**Лабораторна робота №3**

**Базові принципи ООП**

**Мета роботи**: Використовуючи теоретичне підґрунтя про базові принципи ООП та підходи використання в Java виконати дії що будуть вказано в завданні до лабораторної роботи.

# ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

Для виконання лабораторної роботи необхідно використовувати репозиторій на github, що був створений при виконанні завдання до першої лабораторної роботи. Результати роботи необхідно буде завантажити на github для перевірки.

При використанні IDE створити проект під назвою lab03 для розміщення результатів виконання завдань. У випадку використання звичайного текстового редактору створити окрему директорію lab03 для виконання завдань.

Виконайте наступні завдання(кожне завдання має бути виконано в окремому файлі).

**ЗМІСТ**

[1. ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ 1](#_Toc166349197)

[2. ВИКОНАННЯ ЗАВДАННЯ ЛАБОРАТОРНОЇ РОБОТИ 3](#_Toc166349198)

[2.1. Реалізація Ticket 5](#_Toc166349199)

[2.2. Реалізація UserStory 6](#_Toc166349200)

[2.3. Реалізація Bug 7](#_Toc166349201)

[2.4. Реалізація Sprint 9](#_Toc166349202)

[2.5. Реалізація Main. Перевірка роботи коду завдання 10](#_Toc166349203)

[3. ВИСНОВОК 12](#_Toc166349204)

# ВИКОНАННЯ ЗАВДАННЯ ЛАБОРАТОРНОЇ РОБОТИ

|  |  |
| --- | --- |
| 1 | У цьому завданні ви формулюватимете бізнес-логіку планування спринту. [Спринт](https://en.wikipedia.org/wiki/Scrum_(software_development)#Sprint) - це основна одиниця розробки програмного забезпечення [SCRUM](https://en.wikipedia.org/wiki/Scrum_(software_development)). Спринти обмежені за часом, час спринту узгоджується при плануванні. Вам належить організувати планування спринту, що включає завдання, які вам потрібно реалізувати на основі базового класу *Ticket*. Врахуйте, у цьому випадку спринт приймає лише два підтипи класу *Ticket:Bug* та *UserStory*.  Ознайомтеся з діаграмою, яка показує публічний програмний інтерфейс цих класів та їхні відношення:    **Ticket**  Кожен *ticket* має свій ідентифікатор, ім'я та приблизний час для його виконання. Ці значення надаються через конструктор класу *Ticket*.  *Ticket* може вважатися завершеним або незавершеним. Щойно створений *ticket* вважається незавершеним.   * *getId*() - повертає ідентифікатор ticket'у. * *getName*() - повертає ім'я ticket'у. * *getEstimate*() - повертає оцінку ticket'у. * *isCompleted*() - повертає *true*, якщо ticket завершено, інакше — *false*. * *complete*() - переводить ticket у завершений стан.   **UserStory**  *UserStory* (історія користувача) представляє ticket, який може містити залежності. Залежності — це інші екземпляри *UserStory*, які мають бути завершені в першу чергу, щоб потім вдалося завершити і залежне від них *UserStory*. Залежності передаються через конструктор класу *UserStory*.   * *complete*() - як і метод Ticket#complete() - переводить ticket у завершений стан. Різниця лише в тому, що історія користувача може не бути завершена, якщо його залежність ще не завершена. * *getDependencies*() - повертає копію масиву ticket'ів на спринт, що захищена від змін. * *toString*() - повертає рядок, що представляє *UserStory*, використовуючи його ідентифікатор та ім'я. Приклад: з ідентифікатором =1, ім’я *UserStory* = "Об'єкт реєстрації користувача", отримаємо "[US 1] Об'єкт реєстрації користувача".   **Bug**  Багом вважається ticket, що відноситься до завершеної UserStory. Баги не існують самі по собі, без відповідного екземпляра UserStory.   * *createBug*(int id, String name, int estimate, UserStory userStory) - статичний метод створення екземпляра Bug. Повертає null, якщо UserStory має значення null або не завершено. Інакше повертає створений екземпляр Bug. * toString() - повертає рядкове представлення цього бага, використовуючи ідентифікатор, ім'я та ім'я відповідної UserStory. Приклад: з ідентифікатором = 2, ім'ям бага = "Додати пароль, що повторюється" і ім'ям відповідної UserStory = "Форма реєстрації" в результаті отримаємо "[Bug 2] Форма реєстрації: Додати повторюваний пароль".   **Sprint**  Спринти мають тимчасову ємність та обмеження кількості ticket'ів, які задаються через конструктор. Спринт не може містити тікет з сумарною оцінкою часу виконання, що перевищує тимчасову ємність спринту. Не допускається, щоб спринт містив більше тикетів, ніж визначено обмеженням кількості тикетів для цього спринту. Спринт повинен приймати ticket'и за допомогою методів *add*\*. Ці методи повертають *true*, коли вхідний ticket був прийнятий у спринт, і *false* в іншому випадку.  Спринт не повинен приймати   1. Значення null. 2. ticket'и, які вже завершені. 3. ticket'и, що мають значення оцінки часу виконання, яке, у разі додавання ticket'а, призведе до переповнення тимчасової ємності спринту. 4. Будь-який ticket, якщо досягнуто межі кількості ticket'ів у спринті.   Опис методів спринта   * *addUserStory*(UserStory userStory) - приймає userStory, якщо вона не дорівнює null, і не завершена. Повертає true, якщо історія користувача прийнята, у противному випадку false. * addBug(Bug bugReport) - приймає bug, якщо він не дорівнює null і не завершений. Повертає true, якщо баг прийнято, інакше false. * getTickets() - повертає захищену копію масиву ticket'ів на спринт. Переконайтеся, що ticket'и розташовані в тому ж порядку, як вони були прийняті в спринт. * getTotalEstimate() - повертає суму оцінок часу виконання всіх ticket'ів, прийнятих на спринт. |
|  |  |

