Лабораторная работа 2

по курсу Технология программирования

на тему «Объектно-ориентированное программирование в Java»

Составил: Питикин А.Р.

кафедра ИУ-3

Москва, 2024

Содержание

Теоретическая часть

Java – объектно-ориентированный язык, поэтому писать программы необходимо с учетом ООП. Для начала определимся:

Объектно-ориентированное программирование (ООП) — это методология программирования с использованием объектов и классов. Объект характеризует состояние и поведение. Например, у собаки есть такие свойства, как кличка, порода; собака может бегать, спать и есть. На UML диаграмме класс выглядит так:

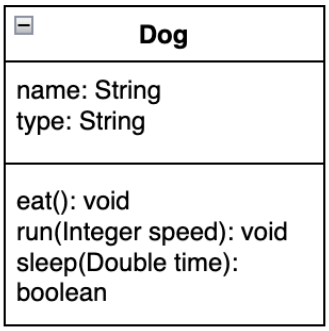


Рисунок 1 – диаграмма класса Dog.

Определимся сразу с плюсами и минусами подхода ООП.

Какие преимущества у ООП?

1. Легко читается – не нужно выискивать в коде функции и выяснять, за что они отвечают.
2. Быстро пишется – можно быстро создать сущности, с которыми должна работать программа.
3. Простота реализации большого функционала – т. к. на написание кода уходит меньше времени, можно гораздо быстрее создать приложение с множеством возможностей.

Какие недостатки у ООП?

1. Потребление памяти – объекты потребляют больше оперативной памяти, чем примитивные типы данных.

2. Снижается производительность – многие вещи технически реализованы иначе, поэтому они используют больше ресурсов.
3. Сложно начать – парадигма ООП сложнее функционального программирования, поэтому на старт уходит больше времени.

Основными принципами ООП являются:

1. Абстракция

2. Инкапсуляция

3. Полиморфизм

4. Наследование

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании. По-другому это называется скрытием данных.

Для достижения инкапсуляции в Java:

1. Объявите переменные класса как private.
2. Предоставьте public к методам установки и получения (сеттеру и геттеру) для изменения и просмотра значений переменных.

Задача программиста — определить, какие атрибуты и методы будут доступны для открытого доступа, а какие являются внутренней реализацией объекта и должны быть недоступны для изменений.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий.

Наследование — это механизм, который позволяет создавать классы на основе других классов. Это дает возможность расширять свойства наследуемого класса, и сохранять работоспособность ранее написанного кода.

- Все классы в мире java, косвенно или прямо, являются наследниками класса Object.
- Наследоваться можно только от одного класса.

Для того чтобы некий класс стал наследником другого класса, в его объявлении применяется ключевое слово **extends** и имя наследуемого класса.

Наследование, необходимо для того, чтобы не дублировать ранее написанный код, а использовать его же, но с новыми возможностями.

Определимся терминологией разработки.

Класс в Java – это шаблон для создания объекта, а **объект –** это экземпляр класса. Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Класс определяет структуру и поведение, которые будут совместно использоваться набором объектов. Класс содержит переменные и методы, которые называются элементами класса, членами класса. **Методы** используются для описания того, что объект класса умеет делать или что можно с ним сделать. **Переменные** – для описания свойств или характеристик объекта.

В Java почти все сущности являются объектами, за исключением примитивных типов. У каждого объекта есть класс. Сами классы тоже является объектами, и они принадлежат классу Class.

Переопределение метода в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую

реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

Чтобы разобраться, возьмем родительский класс **Animal**, обозначающий животных, и создадим в нем метод *voice()* — «голос»:

Животных в мире очень много, и все «говорят» по-разному: собаки гавкают, кошки мяукают, утки крякают, змеи шипят.

Вместо того, чтобы создавать методы voiceCat() для мяуканья, voiceSnake() для шипения и т.д., мы хотим, чтобы при вызове метода voice() змея шипела, кошка мяукала, а собака лаяла. Мы легко добьемся этого с помощью механизма переопределения методов (аннотация **@Override** в Java).

Чтобы задать нужное нам поведение, мы сделали несколько вещей:

1. Создали в каждом классе-наследнике метод с таким же названием, как и у метода в родительском классе.

2. Сообщили компилятору, что мы не просто так назвали метод так же, как в классе-родителе: хотим переопределить его поведение. Для этого мы поставили над методом аннотацию @Override («переопределен»).

3. Написали нужную нам реализацию для каждого класса-потомка. Змея при вызове voice() должна шипеть, медведь — рычать и т.д.

У переопределения есть ряд ограничений:

1. У переопределенного метода должны быть те же аргументы, что и у метода родителя.

