до складання специфікації:

- опис всіх функцій має бути максимально коротким і чітким;
- не допускати двозначних описів: кожна сутність повинна бути гранично зрозуміла будь-якій людині;
- водночас: простота;
- деталізація повинна бути в рамках максимального розуміння та уникнення надлишкового тексту.

Погано описані вимоги згубно позначаються на розумінні членів команди розробки того, який продукт має вийти в результаті. Нечіткі вимоги можуть бути інтерпретовані членами команди по-різному, внаслідок чого можливі ситуації, коли одна задача реалізується декількома фахівцями одночасно в абсолютно різний спосіб та в іншому вигляді. Це спричиняє виникнення помилок у продукті, чого б не сталося, якби і самі вимоги були уточнені, протестовані та виправлені.

Добре описані вимоги визначаються такими критеріями якості:

– **Атомарність**: вимоги повинні бути описані так, щоб їх не можна було уточнити ще детальніше або розбити одну вимогу на декілька. Приклад поганої вимоги: «користувач може зареєструватися та додати у профіль особисту інформацію». Реєстрація та додавання інформації – це дві різні функції, які повинні бути описані і уточнені в різних пунктах специфікації. Приклад хорошої вимоги: «користувач може підписатися на оновлення інших користувачів»;

– **Завершеність**: вимоги до одного функціоналу повинні бути описані в одному пункті специфікації. Не можна допускати ситуацію, коли один і той самий функціонал описаний у різних частинах документа;

– **Послідовність**: вимога не повинна суперечити іншим вимогам та обмеженням системи. Наприклад, із соціальних мереж повинен запитуватися телефонний номер користувача. Очевидно, що дана вимога суперечить обмеженням соціальних мереж, позаяк номер телефона є прихованою інформацією, яку соціальні мережі не надають. Користувач повинен сам ввести номер телефона, щоб система його отримала. Подібних протиріч не повинно бути і між пунктами специфікації. Наприклад, одна і та сама кнопка має називатися однаково у всьому документі;

– **Відстеження (трасування)** – критерій, який дозволяє зрозуміти, чому було прописано саме таку вимогу. Наприклад, для сайту з продажу алкогольних напоїв вимогою може бути реєстрація тільки повнолітніх користувачів через законодавчі обмеження;

– **Актуальність** – критерій, який перевіряє відповідність вимог сучасним реаліям (наприклад, щодо законодавчої/технічної сторони або щодо зручності використання продукту, актуальності його користувацького інтерфейсу). Не варто прописувати у вимогах застарілі браузери, наприклад IE, або застарілі мобільні девайси та їхні ОС, оскільки вони практично не використовуються;

– **Здійсненність**: перевірка на те, що вимогу можливо реалізувати за допомогою актуальних наявних на даний момент технологій. Наприклад, вимога того, що система повинна давати швидкий відклик (тут потрібна конкретизація, наприклад, не більше 3 секунд) при помірному навантаженні (тут також потрібна конкретизація кількості користувачів та виконуваних ними дій) може існувати. Але вимога відклику менше секунди при дуже великому навантаженні – нездійсненна за сучасних реалій вимог;

– **Зрозумілість (доступність)**: вимога має бути сформульована достатньо чітко, конкретно та однозначно, для того, щоб вона однаково розумілася всією командою розробки;

– **Верифікованість** – критерій, за яким визначається, чи можна порівняти готовий продукт з вимогою і перевірити, чи виконується вона. Якщо вимога описана розмито, перевірити її не вийде. Наприклад: фото повинно завантажуватися швидко, налаштування профілю повинні бути інтуїтивно зрозумілими, оформлення замовлення повинно відбуватись легко. Для того, щоб перевірити такі вимоги, явно необхідні уточнення, як це «швидко», «інтуїтивно зрозуміло», «легко»;

– **Обов'язковість** – визначення того, наскільки важливе та пріоритетне виконання даної вимоги. Усім вимогам у специфікації мають бути призначені важливість, стабільність і пріоритет. За допомогою цього вибираються першочергові значення для виконання командою розробки завдання;

– **Ідентифікація**: у кожної вимоги повинен бути унікальний ідентифікатор, за допомогою якого вимога прив'язується до інших артефактів проєкту. Наприклад, в описі тест-кейса вказується ідентифікатор вимоги, яка перевіряється цим тест-кейсом;

– **Модифікація** – вимоги повинні бути легко модифіковані (внесення змін) за потреби;

– **Повнота** – найскладніший критерій, згідно з яким вимоги повинні вичерпно описувати весь функціонал системи. Все, що система повинна виконувати, повинно бути зафіксовано у вимогах, інакше можна припуститися серйозної помилки під час проєктування архітектури. Складність у тому, що на початкових етапах проєктування системи доволі важко точно вказати та описати всі функції. Вимоги спочатку описуються більш загальними твердженнями, а далі уточнюються.

2. Як тестуються вимоги?

Тестування вимог – це необхідний етап, що дозволяє поліпшити їх шляхом уточнення, деталізації, а також забезпечити взаєморозуміння між членами команди та уникнути різного трактування. Окрім цього, тестування вимог допоможе зрозуміти, чи можуть вони бути реалізовані загалом (чи достатньо ресурсів, часу, бюджету або можливо це з точки зору технологій).

Існують такі техніки тестування вимог:

1. **Взаємний перегляд.** Підрозділяється, в свою чергу, на:

– *Побіжний перегляд* – автор вимог надає документ на швидкий перегляд колегам, які дають свої зауваження, рекомендації, ставлять запитання у формі простого неформального обговорення;

– *Технічний перегляд* – вимоги надаються автором на перегляд групі фахівців;

– *Формальна інспекція* – вимоги переглядаються великою групою фахівців із документуванням усіх зауважень.

2. **Ставити запитання.** Якщо під час вивчення вимог виникають запитання, всі суперечливі моменти уточнюються у досвідчених колег або замовника.

3. **Верифікація вимог.** Для можливості перевірки можна спробувати створити чек-лист або тест-кейс для конкретної вимоги. Якщо вдається швидко придумати перевірки для чек-листа або тест-кейса – це вже непогано.

4. **Уявити поведінку реалізованої системи** – роботу користувача із системою, створеною за тестовими вимогами. Можливо вийде помітити незрозумілі або неоднозначні моменти в роботі з системою.

Графічна візуалізація та прототипування – подання інформації у вигляді рисунків, схем, створення прототипу системи (користувацького інтерфейсу)

допомагає краще аналізувати інформацію у специфікації, знаходити невідповідності та неточності.