## Реалізація Ticket

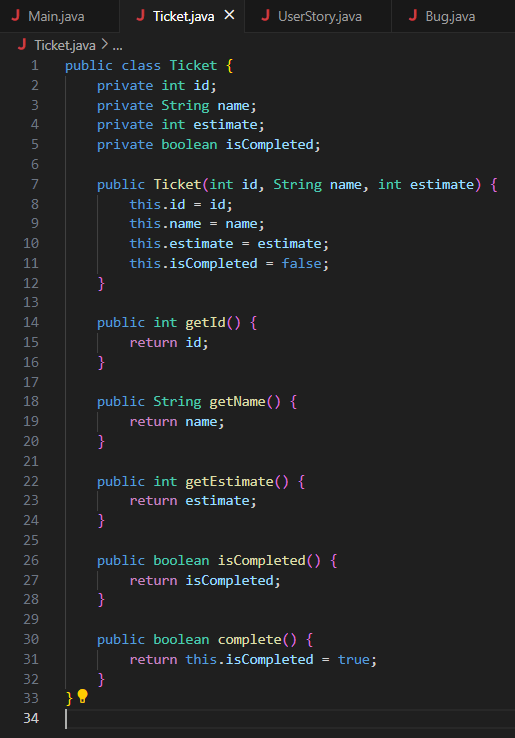


Рис. 2.1. Реалізація класу Ticket.

За умовою завдання кожен ticket має свій ідентифікатор, ім'я та приблизний час для його виконання. Ці значення надаються через конструктор класу Ticket. Ticket може вважатися завершеним або незавершеним. Щойно створений ticket вважається незавершеним.

Створюємо клас Ticket і змінні id, name, estimate, isCompleted з ідентифікатором доступу private і відповідними для них типами. isCompleted відповідатиме за те, чи завершений тікет.

Конструктор класу буде приймати три аргументи: id, name, estimate. Визначаємо їх як атрибути. І ще визначаємо один атрибут isCompleted, який буде дорівнювати false (тікет не завершений).

Потім після цього пишемо чотири методи-гетерів для отримання значень змінних id, name, estimate, isCompleted відповідно.

Ще створюємо метод complete(), який при виклику повертатиме присвоєння змінній isCompleted поточного об’єкта значення true, тобто завершувати тікет. Цей метод для завершення тікета.