2. У переопределенного метода должен быть тот же тип возвращаемого значения, что и у метода родителя.

3. Модификатор доступа у переопределенного метода также не может отличаться от «оригинального»:

В программе мы можем использовать методы с одним и тем же именем, но с разными типами и/или количеством параметров. Такой механизм называется **перегрузкой** методов (method overloading). Стоит отметить, что на перегрузку методов влияют количество и типы параметров. Однако различие в типе возвращаемого значения для перегрузки не имеют никакого значения.

Очень рекомендую для каждого класса переопределить метод toString, реализующий стандартное представление объекта в виде строки. Это поможет вам при тестировании. Например, для класса Car это может выглядеть так:

```
public class Car {
    private double speed;
    private String name;
    private String owner;


    @Override
    public String toString(){
        return "Speed = " + speed + "Name = " + name + "owner = " + owner;
    }
}
```

Модификаторы доступа — это ключевые слова, которые регулируют уровень доступа к разным частям кода.

- **private** (приватный): члены класса доступны только внутри класса. Для обозначения используется служебное слово **private**.
- **default**, (доступ на уровне пакета): видимость класса/членов класса только внутри пакета. Является модификатором доступа по умолчанию – специальное обозначение не требуется.
- **protected** (защищенный): члены класса доступны внутри пакета и в наследниках. Для обозначения используется служебное слово **protected**.
- **public** (публичный): класс/члены класса доступны всем. Для обозначения используется служебное слово **public**.

Последовательность модификаторов по возрастанию уровня закрытости: public, protected, default, private. Во время наследования возможно изменения модификаторов доступа в сторону большей видимости/

Класс может быть объявлен с модификатором public и default.

Для удобства работы принято делить все используемые классы по разным пакетам. Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета:

```
package my.edu.projectone.service.search;
```

За счет пакетов структуру проекта несложно проанализировать, и легко найти нужный класс.

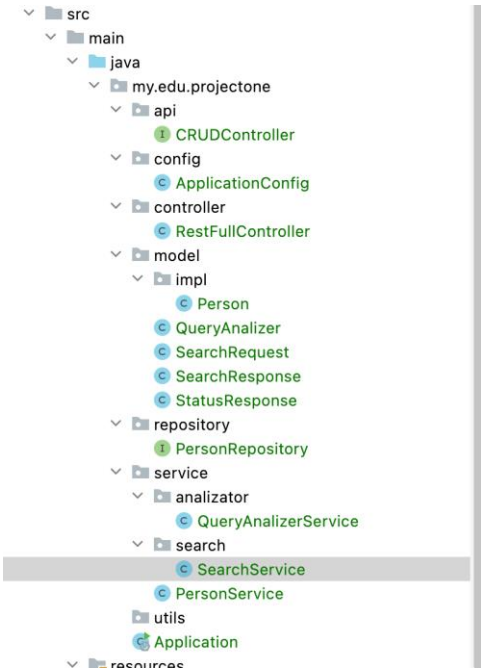


Рисунок 2 – пример структуры проекта с использованием распределения на пакеты.

Практическая часть

«В уездном городе N было так много парикмахерских заведений и бюро похоронных процессий, что казалось, жители города рождаются лишь затем, чтобы побриться, остричься, освежить голову вежеталем и сразу же умереть.»

Ильф и Петров, «Двенадцать стульев»

В рамках этой лабораторной работы предлагается разработка небольшой игры стратегии в сеттинге средневековья, чем-то напоминающей DnD или же нашумевший Baldur`s Gate. Так и назовем – Bauman’s Gate. Т.к. технологий графического интерфейса, таких, как JavaFX или Swing мы еще не касались в нашем курсе, в работе речь пойдет о консольном приложении.

Целью данной работы является закрепление на практике теоретического материала об основных идеях ООП, полученного ранее. Результатом ЛР будет являться прототип некоторой консольной игры.

Введение

Пользователь является руководителем некоторого, условно средневекового, Города N. В рамках ЛР2 основной задачей является обеспечение безопасности города. Для этого пользователю предлагается нанимать различных юнитов, улучшать их и непосредственно сражаться с противником. Также, в рамках индивидуальных заданий речь пойдет о разработке различных событий, так или иначе влияющих на боеготовность юнитов.

Используемые обозначения

Данная лабораторная работа весьма творческая – единственно верного, «идеального» решения для нее не существует, собственно, как и для любой задачи на ООП. Однако, чтобы «свободы» было немного меньше, введем некоторые требования. Требования будут двух видов: бизнес-требования и технические требования.