Крім того, що необхідно добре протестувати вимоги та знайти неточності, потрібно правильно сформулювати і залишити зауваження до вимог. На які помилки при складанні зауважень слід звернути увагу і чого слід уникати? Ось декілька порад:

– Не змінювати формат файлу з документацією. Не потрібно видаляти або змінювати вихідний текст, потрібно залишати коментарі до тексту або пропонувати правки. Документ має залишатися в форматі, придатному для редагування (TXT/Excel/DOC), формат .pdf або картинки не підійдуть;

– Відзначати в коментарях слід тільки проблемні місця. Вимоги, які сформульовані добре, ніяк відзначати не потрібно. Це лише ускладнює роботу із зауваженнями, оскільки серед усіх коментарів доведеться вибирати ті, які потрібно виправляти;

– Не описувати одне і те саме зауваження в декількох місцях. Тут працює той же критерій якості, що і з самими вимогами. Якщо є потреба писати одне і те саме зауваження до однієї і тієї ж інформації, краще це зауваження винести в кінець документа та перерахувати список пунктів, яких воно стосується. А в самих пунктах робити посилання на зауваження наприкінці документа;

– Точно вказувати місце в тексті, якого стосується зауваження. Не варто виділяти весь абзац, якщо зауваження стосується одного речення. У тому ж Word можна виділити потрібну частину тексту та написати коментар саме до неї;

– Якщо необхідно додати уточнююче запитання, його слід формулювати дуже точно та продумано. Наприклад, між запитаннями «Що таке усталене налаштування?» та «Які налаштування усталені?» велика різниця. Друге уточнює важливу для специфікації інформацію, а перше – абсолютно безглузде та некомпетентне;

– Не варто писати дуже довгі коментарі, короткий чітко сформульований і структурований текст без орфографічних помилок сприймається набагато легше;

– Не писати зауваження у вигляді критики тексту або його автора, тільки конструктивні коментарі. Категоричні зауваження в стилі «це реалізувати неможливо» також потрібно доводити;

– Не редагувати самостійно вимоги без узгодження. Вносити правки у специфікацію можна тільки після узгодження з відповідальними особами. Інакше може виникнути вкрай серйозна ситуація, коли щось в продукті реалізоване не так, як планувалося, через неузгоджені зміни у вимогах.

Специфікація вимог – важливий документ, який в ідеалі повинен бути описаний для кожного продукту. Добре описані вимоги спрощують роботу всієї команди розробки – програмістам легше зрозуміти, як повинен бути реалізований продукт і як правильно спроєктувати архітектуру, а для тестувальників, крім усього іншого, це ще й джерело очікуваного результату при перевірці.

Вміння добре писати та тестувати вимоги – показник компетентності та професіоналізму.

SRS не є обов'язковим документом, проте може допомогти у середніх та великих проєктах.

3. Зразок (шаблон) специфікації вимог

Специфікація вимог до програмного продукту
(Software Requirements Specification, SRS)
для <Назва проєкту> (Зразок)

1. ВСТУП

Призначення, мета

<Визначити продукт, вимоги до якого описані в цьому документі. Описати межі продукту, зокрема якщо цей документ описує лише частину системи чи окрему підсистему.>

Термінологія

<Описати всі незрозумілі технічні слова або терміни, які зустрічаються в SRS. При цьому опис незрозумілого слова не може містити іншого незрозумілого слова. Варто описати термін якомога докладніше простою, зрозумілою мовою. Чим більше описано незрозумілих речей, тим простіше буде потім проєктувати.>

Посилання

<Список літератури (документів чи вебадрес), в якій можна знайти підстави використаних технологій та фактів. Може містити зразки користувацьких інтерфейсів, контракти, стандарти, варіанти використання. Посилання та їх опис мають бути максимально повними, щоб у разі чого (лінк помер просто) можна було знайти цей матеріал.>

2. ЗАГАЛЬНИЙ ОПИС

Бачення, перспективи продукту

<Описати частини функціоналу на високому рівні. Більш детально кожна частина функціоналу буде описана у наступному розділі. Тут описують контекст і витоки продукту, наприклад, стан продукту як члена сімейства продуктів, заміна наявних систем чи новий самодостатній продукт. Якщо SRS визначає компонент великої системи, пов’язати вимоги великої системи з функціональністю цього продукту і визначити інтерфейси між ними. Тут бажано розмістити прості діаграми, наприклад DFD[23], що покажуть основні компоненти загальної системи та взаємодію підсистем.>

Характеристики продукту

<Резюмувати основні характеристики продукту чи суттєві функції, що він здійснює чи дозволяє здійснювати користувачу. Деталі подаються в наступному розділі, тому тут потрібне лише узагальнення. Описати функції, щоб вони були зрозумілими будь-якому читачу документа. Ефективним є подання основ-

[23] Діаграма потоків даних (Data Flow Diagram) моделює потоки даних в інформаційній системі.

них груп пов'язаних вимог та їхніх зв'язків діаграмами потоків даних чи діаграмами класів.>

Класи користувачів та їх характеристики

<Визначити різні класи користувачів, які очікувано будуть використовувати продукт. Класи користувачів можуть диференціюватись, базуючись на частоті використання, підмножині функцій продукту, яка використовується, технічній експертизі, рівнях безпеки чи привілеїв, рівню освіти чи досвіду. Описати доцільні характеристики кожного класу користувачів. Відділити пріоритетні класи користувачів від тих, що є менш важливими.>

Середовище функціонування

<Описати середовище, в якому буде функціонувати продукт, включаючи апаратну платформу, операційну систему, версії компіляторів, бази даних, сервера, софт, залізо та інші програмні компоненти чи аплікації, з якими воно має коректно співіснувати.>

Обмеження проєктування і реалізації

<Описати питання чи пункти, що будуть обмежувати можливості, доступні розробникам. Може містити: корпоративні чи регуляторні правила; апаратні обмеження (часові вимоги, вимоги щодо пам'яті); взаємодії з іншими аплікаціями; специфічні технології, інструменти, бази даних; паралельні операції; мовні вимоги; протоколи комунікацій; обговорення безпеки; проєктні програмні стандарти (наприклад, якщо організація замовника буде відповідальною за супровід продукту).>

Документація користувача

<Описати, яка документація потрібна для користувачів даного продукту. Можливо це книга з HTML, якщо це HTML редактор. Можливо це керівництво по використанню, інструкції чи онлайн допомога, що буде надаватися разом з програмним продуктом. Тут варто визначити формати та стандарти документації.>

