**Лабораторна робота №3**

**Тема:** Базові принципи ООП

**Мета роботи**: Використовуючи теоретичне підґрунтя про об’єктно орієнтоване програмування виконати дії що будуть вказано в завданні до лабораторної роботи.

**Хід роботи:**

**Посилання на GitHub:** <https://github.com/VladislavVorona/OOP-KB222-Vladislav-Vorona>

**1. У цьому завданні ви формулюватимете бізнес-логіку планування спринту. [Спринт](https://en.wikipedia.org/wiki/Scrum_(software_development)" \l "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.**

**Main.java**

Спочатку я імпортував клас Scanner з пакету java.util, щоб мати можливість отримувати введення від користувача з консолі, далі я оголосив дві змінні типу int: capxx і ticksLim, які відповідно використовуватимуться для збереження обмежень спринту щодо кількості балів і тікетів. Далі розпочинається безкінечний цикл while, оскільки потрібно продовжувати отримувати введення від користувача до тих пір, поки введені значення не будуть коректними. Це реалізовано за допомогою конструкції try-catch, де в try блоку користувач вводить обмеження спринту, а в catch блоку перехоплюється помилка InputMismatchException, якщо користувач вводить значення не відповідного типу (наприклад, текст замість числа). Після перехоплення помилки виводиться повідомлення про некоректне значення та викликається метод nextLine(), щоб очистити введений рядок і дати користувачеві ще одну спробу. Цикл продовжується до тих пір, поки не будуть введені коректні значення, після чого він переривається за допомогою оператора break. Після отримання коректних обмежень спринту, я створюю новий об'єкт класу Sprint, передаючи йому отримані обмеження за допомогою конструктора класу. Наступні рядки виводять доступні команди для користувача, який може керувати програмою за допомогою цих команд. Всі команди виводяться у верхньому регістрі для спрощення обробки введення. Після виведення команд починається ще один цикл while, який також працює до тих пір, поки не буде викликана команда виходу з програми. В цьому циклі користувач вводить команди з консолі, які потім обробляються. Введене значення конвертується у верхній регістр за допомогою методу toUpperCase(), щоб уникнути проблем з різними регістрами при порівнянні команд. Далі використовується конструкція switch-case для обробки введених користувачем команд. Кожна команда має свій власний блок коду, який виконується залежно від того, яка саме команда була введена. Це реалізовано за допомогою оператора switch, який базується на значенні строки, введеної користувачем, та порівнює його з різними варіантами. У разі, якщо значення команди відповідає одному з варіантів, відповідний блок коду виконується. Перша команда, що обробляється, має ключ "A", що означає додавання нового тікету до спринту. Після введення команди "A" користувачеві пропонується ввести тип тікета: U для UserStory або B для Bug. Введене значення конвертується у верхній регістр для спрощення порівняння. Після цього зчитуються деталі тікета в залежності від типу. Якщо тип - UserStory, то зчитуються ID, назва і оцінка. Якщо тип - Bug, то зчитуються також ID, назва, оцінка, а також ID історії користувача, на яку посилається ця помилка. Після введення даних створюється відповідний об'єкт UserStory або Bug і він додається до спринту за допомогою методів addUserStory або addBug. Якщо додавання відбулося успішно, виводиться повідомлення про успішне додавання тікету, у протилежному випадку виводиться повідомлення про невдачу та вказується причина, наприклад те, що ID вже існує або історії користувача вже завершена. Команда "D" використовується для додавання або видалення залежностей між історіями користувача. Спочатку користувачу