Глоссарий предметной области

В данном разделе договоримся об общих используемых понятиях в рамках данной лабораторной работы, а также в рамках лабораторной работы 3.

Событие – некоторое мероприятие, направленное на улучшение или ухудшение боеготовности юнитов

Город – местность, в которой происходят все События

Здание – некоторый объект в Городе, необходимый для некоторых Событий

Юнит – минимальная единица, участвующая в Сражении

Стоимость – количество денег, которые необходимо отдать за найм Юнита

Оружие – предмет, принадлежащий Юниту и определяющий тип урона, который Юнит может нанести

Животное – существо, используемое для перемещения некоторых типов Юнитов

Атака – свойство Юнита, указывающее на размер урона, который он может нанести

Защита – свойство Юнита, указывающее на размер урона, который он может поглотить

Местность – характеристика клетки Поле битвы, вид, влияющий на Штраф к перемещению Юнитов.

Поле битвы – поле в формате «мир-сетка», состоящее из Местностей, на котором происходит Сражение двух противников

Перемещение – сущность, показывающая перемещение Юнита и набор штрафов, если таковые есть

Штраф перемещения – сущность, показывающая штраф на перемещение по особому типу Местности

Сражение – основной процесс игры, происходящий на Поле битвы, заканчивается тогда, когда у одного из противников заканчиваются Юниты.

Бот – алгоритм компьютера, против которого играет пользователь.

Задание

Юниты

Глобально, существует 3 типа юнитов – Пешие, Лучники и Всадники. В таблицах 1-3 представлены типы Юнитов и основные различия между ними. В таблицах ниже представлены примерные характеристики, вы можете использовать другие, но с сохранением пропорций(например, если в таблице у Мечника атака 3, а у Топорщика 5, то вы можете использовать любые числа, но с сохранением того, что атака Топорщика больше атаки Мечника).

Таблица 1. Пешие юниты

Название	Здоровье	Атака	Дальность атаки	Защита	Перемещение	Стоимость
Мечник	50	5	1	8	3	10
Копьеносец	35	3	1	4	6	15
Топорщик	45	9	1	3	4	20

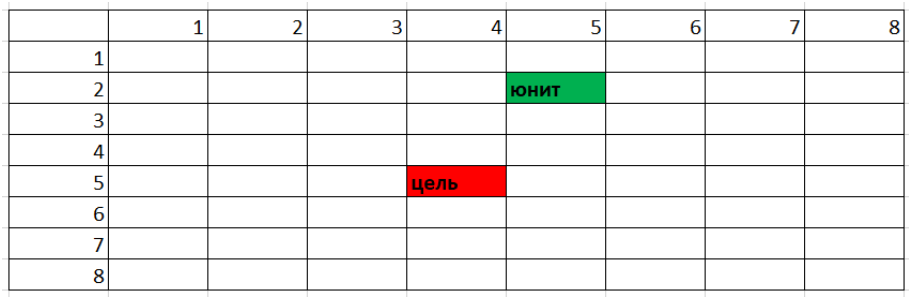
Таблица 2. Лучники

Название	Здоровье	Атака	Дальность атаки	Защита	Перемещение	Стоимость
Лучник(дл. лук)	30	6	5	8	2	15
Лучник(кор. лук)	25	3	3	4	4	19
Арбалетчик	40	7	6	3	2	23

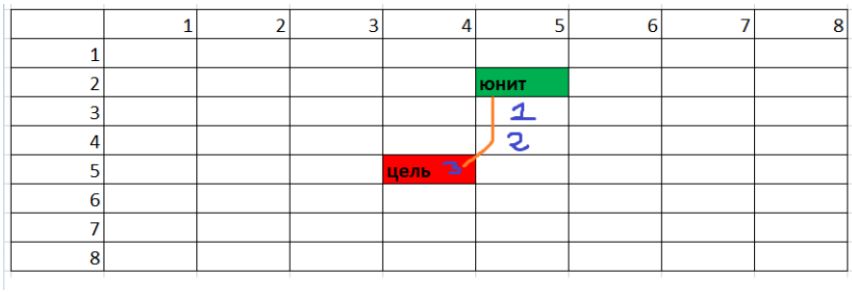
Таблица 3. Всадники

Название	Здоровье	Атака	Дальность атаки	Защита	Перемещение	Стоимость
Рыцарь	30	5	1	3	6	20
Кирасир	50	2	1	7	5	23
Конный лучник	25	3	3	2	5	25