Припущення та залежності

<Список припущень (які суперечать відомим фактам), що можуть мати вплив на вимоги встановлені в SRS. Може включати комерційні компоненти, які планується використовувати, факти щодо розробки середовища функціонування. Визначити залежність проєкту від зовнішніх факторів, таких як програмні компоненти з інших проєктів, які планується повторно використовувати.>

3. ХАРАКТЕРИСТИКИ (ФУНКЦІОНАЛЬНОСТІ) СИСТЕМИ

<Ця частина ілюструє організацію функціональних вимог до продукту через характеристики системи, основні сервіси, які надає продукт. Цей розділ можна організувати через варіанти використання, режими операцій, користувацькі класи, об'єкти класів, функціональну ієрархію, чи їх комбінації, залежно від того, що найбільш логічно придатно для продукту.>

Характеристика системи 1 (2, 3)

Назва

<Вказати змістовну назву характеристики системи (фічі проєкту) кількома словами і надати їй унікальний ідентифікатор, наприклад, server.html.editor.>

Опис і пріоритет

<Надати короткий, але детальний опис функціоналу: навіщо він і що має робити у системі? Також вказати пріоритет виконання відповідної характеристики: високий, середній або низький. Можна також включати специфічні оцінки, кожну з яких оцінити за шкалою від 1 до 9.>

Причинно-наслідкові зв'язки, алгоритми (рух процесів, workflows)

<Тригер запуску фічі. Список послідовностей дій користувача і відгуків системи, що спричиняють режим визначений для цієї характеристики. Це відповідає елементам діалогу асоційованим з варіантами використання. Коли фіча запускається і як поводиться при запуску? Наприклад, HTML-редактор з'являється при натисканні користувачем команди меню *Відкрити HTML-редактор*.>

Функціональні вимоги

<Перелічити детальні функціональні вимоги, асоційовані з цією характеристикою. Це можливості продукту, які мають бути реалізовані, щоб користувач скористався сервісами чи виконав варіант використання, включаючи те, як продукт має реагувати на помилкові умови чи неправильні введення. Вимоги мають бути короткими, повними, недвозначними, верифіковуваними і необхідними. Кожна вимога має бути унікально ідентифікована номером чи значущою міткою певного виду, наприклад: REQ-1, REQ-2 тощо.>

4. ВИМОГИ ЗОВНІШНІХ ІНТЕРФЕЙСІВ

Користувацькі інтерфейси

<Описати логічні характеристики кожного інтерфейсу між ПЗ та користувачами. Може містити зразки зображень екрану, GUI стандарти чи керівництва стилів сімейства продуктів, яких треба дотримуватись, розмітка екрану, типи кнопок і функцій (наприклад, допомога), що з'являються на кожному вікні, комбінації клавіш, вигляд відображення повідомлень про помилки тощо. Визначити програмні компоненти, для яких потрібні користувацькі інтерфейси. Деталі розробки користувацького інтерфейсу мають бути документовані в окремій специфікації користувацького інтерфейсу.>

Апаратні інтерфейси

<Описати логічні та фізичні характеристики кожного інтерфейсу між ПЗ та апаратними компонентами системи. Може містити типи підтримуваних пристроїв, природу даних і керуючих взаємодій між ПЗ та апаратними засобами, а також комунікаційні протоколи, які будуть використані.>

Програмні інтерфейси

<Описати зв'язок між продуктом та іншими специфічними програмними компонентами (назва і версія), включаючи бази даних, операційні системи, інструменти, бібліотеки та інтегровані комерційні компоненти. Визначити дані і повідомлення, які поступають в систему і виходять з неї, описати мету кожної. Описати необхідні сервіси і природу комунікацій. Зіслатися на документи, що описують протоколи програмних інтерфейсів. Визначити дані, які будуть спільно використовуватись програмними компонентами.>

Комунікаційні інтерфейси

<Описати вимоги, пов’язані з комунікаційними функціями: електронна пошта, веббраузер, мережеві протоколи, електронні форми тощо. Визначити прийнятні формати повідомлень. Визначити комунікаційні протоколи, які будуть використовуватись (FTP, HTTP, HTTPS). Визначити безпеку комунікацій або питання шифрування, швидкість передачі даних і механізми синхронізації.>

5. НЕФУНКЦІОНАЛЬНІ ВИМОГИ

Вимоги продуктивності

<Якщо є вимоги продуктивності до продукту в різних середовищах, описати їх та пояснити. Визначити часові залежності для систем реального часу. Описати такі вимоги настільки точно, як це можливо.>

Вимоги надійності

<Вимоги до продуктивності накладають певні обмеження. Наприклад, база даних проєкту має витримувати 1000 запитів за секунду. Такі вимоги призводять до колосальної роботи з оптимізації проєкту. Тут варто визначити вимоги, пов’язані з можливими втратами, пошкодженнями, що можуть виникати при використанні продукту. Визначити заходи безпеки чи дії, які треба прийняти для запобігання цьому.>

Вимоги безпеки

<Визначити вимоги, що стосуються безпеки чи питань секретності, щодо використання продукту чи захисту даних, які використовуються чи створюються. Визначити вимоги аутентифікації користувачів.>

Атрибути якості програмного продукту

<Визначити додаткові якісні характеристики до продукту, важливі для замовників чи розробників. Наприклад, адаптовуваність, придатність, коректність, гнучкість, функціональна сумісність, супроводжуваність, портативність, надійність, стійкість, тестопридатність, зручність використання. Тут вказують те, які тести використати, які метрики використовуватиме визначення якості коду, скільки коду має бути покрито тестами.>

6. ІНШІ ВИМОГИ

<Визначити інші вимоги, що не розкриті в SRS. Це може включати вимоги бази даних, вимоги інтернаціоналізації, юридичні вимоги тощо.>

Додаток А: Словник

<Визначити всі терміни, необхідні для правильної інтерпретації SRS, включаючи абревіатури та скорочення.>

Додаток В: Моделі процесів і предметної області

<Розділ визначає, які діаграми потрібно використовувати при написанні SRS, як-от: діаграми потоків даних, діаграми класів, діаграми станів чи діаграми прецендентів.>

Додаток С: Список пропозицій

<Динамічний список відкритих пунктів вимог, які варто вирішити в майбутньому: незакінчені рішення і конфлікти, які очікують розв’язання.>

Контрольні запитання для самоконтролю

1. Що таке вимоги до програмного забезпечення?
2. Дайте означення специфікації вимог до ПЗ.
3. Які ви знаєте основні типи вимог до ПЗ?
4. Назвіть характеристики якісних вимог.
5. В чому полягають вимоги користувачів?
6. Що таке функціональні вимоги?
7. В чому полягають нефункціональні вимоги?
8. Назвіть основні джерела отримання інформації про потреби клієнтів.
9. Розкажіть про методи збирання вимог.