пропонується ввести ID історії користувача, для якої будуть управлятися залежності, потім програма знаходить в спринті відповідну історію користувача за заданим ID і якщо така історія користувача існує, користувачеві пропонується ввести ID залежності для додавання або видалення, ну або 0 для скасування операції. Після цього програма знаходить історію користувача за введеним ID і перевіряє, чи вона існує і якщо така історія користувача існує, перевіряється, чи вже існує залежність між ними, якщо немає, залежність додається, а якщо вже існує, залежність видаляється. Якщо введений ID залежності недійсний, виводиться відповідне повідомлення. Команда "C" використовується для позначення тікета, як завершеного. Користувачу пропонується ввести ID тікета, який він хоче позначити як завершений, після цього програма викликає метод completeTicket, передаючи йому введений ID, для позначення тікета як завершеного. Якщо операція виконана успішно, виводиться повідомлення про це. Далі йде команда "P", яка призначена для виведення інформації про всі тікети у спринті, разом з деякою статистикою. Спочатку створюється новий список sortTicks, який містить копії всіх тікетів з поточного спринту і він сортується за допомогою методу sort, що базується на ID тікетів. Після цього ініціалізуються змінні totalEst, remPoints, totalTicks, completTicks та remTicks, які використовуються для обчислення різних статистичних показників. Потім виконується цикл for-each, який проходиться по кожному тікету у відсортованому списку. Для кожного тікету виводиться інформація про нього, також виводяться всі пов'язані з ним помилки та залежності, якщо це UserStory, це реалізується шляхом перевірки, чи тікет є UserStory за допомогою оператора instanceof і якщо це UserStory, програма також шукає всі пов'язані з ним помилки і залежності та виводить їх. Після виведення інформації про кожен тікет виводиться роздільник та загальна статистика, яка включає в себе загальну оцінку спринту, бали, які залишилися, загальну кількість тікетів, кількість завершених тікетів та кількість досутпних тікетів. Кількість доступних тікетів для створення обчислюється шляхом віднімання кількості всіх тікетів від кількості завершених тікетів і відняттям цього значення від ліміту тікетів для спринту. Команда "H" виводить на екран список доступних команд разом з їх поясненнями, це робиться шляхом виведення відповідних рядків тексту, які містять опис кожної команди. Команда "X" встановлює змінну xx в логічне значення false, що призводить до завершення головного циклу програми і виходу з неї. Якщо введена команда не існує, то виводиться повідомлення про помилку і користувачеві пропонується ввести команду знову. Блок try-catch використовується для перехоплення помилок, які можуть виникнути при введенні користувачем команди, у випадку, якщо користувач введе нечислове значення там, де очікується число, виведеться відповідне повідомлення про помилку. Після завершення виконання програми метод close() викликається для об'єкта Scanner, щоб звільнити ресурси, пов'язані з введенням з консолі. В кінці цього класу я додав метод findUS, який використовується для пошуку історії користувача за його ID у списку тікетів. Метод отримує два параметри: список тікетів ticks і відповідне ID id. Він проходиться по кожному тікету у списку ticks за допомогою циклу for-each. Для кожного тікету перевіряється, чи є він екземпляром класу UserStory та чи його ID співпадає з введеним ID, якщо обидві умови виконуються, метод повертає цей тікет, як об'єкт класу UserStory. Якщо ж жоден з тікетів не відповідає заданому ID, метод повертається значення null, що означає, що історію користувача з таким ID не знайдено у списку.