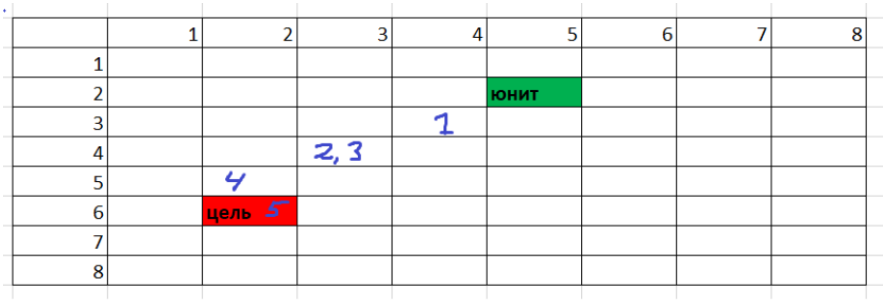
Примечание 1 – логика расчета дальности атаки лучников остается на усмотрение студента. Можно использовать округление от Евклидова расстояния между двумя точками, можно использовать систему, аналогичную классическим НРИ – клетки считаются за 1, каждая вторая диагональная клетка считается за 2, можно любой другой способ. Логика расчета перемещения такая же. Приведем пример логики расчета аналогичную классическим НРИ:



Очков перемещения необходимо 3, покажем это:



Если на пути более 1 диагонали – каждая вторая диагональ считается за 2 клетки, то есть:

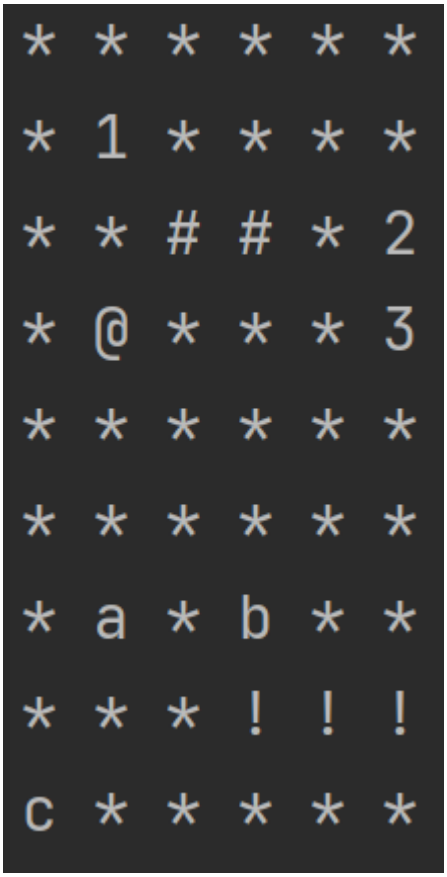


Расчет сражения

Если позволяет Дальность атаки, то в рамках хода игрок может совершить Атаку. Урон Здоровью юнита определяется как Урон атаки – Оставшаяся защита Юнита. Пусть был Мечник и ХП 15, защитой 6. На него нападет Топорщик с атакой 9. В результате у Мечника останется 12 ХП и защита 0. Очки атаки в первую очереди тратятся на пробитие брони. В следующем столкновении с Мечником его броня считается равной 0.

Поле битвы

Так как наша игра исключительно консольная, поле битвы представляет собою квадрат, размером пусть 15 на 15 клеток. Каждая клетка может быть занята только одним Юнитом. Клетка может быть обычной или специальной(это клетки, перемещение по которым ведет к штрафу). Пример:



Легенда:

- * - обычная клетка
- # - Болото
- @ - Холм
- ! – Дерево
- 1, 2, 3 – Юниты игрока
- a, b, c – Юниты противника

Подчеркиваю, это исключительно пример. Вы можете использовать любые другие обозначения, любого вида, цвета и т.д.

Штраф к перемещению

В таблице ниже приведены примерные штрафы на перемещение, накладываемые на тип Юнита в зависимости от типа препятствия.

Тип Юнита	Препятствие	Штраф

Пеший	Болото	1.5
Пеший	Холм	2
Пеший	Дерево	1.2
Лучник	Болото	1.8
Лучник	Холм	2.2
Лучник	Дерево	1
Всадник	Болото	2.2
Всадник	Холм	1.2
Всадник	Дерево	1.5

Примечание: если штраф равен 1 – значит, штрафа нет. В остальных случаях считается коэффициент. Например, если Пеший юнит идет через Болото, то каждая клетка Болота считается за 1.5 клетки при перемещении. Также, препятствия можно только обходить, находится на них нельзя.

Ход

За ход пользователь может сделать каждым своим юнитом перемещение и атаку, или что-то одно, или не делать вообще ничего. После каждого действия пользователя выводится обновленное Поле битвы и статистика (если это была атака). Когда заканчиваются все действия – ход переходит к другому игроку.