Лабораторне завдання

1. Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними етапами роботи над вимогами та їх тестуванням.

2. Розробити специфікацію вимог для певного програмного продукту та оформити її на кшталт зразка, поданого в п.3 Теоретичних відомостей. Вибір програмного продукту для створення специфікації може бути довільним (із набору наявних програмних застосунків на вашому телефоні, якийсь конкретний сайт тощо).

3. Оформити звіт і захистити лабораторну роботу.

Самостійна робота 7
Тестування вимог

Мета роботи: навчитися аналізувати вимоги до ПЗ.

Теоретичні відомості

Вимога (requirement) – це специфікація (опис) того, що має бути реалізовано у ПЗ без деталізації технічного боку рішення.

Вимоги до вимог:
- коректність;
- недвозначність;
- повнота набору вимог;
- несуперечність набору вимог;
- перевірюваність (тестопридатність);
- трасування;
- зрозумілість.

У книзі «Exploring Requirements: Quality before Design» її автори Дональд Гаус та Джеральд Вайнберг досліджують зв'язок між вимогами та тестами: "Одним із найефективніших способів тестування вимог є тест-кейси, дуже схожі на ті, що використовуються для тестування готової системи. Григорій Мельник і Боб Мартін[24] далі розширюють це і стверджують: "Якщо формалізувати це твердження, тести та вимоги стають рівнозначними". Тести мають бути такими чіткими, щоб їх можна було автоматизувати.

Тестування вимог (документації) важливо починати на ранніх етапах розробки, коли тільки починають з'являтися вимоги до продукту.

Під час тестування вимог та документації тестувальник має справу з різною документацією, що описує вимоги до системи, з посібниками для користувача, посібниками з експлуатації та встановлення тощо. При тестуванні вимог фахівець повинен мати знання в предметній галузі, на яку писалися ці вимоги. Інакше тестувальник може просто не побачити помилки у вимогах або, навпаки, припуститись помилки, не знаючи суті того, про що сказано в документі.

При тестуванні вимоги перевіряються на відповідність таким критеріям[25]:

– **завершеність** – вимога щодо повноти і закінченості з погляду надання всієї необхідної інформації. Типовими проблемами можна назвати відсутність нефункціональних вимог або посилань на відповідні нефункціональні вимоги. Наприклад, "паролі повинні зберігатися в зашифрованому вигляді", а який ал-

---

[24] Martin R., Melnik G. Tests and Requirements, Requirements and Tests: A Möbius Strip. 2008. Vol. 25, No. 1. URL: http://gmelnik.com/papers/Melnik-Martin-Tests-and-Requirements-The-Moebius-Strip-IEEE-Software-2008.pdf

[25] ОПІ та ТПЗ : лабораторний практикум для студентів напряму підготовки "Комп'ютерні науки" / Укл.: Петровський С.С. Хмельницький: ХНУ, 2020. 76 с.

горитм шифрування не зазначено. Крім того, може бути вказано лише частину переліку, наприклад, "експорт здійснюється у формати PDF, PNG та інші", а що розуміється під "та інші" не зрозуміло;

– **атомарність**: вимога є атомарною, якщо її не можна розбити на окремі вимоги без втрати завершеності, і якщо вона описує одну і тільки одну ситуацію. Прикладом типової проблеми атомарності є наявність в одній вимозі кількох незалежних, приміром, "кнопка Restart не повинна відображатися при зупиненому сервісі, вікно Log має вміщувати не менше 20 записів про останні дії користувача". Або інший приклад: «кнопка Stop повинна відображатися при піднятому VPN сервісі, у тарифних планах, на одному з тарифів має бути плашка Best Seller». В цих прикладах навіщось в одному реченні описані різні елементи інтерфейсу в різних контекстах. Іншим прикладом може бути ситуація, коли вимога допускає різночитання через граматичні особливості мови. Наприклад: «якщо користувач підтверджує замовлення та редагує замовлення або відкладає замовлення, то має надаватися запит на оплату». Тут описані три різні випадки, і тому цю вимогу варто розбити на три окремі, щоб уникнути плутанини. Таке порушення атомарності часто спричиняє суперечливість;

– **несуперечність**: вимога не повинна містити внутрішніх протиріч та протиріч іншим вимогам та документам. Типовою проблемою щодо несуперечності є протиріччя всередині однієї вимоги, наприклад: «після успішного входу в систему користувача, який не має права входити в систему…». – Як він успішно увійшов у систему, якщо не мав такого права? Іншим прикладом є протиріччя між двома і більше вимогами, між таблицею і текстом, малюнком і текстом, вимогою і прототипом тощо, наприклад: "кнопка "Start" завжди повинна бути зеленою" і "кнопка "Start" скрізь повинна бути синьою". Крім того, вимога несуперечності стосується використання неправильної термінології або використання різних термінів для позначення одного і того самого об'єкта або явища, наприклад: "у разі, якщо розширення вікна становить менше 1024x768 ...". – Розширення – це позначення типу файлу, а для вікна – розмір або роздільна здатність;

– **однозначність** (недвозначність): вимога має бути описана без використання жаргону, неочевидних абревіатур та розпливчастих формулювань. Вона допускає лише однозначне об'єктивне розуміння. Автор вимог може бути впевнений, що вдаватися до деталей не потрібно, бо "це очевидно", "інтуїтивно зрозуміло". Але на кожному з наступних етапів (розробки, тестування, документування) можуть виникнути нові сенси та тлумачення вимог. Тому треба обов'язково уточнювати вимоги, де немає однозначності. Одною з типових проблем щодо однозначності є використання термінів або фраз, які допускають суб'єктивне тлумачення, наприклад: "програма повинна підтримувати передачу великих обсягів даних", а наскільки великих? Іншим прикладом є розпливчасті формулювання, наприклад: "у разі потреби оптимізації передачі великих файлів система повинна ефективно використовувати мінімум оперативної пам'яті, якщо це можливо". Ще неоднозначність спричиняє використання неочевидних або двозначних абревіатур без розшифровки, наприклад: "доступ до ФС здійснюється за допомогою системи прозорого шифрування". Проблеми такого роду