|  |
| --- |
| import java.util.\*;    public class Main {  public static void main(String[] args) {  Scanner inpScn = new Scanner(System.in);  int capxx = 0;  int ticksLim = 0;    while (true) {  try {  System.out.println("Введіть обмеження спринту");  System.out.println("Доступна кількість балів:");  capxx = inpScn.nextInt();  System.out.println("Доступна кількість тікетів:");  ticksLim = inpScn.nextInt();  inpScn.nextLine();  break;  } catch (InputMismatchException e) {  System.out.println("Було введено некоректне значення, спробуйте ще раз.");  inpScn.nextLine();  }  }    Sprint sprint = new Sprint(capxx, ticksLim);    System.out.println("Команди:");  System.out.println("A: Додати тікет");  System.out.println("C: Позначити тікет, як завершений");  System.out.println("D: Додати залежність тікету");  System.out.println("P: Вивести всі тікети");  System.out.println("H: Показати доступні команди");  System.out.println("X: Вийти");    boolean xx = true;  while (xx) {  System.out.print("\nВведіть команду: ");  String command = inpScn.nextLine().toUpperCase();    try {  switch (command) {  case "A":  System.out.print("Введіть тип тікета (U для UserStory, B для Bug): ");  String type = inpScn.nextLine().toUpperCase();  if (type.equals("U")) {  System.out.println("Введіть деталі історії (ID, назва, оцінка):");  int id = inpScn.nextInt();  String name = inpScn.next();  int estimate = inpScn.nextInt();  inpScn.nextLine();  UserStory ustory = new UserStory(id, name, estimate);  if (sprint.addUserStory(ustory)) {  System.out.println("Історію користувача додано: " + ustory);  } else {  System.out.println("Не вдалося додати історію користувача. Перевірте, чи існує ID історії користувача або чи вона вже завершена.");  }  } else if (type.equals("B")) {  System.out.println("Введіть деталі помилки (ID, назва, оцінка, ID історії користувача):");  int id = inpScn.nextInt();  String name = inpScn.next();  int estimate = inpScn.nextInt();  int usid = inpScn.nextInt();  inpScn.nextLine();  UserStory ustory = findUS(sprint.gtTickets(), usid);  if (ustory != null) {  Bug bug = Bug.createBug(id, name, estimate, ustory);  if (bug != null && sprint.addBug(bug)) {  System.out.println("Помилку додано: " + bug);  } else {  System.out.println("Не вдалося додати баг. Перевірте, чи існує ID помилки, ID історії користувача, або чи історію користувача вже завершено.");  }  } else {  System.out.println("Історію користувача з ID " + usid + " не знайдено.");  }  } else {  System.out.println("Неправильний тип тікета. Будь ласка, спробуйте знову.");  }  break;  case "D":  System.out.print("Введіть ID історії користувача для управління залежностями: ");  int usid = inpScn.nextInt();  inpScn.nextLine();  UserStory targtUserStory = findUS(sprint.gtTickets(), usid);  if (targtUserStory != null) {  System.out.print("Введіть ID залежності для додавання або видалення (0 для скасування): ");  int dependId = inpScn.nextInt();  inpScn.nextLine();  if (dependId != 0) {  UserStory depend = findUS(sprint.gtTickets(), dependId);  if (depend != null) {  if (!targtUserStory.gtdepends().contains(depend)) {  targtUserStory.addDepend(depend);  System.out.println("Залежність додано: " + depend);  } else {  targtUserStory.remDepend(depend);  System.out.println("Залежність видалено: " + depend);  }  } else {  System.out.println("Історію користувача з ID " + dependId + " не знайдено.");  }  }  } else {  System.out.println("Історію користувача з ID " + usid + " не знайдено.");  }  break;  case "C":  System.out.print("Введіть ID тікета для позначення, як завершений: ");  int ticketId = inpScn.nextInt();  inpScn.nextLine();  if (sprint.completeTicket(ticketId)) {  System.out.println("Тікет з ID " + ticketId + " позначено, як завершений.");  }  break;  case "P":  List<Ticket> sortTicks = new ArrayList<>(sprint.gtTickets());  sortTicks.sort(Comparator.comparingInt(Ticket::gtId));  int totalEst = 0;  int remPoints = sprint.gtCapacity() - sprint.gtTotalEst();  int totalTicks = 0;  int completTicks = 0;  int remTicks = sprint.gtTickLimit();    for (Ticket ticket : sortTicks) {  totalTicks++;    if (ticket.isCompleted()) {  completTicks++;  }    if (ticket instanceof UserStory) {  System.out.println(ticket);  UserStory ustory = (UserStory) ticket;  List<Bug> bugs = new ArrayList<>();  List<UserStory> depends = ustory.gtdepends();  for (Ticket t : sortTicks) {  if (t instanceof Bug && ((Bug) t).gtUStory().gtId() == ustory.gtId()) {  bugs.add((Bug) t);  }  }  bugs.sort(Comparator.comparingInt(Bug::gtId));  for (Bug bug : bugs) {  System.out.println(" " + bug);  }  for (UserStory depend : depends) {  System.out.println(" Залежність: " + depend);  List<Bug> dependBugs = new ArrayList<>();  for (Ticket t : sortTicks) {  if (t instanceof Bug && ((Bug) t).gtUStory().gtId() == depend.gtId()) {  dependBugs.add((Bug) t);  }  }  dependBugs.sort(Comparator.comparingInt(Bug::gtId));  for (Bug xxBug : dependBugs) {  System.out.println(" " + xxBug);  }  }  }  }  System.out.println("----------------------------------");  System.out.println("Загальна оцінка: " + sprint.gtTotalEst());  System.out.println("Залишилось балів: " + remPoints);  System.out.println("Загальна кількість тікетів: " + totalTicks);  System.out.println("Завершені тікети: " + completTicks);  remTicks -= totalTicks - completTicks;  System.out.println("Залишилося тікетів для створення: " + remTicks);  break;    case "H":  System.out.println("Команди:");  System.out.println("A: Додати тікет");  System.out.println("C: Позначити тікет, як завершений");  System.out.println("D: Додати залежність тікету");  System.out.println("P: Вивести всі тікети");  System.out.println("H: Показати доступні команди");  System.out.println("X: Вийти");  break;  case "X":  xx = false;  break;  default:  System.out.println("Невірна команда, спробуйте знову.");  }  } catch (InputMismatchException e) {  System.out.println("Неправильний ввід, потрібно вводити число.");  }  }    inpScn.close();  }    private static UserStory findUS(List<Ticket> ticks, int id) {  for (Ticket ticket : ticks) {  if (ticket instanceof UserStory && ticket.gtId() == id) {  return (UserStory) ticket;  }  }  return null;  }  } |