## Реалізація UserStory

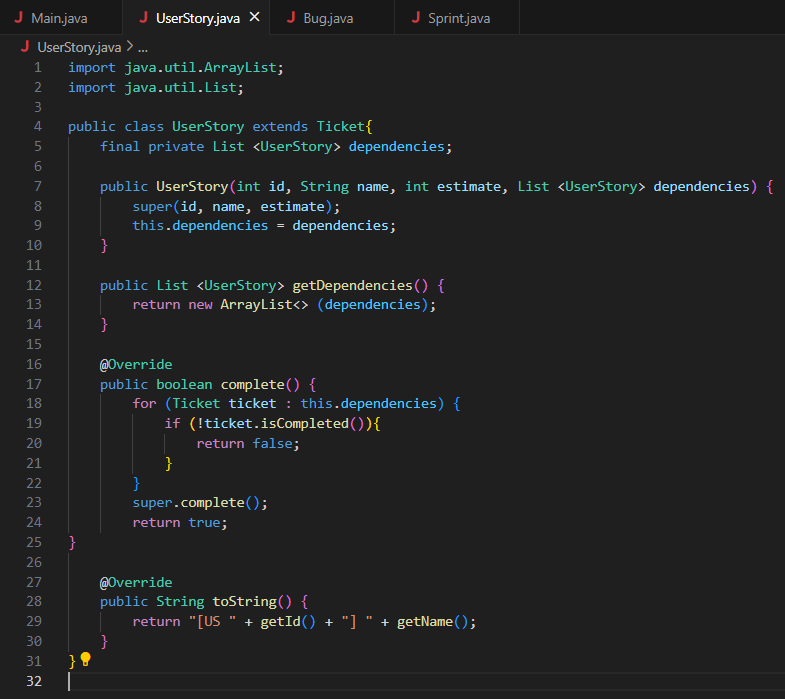


Рис. 2.2. Реалізація класу UserStory.

UserStory (історія користувача) за умовою завдання представляє ticket, який може містити залежності.

Створюємо клас UserStory, який успадкує клас Ticket. Оголошуємо список типу UserStory з назвою dependencies. Він буде потрібен для зберігання залежностей, що пов’язані з тікетом і юзерсторі водночас.

Конструктор буде приймати чотири аргументи: id, name, estimate та dependencies. У конструкторі визначаємо атрибути, три з них (id, name, estimate) беремо з батьківського класу Ticket за допомогою методу super(). І визначаємо атрибут залежності (dependencies).

Створюємо метод getDependencies(), який потім буде повертати копію масиву ticket'ів(тобто, dependencies) на спринт, що захищена від змін.

Далі перевизначаємо метод complete() у поточному класі. Тепер він циклом for буде перебирати кожен тікет в масиві dependencies поточного об’єкта UserStory і перевіряти, чи не були залежності тікета завершені, якщо не були(false), то метод поверне false, це означає, що не всі залежності були завершені. Якщо всі залежності були завершені, то викликаємо метод complete() з батьківського класу Ticket, щоб завершити тікет і повертаємо true.

Потім перевизначаємо метод toString() для зміни рядкового представлення об’єкта UserStory в ось такому форматі: "[US (id)] (name)". Тобто, щоб виходило, на кшталт такого: "[US 1] Об'єкт реєстрації користувача".

## Реалізація Bug

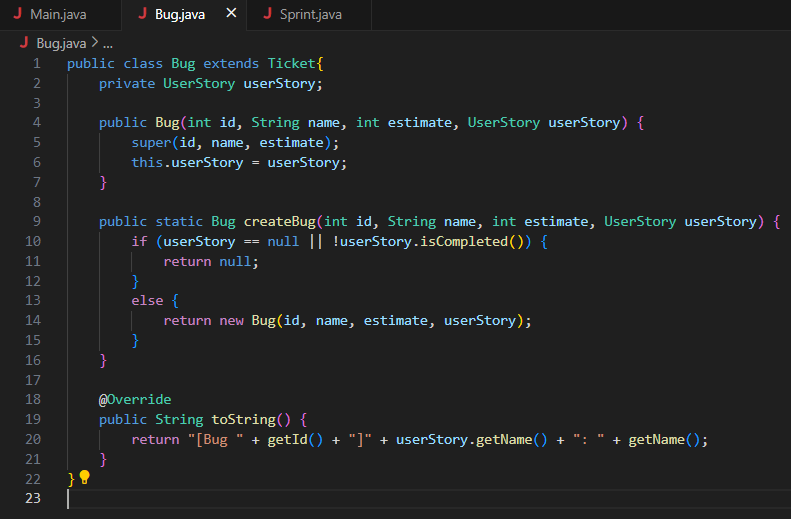


Рис. 2.3. Реалізація класу Bug.