спричиняє і формулювання вимог з міркувань, що щось має бути очевидним, наприклад: "система конвертує вхідний файл із формату PDF у вихідний файл формату PNG". При цьому автор вважає цілком очевидним, що імена файлів система отримує з командного рядка, а багатосторінковий PDF конвертується в кілька файлів PNG, до імен яких додається "page-1", "page-2" тощо. Ця проблема перегукується із порушенням коректності. При тестуванні на однозначність слід перевіряти "галузі" вимог. Якщо є умови та винятки – треба перевіряти, що всі вони прописані і не було упущень, які можна трактувати по-різному;

– **виконуваність**: вимога має бути технологічно здійсненна і може бути реалізована в рамках бюджету та термінів розробки проєкту. Однією з проблем щодо виконуваності може бути "озолочення" (gold plating) – вимоги, які дуже довго і/або дорого реалізуються, але при цьому практично не корисні для кінцевих користувачів, наприклад: "налаштування параметрів для підключення до бази даних має підтримувати розпізнавання символів із жестів, отриманих з пристроїв тривимірного введення". Іншою проблемою є вимоги, які не реалізуються на сучасному рівні розвитку технологій, наприклад: "аналіз договорів має виконуватися із застосуванням штучного інтелекту, який виноситиме однозначний коректний висновок про рівень вигоди від укладення договору". Ще проблемними є вимоги, які в принципі не можливо реалізувати, наприклад: "система пошуку повинна передбачати всі можливі варіанти пошукових запитів і кешувати їхні результати";

– **обов'язковість**: якщо вимога не є обов'язковою для реалізації, вона повинна бути виключена з набору. Якщо вимога потрібна, але не дуже важлива, для вказівки цього використовується знижений пріоритет. Також виключити (або переробити) треба вимоги, які втратили актуальність. Типовими проблемами з обов'язковістю та актуальністю є вимоги, які були додані «про всяк випадок», хоча реальної потреби в ній не було і немає. Або коли вимозі виставлені неправильні значення пріоритету за критеріями важливості та/або терміновості. Ще одним випадком є застарілі вимоги, які не були перероблені або видалені;

– **простежуваність** (трасування) буває вертикальною (vertical traceability) та горизонтальною (horizontal traceability). Вертикальна дозволяє співвідносити між собою вимоги на різних рівнях, горизонтальна – співвідносити вимоги з тест-планом, тест-кейсами, архітектурними рішеннями. Для забезпечення простежуваності часто використовуються спеціальні інструменти управління вимогами (requirements management tool) та/або матриці простежуваності (traceability matrix). Типовою проблемою із простежуваністю є непронумеровані, неструктуровані вимоги, які не мають змісту та працюючих перехресних посилань. Проблеми виникають, коли під час розробки вимог не було використано інструментів та техніки управління вимогами або коли набір вимог неповний, має уривчастий характер з явними "пробілами";

– **модифікованість** характеризує простоту внесення змін в окремі вимоги та у весь набір. Про наявність модифікованості можна говорити, якщо при доопрацюванні вимог потрібну інформацію легко знайти, а її змінення не призводить до порушення інших описаних у цьому переліку властивостей. Типовою проблемою з модифікованістю є від початку суперечливі вимоги, позаяк у такій

ситуації внесення змін (не пов'язаних із усуненням суперечливості) лише посилює ситуацію. Крім того, коли вимоги не атомарні і не простежуються, то їх змінення з високою ймовірністю породжує суперечливість. Ще проблемними є вимоги, представлені у незручній для оброблення формі (наприклад, не використані інструменти управління вимогами, і як наслідок команді доводиться працювати з десятками великих текстових документів);

– **проранжованість**: вимоги можуть бути проранжовані за важливістю, стабільністю, терміновістю. Важливість характеризує залежність успіху проєкту від виконання вимоги. Стабільність – ймовірність того, що в найближчому майбутньому у вимогу не буде внесено жодних змін. Терміновість визначає розподіл у часі зусиль проєктної команди щодо реалізації тієї чи іншої вимоги. Типові проблеми з проранжованістю полягають у її відсутності чи неправильній реалізації. Вони призводять до проблем з проранжованістю за важливістю, підвищують ризик неправильного розподілу зусиль проєктної команди, спрямування зусиль на другорядні завдання та кінцевий провал проєкту через нездатність продукту виконувати ключові завдання з дотриманням ключових умов. Проблеми з проранжованістю за стабільністю підвищують ризик виконання безглуздої роботи з удосконалення, реалізації та тестування вимог, які найближчим часом можуть зазнати кардинальних змін (аж до повної втрати актуальності). Також проблеми з проранжуваністю за терміновістю підвищують ризик порушення зазначеної замовником послідовності реалізації функціональності та введення її в експлуатацію;

– **коректність та перевірюваність** випливають із дотримання всіх перелічених вище характеристик. Перевірюваність передбачає можливість створити об'єктивні тест-кейси, які однозначно показують, що вимога реалізована правильно і поведінка програми точно відповідає їй. До типових проблем із коректністю також можна віднести помилки (особливо небезпечні помилки в абревіатурах, що перетворюють одну осмислену абревіатуру на іншу, що не має стосунку до актуального контексту. Такі помилки вкрай складно помітити). Також проблеми спричиняють наявність неаргументованих вимог до дизайну й архітектури та погане оформлення тексту і супутньої графічної інформації, граматичні, пунктуаційні та інші помилки в тексті. До такого роду проблем відносять і неправильний рівень деталізації, наприклад, надто глибока деталізація на рівні бізнес-вимог або недостатня – на рівні вимог до продукту. Крім того, вимоги до користувача, а не до програми, наприклад: "користувач повинен мати можливість надіслати повідомлення", але ми не можемо впливати на стан користувача.

Інші типи документації описують вже реалізоване ПЗ. Вони перевіряються на актуальність, коректність та однозначність.

Зазвичай на тестування вимог не виділяють час у проєкті, але є методи експрес-оцінки, які дозволять без великих трудовитрат виявити більшу частину помилок у вимогах.

Контрольні запитання

1. Що таке вимоги до ПЗ?
2. Чим відрізняється валідація від верифікації?
3. Які виділяють критерії якості?
4. Які існують етапи роботи над вимогами?
5. Хто виконує роботу з вимогами?
6. Які існують рівні вимог?
7. Які вимоги вважаються перевірюваними?
8. Які вимоги вважаються модифікованими?
9. Які вимоги вважаються недвозначними?
10. Які вимоги вважаються повними?
11. Які вимоги вважаються атомарними?
12. Які вимоги вважаються трасованими?
13. Які існують методи тестування вимог?