**Ticket.java**

В цьому файлі я спочатку визначив клас Ticket, який містить наступні поля: id - цілочисельне значення, що ідентифікує тікет, name - рядок, що містить назву тікета, estimate - цілочисельне значення, що представляє оцінку тікету, completed - логічне значення, яке показує, чи завершений тікет. Конструктор класу приймає три параметри: id, name і estimate та ініціалізує відповідні поля, поле completed при цьому завжди встановлюється в false, оскільки по замовчуванню тікети не завершені. Клас також містить методи доступу до полів, такі, як gtId, gtName і gtEstimate для зчитування значень цих полів. Метод isCompleted повертає значення поля completed, що дозволяє отримати статус тікета. Метод complete встановлює значення поля completed в true, щоб позначити тікет, як виконаний.

|  |
| --- |
| import java.util.\*;    class Ticket {  protected int id;  private String name;  private int estimate;  private boolean completed;    public Ticket(int id, String name, int estimate) {  this.id = id;  this.name = name;  this.estimate = estimate;  this.completed = false;  }    public int gtId() {  return id;  }    public String gtName() {  return name;  }    public int gtEstimate() {  return estimate;  }    public boolean isCompleted() {  return completed;  }    public void complete() {  completed = true;  }  } |

**UStory.java**

Клас UserStory унаслідується від класу Ticket і представляє собою об'єкти історії користувачів. У конструкторі класу UserStory наслідувані поля, такі, як id, name і estimate, ініціалізуються за допомогою конструктора батьківського класу Ticket. Поля bugs та depends ініціалізуються порожніми списками, що представляють список пов'язаних помилок та список залежностей User Story від інших User Story. Методи gtBugs і gtdepends повертають копії відповідних списків, щоб запобігти змінам вихідних даних клієнтами класу. Методи addBug і addDepend додають нові елементи до списків bugs та depends відповідно, якщо переданий об'єкт не є нульовим та не є виконаним. Метод remDepend видаляє вказаний об'єкт зі списку залежностей User Story. Метод toString перевизначений для надання текстового представлення User Story, яке містить ідентифікатор, назву історії, її оцінку та статус виконання.