Бот

Компьютер в данной ситуации играет очень просто – если есть возможность атаковать – он ТОЛЬКО атакует, не осуществляет перемещения. Если атаковать некого – он просто перемещается вперед на некоторую свою скорость (можно считать на максимальную).

Конец игры

Игра заканчивается тогда, когда заканчиваются все Юниты у одного из игроков. Когда ХП Юнита становится равным 0 он считается убитым и исчезает с карты.

Описание функционирования программы

При старте программы, Игроку сообщается, что на Город готовится нападение. При этом, сообщается, что в городской казне есть некоторая сумма денег, а также выводится список всех юнитов с их характеристиками и стоимостью. Далее, Игрок покупает Юнитов. Необходимо предусмотреть изменение баланса и добавить невозможность покупки Юнита при недостаточном балансе.

Далее, начинается Сражение. Для простоты, на старте все Юниты что Бота, что Игрока находятся на самых дальних линиях поля (Юниты Игрока вверху, Юниты Пользователя – внизу). Их расположение может быть зафиксировано, может быть рандомно в указанных пределах.

Пользователю предлагается выполнить ход за каждого своего Юнита. Пользователь может осуществить Перемещение, осуществить Атаку, не делать ничего или выполнить одно из действий. После каждого действия Поле битвы обновляется. В начале каждого хода Игрока выводится информация о каждом юните Бота – его тип и оставшееся здоровье и защита. Также, может выводится информация о юнитах Игрока. В конце игры выводится сообщение о победе или поражении Игрока.

Требования

Общие требования – стоит избегать публичных полей, все поля должны иметь модификатор доступа private, обращение и доступ к ним должен осуществляться посредством геттеров и сеттеров.

Таблица 4. Технические требования

Суть	Требование
Хранение данных	Использование объектов и классов
Хранение массивов данных	Использование коллекций, обоснованный выбор типа коллекции

Реализация функционала	1 задача = 1 маленький метод, избегать длинных методов, выполняющих большое количество действий
Хранение сущностей(Меню, Поле битвы и тд)	Использование объектов и классов
Дублирование кода	По максимуму избегание дублирования, использование механизмов ООП
Алгоритмы	Использование готовых решений или своих, но с обязательным обоснованием

Под использованием механизмов ООП следует понимать использование основных принципов, ведущих к снижению связности кода и уменьшению его дубляжа. Каждое свое решение студент должен быть в состоянии обосновать.

Глобальная идея таблицы сводится к тому, что каждая сущность программы должна быть представлена в виде отдельного класса. Каждое действие, которое может совершать сущность, представлено в виде отдельного метода.

С точки зрения бизнес требований, функционал должен быть удобным настолько, насколько это возможно в рамках консоли😊. Можно использовать возможности кастомизации консоли, почитайте об этом самостоятельно при желании.

Тестирование работоспособности

Важный момент с точки зрения оптимизации тестирования программы и защиты индивидуального задания лабораторной работы. Необходимо реализовать класс и/или метод, который при старте программы автоматически инициализирует всех юнитов игрока и Бота. Например, при запуске такого метода игрок автоматически получает 2-х Мечников и Рыцаря, а Бот – Арбалетчика и Копейщика.

Также, с целью ускорения тестирования предлагается сократить размер поля, например, до 10x10, а также ограничить количество юнитов, чтобы игра была быстрее.

Важное замечание

Понятия, представленные в разделе Глоссарий, не являются жесткой диаграммой классов. Другими словами, вы не обязаны реализовать все понятия Глоссария в виде классов, вы можете реализовать как меньшее число классов, так и большее. В любом случае, выбор того или иного решения нужно обосновать.

Вы можете оспаривать некоторые Технические требования. В этом случае необходимо быть готовым указать на их недостатки в вашей конкретной реализации этой задачи, предложить свое решение и указать на достоинства.

Приложение 1

В этом приложении приведу наиболее полезные сочетания клавиш для облегчения работы в Intellij IDEA.

Shift + F10	Запуск программы
Alt + Insert	Позволяет быстро создавать конструкторы, геттеры и сеттеры в классе
Ctrl + O	Переопределение метода
Ctrl + /	Однострочное комментирование / раскомментирование
Ctrl + Shift + /	Многострочное комментирование / раскомментирование
Ctrl + C	Копировать
Ctrl + V	Вставить
Ctrl + X	Вырезать
Ctrl + D	Дублирование строки

Полный список, например, тут: https://devcolibri.com/%D0%B3%D0%BE%D1%80%D1%8F%D1%87%D0%B8%D0%B5-%D0%BA%D0%BB%D0%B0%D0%B2%D0%B8%D1%88%D0%B8-intellij-idea/