Завдання

1. Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними етапами роботи над вимогами та їх тестуванням.

2. Дати відповіді на контрольні запитання.

3. Проаналізувати (протестувати) специфікацію з вимогами до програмного продукту, розроблену під час виконання попередньої лабораторної роботи, методом перегляду на предмет відповідності критеріям якості вимог і скласти список уточнювальних питань до замовника для вироблення якісних вимог. При цьому варто зазначати розділ вимог, до якого виникли претензії.

4. Для виявлених дефектів вказати, який критерій якості порушений, аргументувати свою точку зору, наприклад:

Вимога *FNC-ACC 2.5. My Account*

- Мета: надати користувачеві можливість керування та внесення змін до особистого облікового запису.
- За бажання реалізувати можливість вставляти замість фото файл з аватаром.
- Ім'я не може бути порожнім.
- При виборі вже заняті кольори мають бути позначені і бути недоступними для вибору поточним користувачем.

Зауваження:

- Варто уточнити, які типи файлів дозволені для завантаження і якого розміру.
- Якщо є вимога, яка не є обов'язковою, як наприклад з аватарами, її не обов'язково описувати.
- Як має відбуватися синхронізація кольорів між девайсами?
- Уточнити: які символи можна вводити в поле з ім'ям, довжину імені, чи припустимі однакові імена у різних користувачів.

Лабораторна робота 13
Добір видів тестування

Мета роботи: навчитися добирати види тестів для конкретних тестових ситуацій.

Теоретичні відомості

Тестування ПЗ – креативна й інтелектуальна робота. Розробка правильних й ефективних тестів – доволі непросте заняття.

Сім принципів тестування,[26] розроблені за останні 40 років, є загальним керівництвом для тестування загалом.

1. Тестування показує наявність дефектів

Тестування може виявити наявність дефектів у програмі, але не довести їхню відсутність. Тим не менш, важливо складати тест-кейси, які будуть знаходити якнайбільше багів. Тим самим при належному тестовому покритті тестування дозволяє знизити ймовірність наявності дефектів у ПЗ. У той самий час, навіть якщо дефекти не були виявлені у процесі тестування, не можна стверджувати, що їх немає.

2. Вичерпне тестування неможливе

Неможливо провести вичерпне тестування, яке б покривало всі комбінації при введенні даних користувача і станів системи, за винятком зовсім примітивних випадків. Натомість необхідно використовувати аналіз ризиків та розстановку пріоритетів, що дозволить більш ефективно розподіляти зусилля щодо забезпечення якості ПЗ.

3. Раннє тестування

Тестування має починатися якомога раніше у життєвому циклі розробки ПЗ, а його зусилля мають бути сконцентровані на певних цілях.

4. Скупчення дефектів

Різні модулі системи можуть містити різну кількість дефектів, тобто щільність накопичення дефектів в різних елементах програми може відрізнятися. Зусилля з тестування повинні розподілятися пропорційно до фактичної щільності дефектів. Здебільшого більшість критичних дефектів знаходять в обмеженій кількості модулів. Це прояв принципу Парето: 80% проблем міститься у 20% модулів.

5. Парадокс пестициду

Проганяючи ті самі тести знову і знову, Ви зіткнетеся з тим, що вони знаходять все менше нових помилок. Оскільки система еволюціонує, багато раніше знайдених дефектів виправляють і старі тест-кейси більше не спрацьовують.

---

[26] Принципы тестирования. URL: https://qalight.ua/ru/baza-znaniy/printsipyi-testirovaniya/

Щоб подолати цей парадокс, необхідно періодично вносити зміни до наборів тестів, рецензувати і коригувати їх з тим, щоб вони відповідали новому стану системи і дозволяли знаходити якомога більшу кількість дефектів.

6. Тестування залежить від контексту

Вибір методології, техніки та типу тестування безпосередньо залежить від природи самої програми. Наприклад, ПЗ для медичних потреб вимагає набагато суворішої та ретельнішої перевірки, ніж, скажімо, комп'ютерна гра. З тих самих міркувань сайт з великою відвідуваністю повинен пройти через серйозне тестування продуктивності, щоб показати можливість роботи за умов високого навантаження.

7. Помилка про відсутність помилок

Той факт, що тестування не виявило дефектів, ще не означає, що програма готова до релізу. Знаходження та виправлення дефектів будуть неважливими, якщо система виявиться незручною у використанні і не задовольнятиме очікуванням та потребам користувача.

І ще кілька важливих принципів:

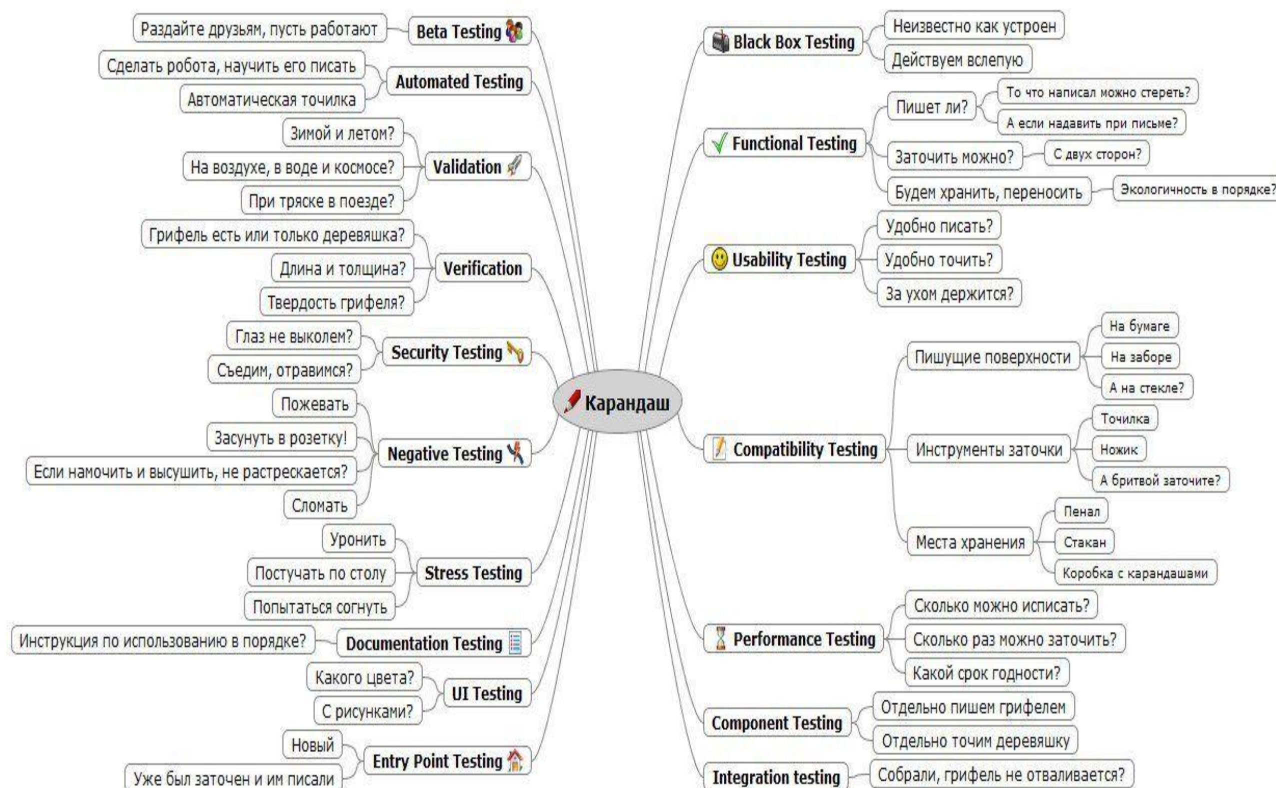
- тестування має проводитись незалежними фахівцями;
- залучення кращих професіоналів;
- тестування як позитивних, так і негативних сценаріїв;
- не допускати змінення у програмі у процесі тестування;
- вказувати очікувані результати виконання тестів.

Різновидів тестів багато. Маючи розуміння мети проведення тестування та систематизоване уявлення про види тестів, вже цілком можна відповісти на запитання "які види тестів вам потрібні".

Наведемо запитання, на які треба відповісти, визначаючи потрібні види тестів[27]:

- Які функціональні та нефункціональні вимоги висунуті до системи?
- З чого складатиметься система?
- На скільки добре тестувальники знають будову системи?
- Як та чим відтворювати тестові ситуації?
- На яких ділянках та в яких масштабах тестуватиметься система?
- На якому етапі розробки проводитимуться тести?
- Як ви будете описувати та зберігати тести?
- Чи добре знають тестувальники методи складання тестових сценаріїв?
- Чи розробники перевірятимуть самі себе та один одного?
- Чи досконала специфікація?

[27] Які випробування вам потрібні? Частина 2. Матриця видів тестування. URL: https://habr.com/ru/post/257159/



Контрольні запитання для самоконтролю

1. Що таке тестування?

2. Які існують види тестування залежно від запуску коду виконання? Охарактеризувати кожний з них.

3. Які існують види функціонального тестування? Охарактеризувати їх.

4. Які існують види нефункціонального тестування? Охарактеризувати їх.

5. Які існують види тестування залежно від рівня тестування у процесі розробки? Охарактеризувати кожний з них.

6. Які існують методи тестування залежно від доступу до коду та архітектури програми? Охарактеризувати кожний з них.

7. Які існують види тестування залежно від ступеня автоматизації? Охарактеризувати кожний з них.

8. Що таке регресійне тестування? Коли варто його робити?

9. Коли доцільно робити стрес тестування?

10. У чому суть димового (smoke) тестування?

Лабораторне завдання

1. Використовуючи методичний матеріал та рекомендовану літературу за відповідною тематикою, ознайомитись з основними етапами роботи над вимогами та їх тестуванням.

2. Визначити, до якого саме виду тестування належать ситуації, описані в табл. 13.1 (до кожного випадку може бути кілька видів тестування).

Приклад:

Тестова ситуація	Вид тестування
Переконатися, що при натисканні на "Log out" користувач виходить із системи (розлогінюється)	динамічне, функціональне, тестування безпеки

Таблиця 13.1

№	Тестова ситуація	Вид тестування
1	Логін користувача в програмному застосунку	
2	Треба перевірити, що у користувача є можливість логіна за допомогою FB акаунта	
3	Треба проверіфікувати дефекти в новому білді клієнта під платформу Windows	
4	Для тестування отримуємо білд з новою функціональністю	
5	Пробуємо замість фото прикріпити відеофайл в налаштуваннях акаунта користувача	
6	Перевірити, що після виконання операції покупки в БД записалась правильна ціна	
7	Треба перевірити швидкість відповіді з API сервера при збільшенні активних водночас користувачів в 4 рази, що на 10% більше, ніж заявлено у вимогах	
8	Виклали тестовий білд в інтернет і дали посилання на скачування обмеженій кількості користувачів	
9	Для користувачів з різних регіонів виводяться різні тарифи на вкладці Purchases	
10	Перевірити роботу застосунку, якщо на авторизований сервер одночасно спробують авторизуватися 100 000 користувачів при припустимому навантаженні 50 000/ час	
11	Тестувальнику дали для ознайомлення документацію по новій функціональності	
12	Застосунок переведено на ESP і RUS локалі. Дизайнер вирішив змінити розмір кнопки "Sign In"	
13	Після впровадження нового дизайну екрана покупок піймали колегу в коридорі, дали йому девайс з новим білдом і попросили зробити покупку. При цьому спостерігали за його реакцією на новий дизайн і записували всі зауваження та складнощі, які виникали у нього в процесі купівлі	
14	Залишили застосунок працювати всю ніч	
15	Тестується новий білд. Документації немає, тестування виконують по заготовленим зарані користувальницьким сценаріям	
16	Після закінчення робіт по розробці нової функціональності розробник передав код застосунку для ознайомлення колезі	
17	Після того, як час сесії минає, користувача викидає (розлогінює) із застосунку	

3. На прикладі конкретного програмного застосунку, наявного у вашому мобільному телефону (узгодити з викладачем), придумати по 3 тести на види тестування, зазначені в табл. 13.2.