|  |
| --- |
| import java.util.\*;    class UserStory extends Ticket {  private List<Bug> bugs;  private List<UserStory> depends;    public UserStory(int id, String name, int estimate) {  super(id, name, estimate);  this.bugs = new ArrayList<>();  this.depends = new ArrayList<>();  }    public List<Bug> gtBugs() {  return new ArrayList<>(bugs);  }    public boolean addBug(Bug bug) {  if (bug != null && !bug.isCompleted()) {  bugs.add(bug);  return true;  }  return false;  }    public void addDepend(UserStory depend) {  if (depend != null) {  depends.add(depend);  }  }    public void remDepend(UserStory depend) {  depends.remove(depend);  }    public List<UserStory> gtdepends() {  return new ArrayList<>(depends);  }    @Override  public String toString() {  return "[US " + id + "] Назва: " + gtName() + ", Оцінка: " + gtEstimate() + ", Завершено: " + isCompleted();  }  } |

**Bugs.java**

В цьому файлі визначається клас Bug, який є підкласом класу Ticket і представляє собою об'єкти "Bug". У класі Bug є додаткове приватне поле ustory, яке вказує на історію користувача, до якої прив'язана ця помилка, це поле ініціалізується в конструкторі класу Bug, який приймає параметри id, name, estimate та ustory. Поле ustory використовується для збереження посилання на об'єкт історії користувача, до якого відноситься ця помилка. Метод createBug є статичним і використовується для створення нового об'єкта класу Bug, він перевіряє, чи історія користувача, до якої прикріплюється ця помилка, не є виконаною і якщо це так, метод створює новий об'єкт Bug і повертає його, у протилежному випадку метод повертає null, що вказує на неможливість створення об'єкту Bug. Метод gtUStory повертає об'єкт історії користувача, до якого прикріплена ця помилка. Метод toString визначений для надання текстового представлення багу, яке містить ідентифікатор, назву, оцінку, ID історії користувача, до якої вона відноситься, та статус виконання.

|  |
| --- |
| import java.util.\*;    class Bug extends Ticket {  private UserStory ustory;    private Bug(int id, String name, int estimate, UserStory ustory) {  super(id, name, estimate);  this.ustory = ustory;  }    public static Bug createBug(int id, String name, int estimate, UserStory ustory) {  if (ustory != null && !ustory.isCompleted()) {  return new Bug(id, name, estimate, ustory);  }  return null;  }    public UserStory gtUStory() {  return ustory;  }    @Override  public String toString() {  return "[BUG " + id + "] Назва: " + gtName() + ", Оцінка: " + gtEstimate() + ", ID історії користувача: " + ustory.gtId() + ", Завершено: " + isCompleted();  }  } |