За умовою завдання багом вважається ticket, що відноситься до завершеної UserStory. Баги не існують самі по собі, без відповідного екземпляра UserStory.

Створюємо клас Bug, який успадкує клас Ticket. Оголошуємо зміну типу класу UserStory з назвою userStory.

Конструктор буде приймати чотири аргументи: id, name, estimate та UserStory userStory. У конструкторі визначаємо атрибути, три з них (id, name, estimate) беремо з батьківського класу Ticket за допомогою методу super(). І визначаємо атрибут userStory, який буде дорівнювати аргументу userStory.

Далі реалізуємо метод createBug(), який прийматиме 4 аргумента: id, name, estimate, userStory. У цьому методі я реалізую перевірку, що якщо userStrory рівна null або якщо залежності userStrory не були завершені, то повернути null, інакше створити об’єкт класу Bug, передавши аргументи id, name, estimate, userStory і повернути його.

Перевизначимо метод toString() для зміни рядкового представлення об’єкта в ось такий формат: "[Bug (id)] (userStory.name): (nameBug)", тобто, має вийти щось подібне до цього: "[Bug 2] Форма реєстрації: Додати повторюваний пароль". Тобто, виводяться ідентифікатор бага, ім’я юзерсторі, ім’я багу.

## Реалізація Sprint

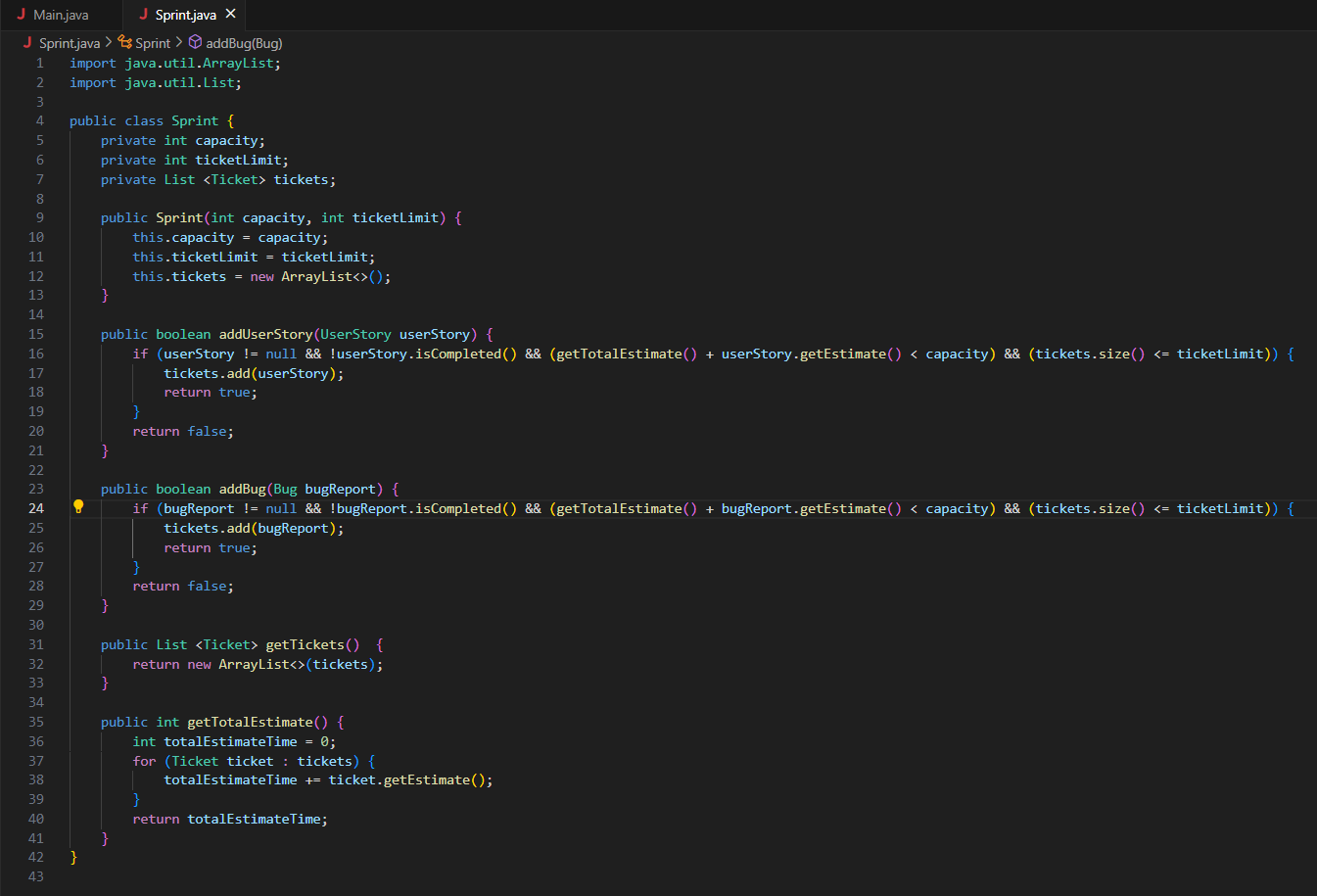


Рис. 2.4. Реалізація класу Sprint.

За умовою задачі спринти мають тимчасову ємність та обмеження кількості ticket'ів, які задаються через конструктор. Спринт не може містити тікет з сумарною оцінкою часу виконання, що перевищує тимчасову ємність спринту. Не допускається, щоб спринт містив більше тикетів, ніж визначено обмеженням кількості тикетів для цього спринту. Спринт повинен приймати ticket'и за допомогою методів add\*. Ці методи повертають true, коли вхідний ticket був прийнятий у спринт, і false в іншому випадку.