Приклад:

Вид тестування	**Що тестувати**
GUI (graphical user interface) тестування	1. Перевірити, що елементи інтерфейсу не виходять за межі вікна застосунка. 2. Перевірити, що при активації / деактивації змінюється стан (колір) вікна. 3. Іконки пунктів меню мають однаковий розмір і круглу форму. 4. Перевірити, що в усьому застосунку використовуються однакові шрифти

Таблиця 13.2

Вид тестування	**Що тестувати**
Функціональне тестування методом чорної скриньки	1. 2. 3.
Негативне функціональне тестування	1. 2. 3.
Навантажувальне тестування	1. 2. 3.
Smoke тестування	1. 2. 3.
Sanity тестування	1. 2. 3.
Тестування критичного шляху	1. 2. 3.
Тестування локалізації	1. 2. 3.
Стрес тестування	1. 2. 3.

4. Оформити звіт і захистити лабораторну роботу

Список літератури

1) Boehm B. A Spiral Model of Software Development and Enhancement. *IEEE Computer*. 1988. Vol.21, No.5. P.61-72. URL: http://www.sce.carleton.ca/faculty/ajila/4106-5006/Spiral%20Model%20Boehm.pdf
2) Capability Maturity Model for Software, Version 1.1 / M.Paulk, B.Curtis et al. *CMU–SEI–TR–024*, Soft. Engin. Institute, Pittsburg PA 15213, Feb. Pittsburg, 1993. 82 p.
3) DevOps: що ж це таке насправді. URL: https://habr.com/ru/company/piter/blog/430508/
4) DevSecOps Consulting Services. Integrating ‘Security as Code’ Culture within DevOps. URL: https://www.veritis.com/solutions/devops/devsecops-services/
5) Git. URL: https://git-scm.com/
6) Glossary of Software Engineering terms. URL: http://www.shellmethod.com/refs/seglossary.pdf.
7) IEEE Guide to the Software Engineering Body of Knowledge - SWEBOK®, 2004.
8) IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. URL: http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html.
9) IEEE-CS/ACM Software Engineering Ethics and Professional Practices. URL: http://www.computer.org/portal/site/ieeecs/menuitem.c5efb9b8ade9096b8a9ca0108bcd45f3/index.jsp?&pName=ieeecs_level1&path=ieeecs/content&file=ethics.xml&xsl=generic.xsl&.
10) Mora M. Balancing Agile and Disciplined Engineering and Management Approaches for IT Services and Software Products. 1st ed. IGI Global. 356 p.
11) Paasivaara M., Kruchten Ph. Agile Processes in Software Engineering and Extreme Programming – Workshops. Copenhagen: Springer, 2020. 330 p.
12) Pohl K, Rupp Ch. Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB compliant May 20, 2015. 175 p.
13) ProjectLibre documentation. URL: https://www.projectlibre.com/tags/projectlibre-documentation
14) Ronald J. L. Introduction to Software Engineering. CRC Press, 2016. 402 p.
15) Smith Gr. Next Gen DevOps: A manager's guide to DevOps and SRE Paperback, 3rd edition. 2019. 234 p.
16) Software Engineering - Guide to the Software Engineering Body of Knowledge (SWEBOK) Technical report ISO/IEC TR 19759 IEEE First edition 2005-09-15.
17) Sommerville I. Software Engineering. 9th ed. Addison-Wesley, 2011. 773 p.

18) Software Engineering. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. A Volume of the Computing Curricula Series. URL: http://sites.computer.org/ccse/SE2004Volume.pdf

19) SWEBOK V3.0. The Guide to the Software Engineering Body of Knowledge. IEEE Computer Society Professional Practices Committee (Керівництво з областей знань програмної інженерії). Tokio, 2014. 335 p.

20) Бабенко Л.П., Лавріщева К.М. Основи програмної інженерії: навч. посібник. К.: Знання, 2001. 269 с.

21) Бородкіна І. Л., Бородкін Г. О. Інженерія програмного забезпечення: навч. посібн. Київ: Центр навчальної літератури. 2018. 204 с.

22) Дегтярьова Л.М., Гроза П.М., Сомов С.В. Технології розробки програмного забезпечення: навч. посібник. Полтава: ПолтНТУ, 2017. 218 с.

23) ДСТУ ISO/IEC/IEEE 16326:2015 Розроблення систем та програмного забезпечення. Процеси життєвого циклу. Керування проєктами (ISO/IEC/IEEE 16326:2009, IDT). URL: http://online.budstandart.com/ru/catalog/doc-page?id_doc=67052

24) Лаврищева Е.М. Методы программирования. Теория, инженерия, практика. К.: Наук. думка, 2006. 450 с.

25) Лаврищева Е.М., Коваль Г.И., Коротун Т.М. Подход к управлению качеством программных систем обработки данных. *Кибернетика и системный анализ*. 2006. № 5. С.174-185.

26) Методичні вказівки до виконання лабораторних робіт з дисципліни «Конструювання програмного забезпечення» / Укладачі: Гусєва-Божаткіна В.А., Пономоренко Т.В. Миколаїв, 2013.

27) Методичні вказівки до лабораторних занять з дисципліни «Основи програмної інженерії» для студентів напряму 6.050103 Програмна інженерія / Укл.: В.В. Любченко. Одеса: ОНПУ, 2012. 36 с.

28) ОПІ та ТПЗ : лабораторний практикум для студентів напряму підготовки “Комп’ютерні науки” / Укл.: Петровський С.С. Хмельницький: ХНУ, 2020. 76 с.

29) Основы инженерии качества программных систем / Ф.И. Андон, Г.И. Коваль, Т.М. Коротун, Е.М. Лаврищева, В.Ю. Суслов. К.: Академпериодика. 2007. 678 с.

30) Петрик М.Р., Петрик О.Ю. Моделювання програмного забезпечення : навч.-метод. посібник. Тернопіль: Вид-во ТНТУ ім. Івана Пулюя, 2015. 200 с.

31) Про систему контролю версій. URL: https://git-scm.com/book/uk/Вступ-Тристани

32) Як працює система керування проєктами? URL: https://e-contentum.com.ua/projects-features

Навчально-методичне видання

Олена Григорівна Трофименко,
Сергій Юрійович Манаков,
Дмитро Георгійович Ларін

Основи програмної інженерії

Навчально-методичний посібник
для підготовки здобувачів вищої освіти
галузі знань 12 «Інформаційні технології»

Електронне видання

Підписано до друку 28.06.2022.
Ум-друк. арк. 24,36. Зам. № 2206-13.

Видано в ПП «Фенікс»
(Свідоцтво суб’єкта видавничої справи ДК № 1044 від 17.09.02).
Україна, м. Одеса, 65009, вул. Зоопаркова, 25.
Тел. +38 050 7775901 +38 048 7959160
e-mail: fenix-izd@ukr.net
www.feniksbooks.com