**Sprint.java**

У цьому файлі визначається клас Sprint, у якому є приватні поля: capacity - це числове значення, яке визначає максимальну кількість балів, яку можна витратити протягом спринту. ticksLimit - це ціле число, що визначає максимальну кількість тікетів, які можна додати до спринту. ticks - це список, який зберігає усі тікети, що включені до поточного спринту. Конструктор класу Sprint приймає два аргументи: capacity та ticksLimit, ініціалізує поля capacity та ticksLimit відповідними значеннями, а поле ticks ініціалізується порожнім списком. Метод addUserStory() використовується для додавання нової історії користувача до поточного спринту. Спочатку метод обчислює кількість незавершених тікетів у поточному спринті за допомогою методу calcNoComplTicks(), а потім він розраховує загальну оцінку усіх тікетів у спринті за допомогою методу gtTotalEst(). Перевіряється, чи історія користувача не є нульовим об'єктом, чи вона не існує вже у спринті, чи вона не є завершеною, чи кількість незавершених тікетів у спринті не перевищує ліміту і чи загальна оцінка тікетів у спринті, разом із новою історією користувача, не перевищує загальної місткості спринту. Якщо всі перевірки пройдені успішно історія користувача додається до списку тікетів спринту і метод повертає true, а якщо якуйсь із перевірок не було пройдено - false. Цей клас дозволяє додавати нові тікети до спринту та контролювати, щоб їхня загальна оцінка та кількість не перевищували визначені ліміти. Метод addBug() призначений для додавання нового багу до поточного спринту. Він спочатку обчислює кількість незавершених тікетів у поточному спринті за допомогою приватного методу calcNoComplTicks(), потім він обчислює загальну оцінку усіх тікетів у спринті, включаючи нову помилку, за допомогою методу gtTotalEst(). Перевіряється, чи об'єкт bug не є нульовим, чи він не існує вже у спринті, чи він не є завершеним, чи кількість незавершених тікетів у спринті не перевищує ліміту і чи загальна оцінка тікетів у спринті, разом із новою помилкою, не перевищує загальної місткості спринту. Якщо всі перевірки пройдені успішно, помилка додається до списку тікетів спринту і метод повертає true, а якщо хоча б якуйсь перевірку не було пройдено - false. Метод gtCapacity() є публічним і повертає максимальну кількість балів, яку можна витратити протягом поточного спринту. Приватний метод calcNoComplTicks() використовується для обчислення кількості незавершених тікетів у поточному спринті, він перебирає всі тікети у списку ticks і якщо тікет не завершений, збільшує лічильник. Після завершення перебору метод повертає обчислену кількість незавершених тікетів. Метод gtTickLimit() є публічним і повертає максимальну кількість тікетів, які можна додати до поточного спринту. Метод completeTicket() призначений для позначення тікета з вказаним ідентифікатором, як виконаний у поточному спринті. Він перебирає всі тікети у списку ticks і знаходить тікет з ідентифікатором, який відповідає переданому ticketId. Якщо такий тікет знайдено і він не є завершеним, метод перевіряє, чи є цей тікет історією користувача, якщо так, він перевіряє, чи всі залежності цієї історії користувача завершені, якщо якась залежність не завершена, метод повертає false і виводить повідомлення про невдалу спробу завершення. Після цього метод перевіряє, чи є в поточному спринті невирішені помилки, які залежать від цієї історії користувача і якщо такі помилки існують, метод також повертає false і виводить відповідне повідомлення. Якщо всі перевірки пройшли успішно, тікет позначається як завершений за допомогою методу complete() і метод повертає true. Метод gtTickets() є публічним і повертає копію списку всіх тікетів у поточному спринті, це зроблено для того, щоб забезпечити безпечний доступ до списку тікетів із зовнішніх частин програми, щоб уникнути його непередбаченої змінени. Метод completeTicket(int ticketId) призначений для позначення тікета з вказаним ідентифікатором, як виконаний у поточному спринті. Спочатку метод перебирає всі тікети у списку ticks та знаходить тікет з ідентифікатором, який відповідає переданому ticketId. Якщо такий тікет знайдено і він не є завершеним, метод перевіряє, чи є цей тікет історією користувача. Якщо так, він перевіряє, чи всі залежності цієї історії користувача завершені. Якщо хоча б одна залежність не завершена, метод повертає false та виводить повідомлення про невдалу спробу завершення. Після цього метод перевіряє, чи є в поточному спринті невирішені помилки, які залежать від цієї історії користувача, якщо такі помилки існують, метод також повертає false та виводить відповідне повідомлення. Якщо всі перевірки пройшли успішно, тікет позначається, як виконаний за допомогою методу complete() і метод повертає true. Метод gtTickets() є публічним і повертає копію списку всіх тікетів у поточному спринті. Такий підхід забезпечує безпечний доступ до списку тікетів ззовні класу, щоб уникнути його непередбаченої зміни. Метод gtTotalEst() призначений для обчислення загальної оцінки, яка відображає загальну кількість балів, необхідних для завершення всіх тікетів у поточному спринті. Він ітерується по списку всіх тікетів у спринті та додає оцінку кожного тікета до загальної оцінки, якщо тікет не завершено. Якщо тікет є історією користувача і він незавершений, метод також додає до загальної оцінки оцінку всіх пов'язаних з ним невиконаних помилок. Після обчислення загальної оцінки метод повертає її. Метод exists() є приватним та призначений для перевірки наявності певного тікета у списку тікетів у поточному спринті. Він перебирає усі тікети у списку та порівнює їх ідентифікатори з ідентифікатором переданого тікета. Якщо знайдений тікет з ідентичним ідентифікатором, метод повертає true, що свідчить про те, що такий тікет вже існує у списку, у протилежному випадку метод повертає false, що означає, що тікета з таким ідентифікатором немає у списку.