Створюємо клас Sprint та змінні capacity, ticketLimit і список tickets класу Ticket з ідентифікатором private. Його конструктор прийматиме два аргументи: capacity(ємність спринта) та ticketLimit(лімітів тікетів у спринті), у конструкторі визначаємо атрибути capacity, ticketLimit та tickets, якому призначаємо пустий список (для тікетів).

Реалізую метод getTickets(), який повертатиме копію списку tickets (тікетів) на спринт.

Тепер реалізуємо метод getTotalEstimate(), який буде повертати суму оцінок часу виконання всіх ticket'ів, прийнятих на спринт. Перебираємо кожен тікет у масиві tickets і отримуємо їхні оцінки та сумуємо їх. Повертаємо суму цих оцінок.

Реалізуємо метод addUserStory(), що приймає в якості аргументу userStory. Цей метод додаватиме тікет з юзерсторі до спринту. Для цього я реалізував перевірку, що якщо юзерсторі не дорівнює null і юзерсторі не є завершеною, і сума оцінки всіх тікетів плюс оцінка юзерсторі, яку я хочу додати до спринту, є менша ємності спринту, і кількість тікетів в списку tickets є меншою ніж, або дорівнює ліміту тікетів, то додати у список tickets userStory(юзерсторі) і повернути true. Інакше повернути false.

Реалізуємо метод addBug(), який в якості аргументу приймати об’єкт класу Bug. Цей метод додаватиме баг відповідно до певної юзерсторі, якщо про нього повідомлять. Для цього реалізується повністю аналогічна перевірка, як у методі addUserStory(), але тут об’єкт перевірки буде не юзерсторі, а саме баг, то ж скопіювавши, все замінюємо з userStory на bugReport.