|  |
| --- |
| import java.util.\*;    class Sprint {  private int capacity;  private int ticksLimit;  private List<Ticket> ticks;    public Sprint(int capacity, int ticksLimit) {  this.capacity = capacity;  this.ticksLimit = ticksLimit;  this.ticks = new ArrayList<>();  }    public boolean addUserStory(UserStory ustory) {  int noComplete = calcNoComplTicks();  int totalEst = gtTotalEst() + ustory.gtEstimate();  if (ustory != null && !exists(ustory) && !ustory.isCompleted() && noComplete < ticksLimit && totalEst <= capacity) {  ticks.add(ustory);  return true;  }  return false;  }    public boolean addBug(Bug bug) {  int noComplete = calcNoComplTicks();  int totalEst = gtTotalEst() + bug.gtEstimate();  if (bug != null && !exists(bug) && !bug.isCompleted() && noComplete < ticksLimit && totalEst <= capacity) {  ticks.add(bug);  return true;  }  return false;  }    public int gtCapacity() {  return capacity;  }    private int calcNoComplTicks() {  int count = 0;  for (Ticket ticket : ticks) {  if (!ticket.isCompleted()) {  count++;  }  }  return count;  }    public int gtTickLimit() {  return ticksLimit;  }    public boolean completeTicket(int ticketId) {  for (Ticket ticket : ticks) {  if (ticket.gtId() == ticketId && !ticket.isCompleted()) {  UserStory ustory = null;  if (ticket instanceof UserStory) {  ustory = (UserStory) ticket;  }  if (ustory != null) {  List<UserStory> depends = ustory.gtdepends();  for (UserStory depend : depends) {  if (!depend.isCompleted()) {  System.out.println("Не вдалося завершити історію користувача з ID " + ticketId + ", оскільки є невирішені залежності.");  return false;  }  }  }  for (Ticket t : ticks) {  if (t instanceof Bug && !t.isCompleted() && ((Bug) t).gtUStory().gtId() == ticketId) {  System.out.println("Не вдалося завершити історію користувача з ID " + ticketId + ", оскільки є невирішені помилки.");  return false;  }  }  ticket.complete();  return true;  }  }  System.out.println("Не вдалося знайти тікет з ID " + ticketId + ", або він вже завершений.");  return false;  }    public List<Ticket> gtTickets() {  return new ArrayList<>(ticks);  }    public int gtTotalEst() {  int totalEst = 0;  for (Ticket ticket : ticks) {  if (!ticket.isCompleted()) {  totalEst += ticket.gtEstimate();  }  if (ticket instanceof UserStory && ticket.isCompleted()) {  UserStory ustory = (UserStory) ticket;  List<Bug> bugs = ustory.gtBugs();  for (Bug bug : bugs) {  if (!bug.isCompleted()) {  totalEst += bug.gtEstimate();  }  }  }  }  return totalEst;  }    private boolean exists(Ticket ticket) {  for (Ticket t : ticks) {  if (t.gtId() == ticket.gtId()) {  return true;  }  }  return false;  }  } |

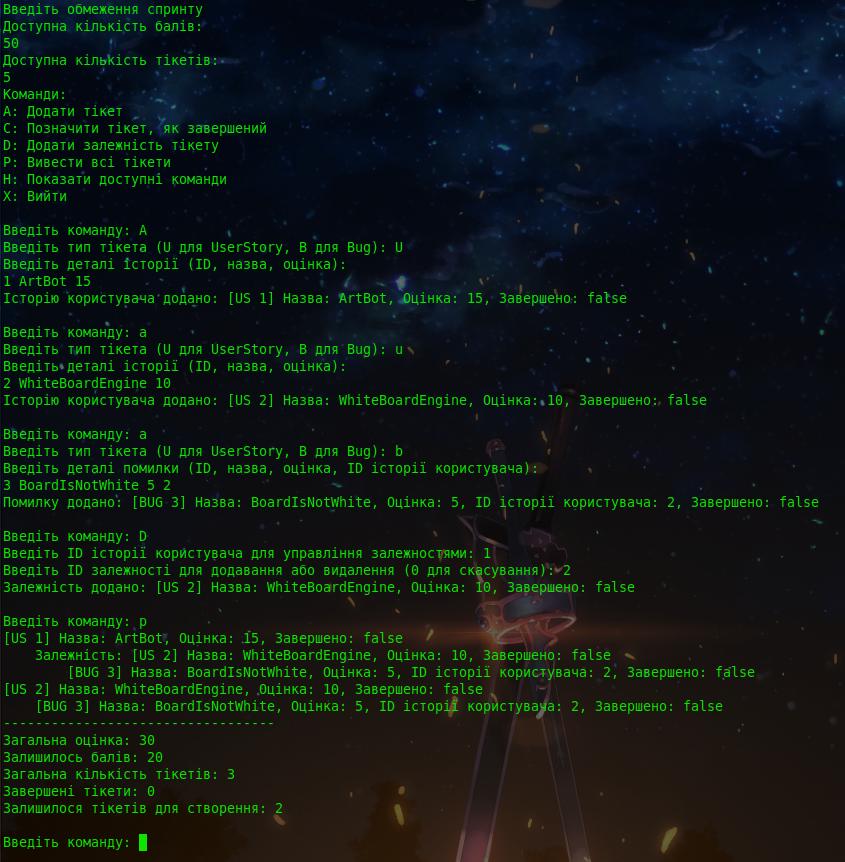
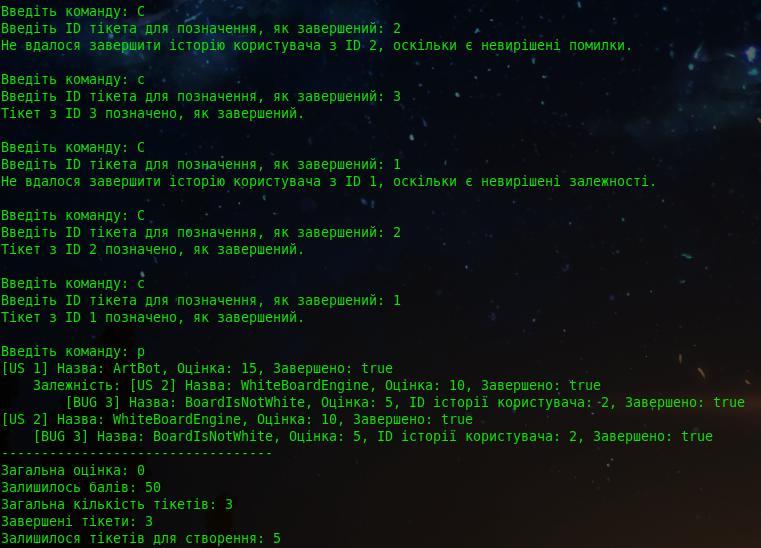


Рисунок 1 - Приклад роботи програми

Рисунок 2 - Приклад роботи програми

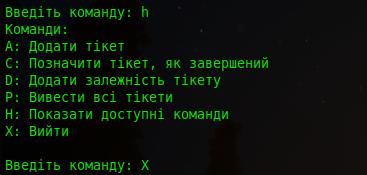


Рисунок 3 - Приклад роботи програми

**Висновок:** Під час виконання цієї лабораторної роботи я покращив свої знання про базові принципи об'єктно-орієнтованого програмування та їх використання в мові програмування Java. За допомогою цієї лабораторної роботи я закріпив теоретичні знання, а також покращив свої навички застосовування їх на практиці, розробляючи програмне забезпечення для планування спринтів у методології розробки Scrum. Під час виконання лабораторної роботи я краще засвоїв основні поняття об'єктно-орієнтованого програмування, такі як класи, об'єкти, спадкування та інкапсуляція. Також під час розробки програми для планування спринтів я навчився краще працювати з класами, створювати конструктори, методи та змінні класів, а також реалізувавати різні методи для взаємодії з об'єктами. Також я покращив свої знання про використання списків для зберігання та керування об'єктами у Java. Загалом, у результаті виконання цієї лабораторної роботи я покращив свої навички програмування на мові Java та навчився краще працювати із об'єктами, класами та колекціями.