## Реалізація Main. Перевірка роботи коду завдання

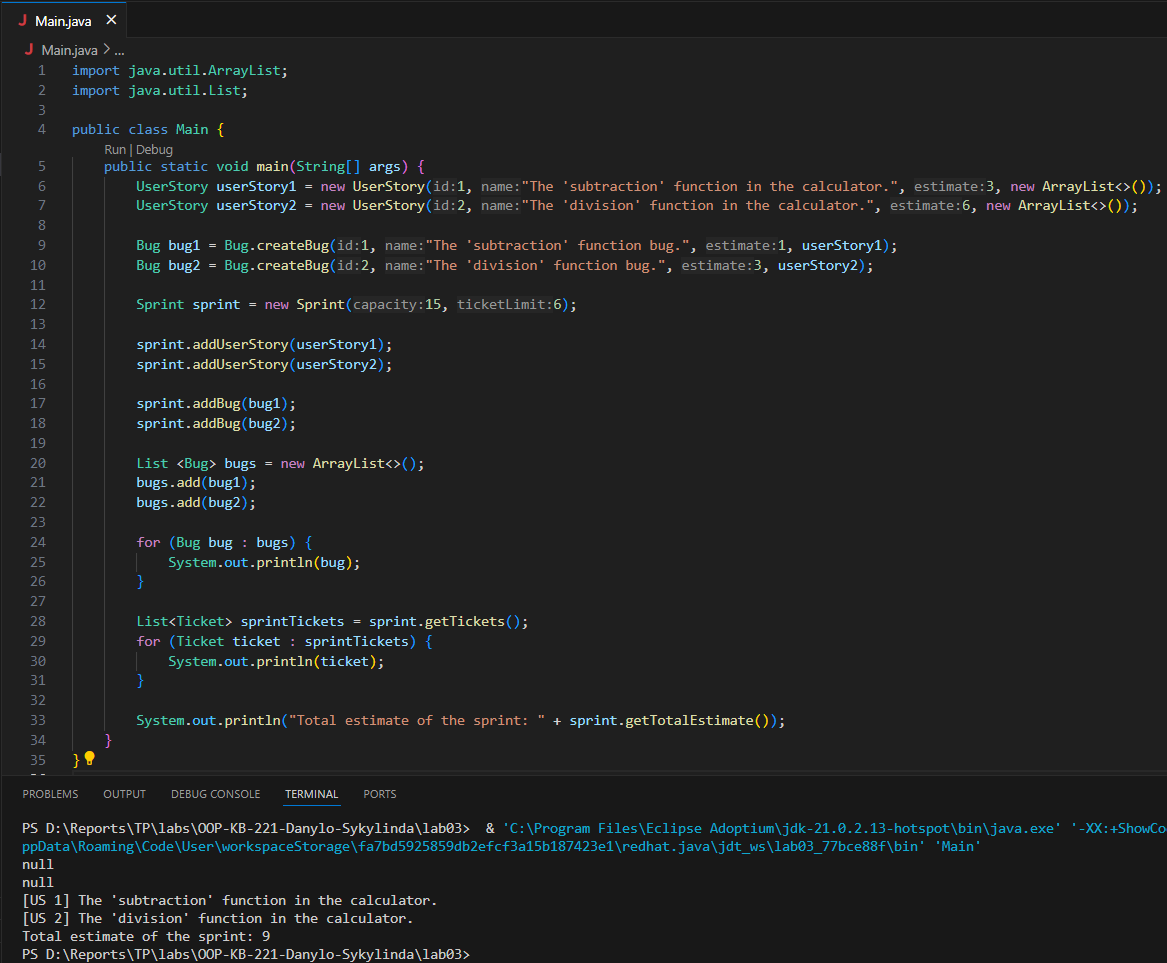


Рис. 2.5. Реалізація класу Main. На фото нижче перевірка роботи коду завдання.

Я створив по два об’єкти із таких класів як UserStory та Bug з відповідними аргументами. Потім створив об’єкт Sprint з відповідними аргументами. Пододавав у юзерсторі та баги в спрінт. Уточнення: спочатку я всі баги зібрав у масив. Потім я вивів усі баги, вивів усі тікети спринта. Як бачимо, два результати по багам дали null, оскільки перша та друга юзерсторі ще не завершені, і бачимо, що всі тікети були виведені по порядку. Також вивів загальну оцінку тікетів спринта. Тобто, все працює і було виконано правильно.

# ВИСНОВОК

Я ознайомився з матеріалом, який був поданий до цієї лабораторної роботи, в ньому я ознайомився з базовими принципами та концепціями ООП. Виконав завдання лабораторної роботи і на практиці засвоїв застосування об'єктно-орієнтованого програмування та класів, покращив свої навички у вирішенні задач. Завантажив всі виконані завдання та звіт лабораторної роботи на Github та Moodle.