Модуль 3. Основы программирования Android приложений

Тема 3.1. Объектно-ориентированное проектирование на примерах

4 часа

Оглавление

3.1. Объектно- ориентированное проектирование на примерах

Класс в Java — это абстрактный тип данных и состоит из полей (переменных) и методов (функций).

В процессе выполнения Java-программы система использует определения классов для создания экземпляров классов или объектов.

Для того чтобы создать новую программу необходимо произвести проектирование. Проектирование программ в объектно-ориентированной парадигме отличается от классической.

Рассмотрим примеры проектирования на разных задачах.

Задача 1. Дроби.

Вычисление и вывод дробей

Прежде, чем рассмотреть процесс проектирования, рассмотрим пример. Ввести дроби A и B и вычислить и вывести на экран в виде правильной дроби. Решить без применения ООП и с применением ООП.

$$\left(\frac{An}{Ad}+3\right):\left(\frac{Bn}{Bd}-\frac{1}{3}\right)$$

1a. Не ООП решение (1a/Drob1.java)

При решении задачи выполняем шаг за шагом последовательно, как изложено в задаче. То есть выполняем следующую последовательность действий:

- вводим обе дроби — A_n,A_d,B_n,B_d
- приводим первую скобку к общему знаменателю и складываем числители (A_n+3*A_d)/A_d,
- приводим вторую скобку к общему знаменателю и вычитаем числители (3*B_n-B_d)/3*B_d,
- делим результат первой операции на результат второй,
- выводим результат в виде числовой дроби и в виде десятичной дроби.

Получившийся код:

```
Scanner sc = new Scanner(System.in);
System.out.println("Введите первую дробь");
int An = sc.nextInt();
int Ad = sc.nextInt();
System.out.println("Введите вторую дробь");
int Bn = sc.nextInt();
int Bd = sc.nextInt();
// считаем первую скобку (приводим к
```

```
//общему знаменателю  и складываем числитель)
      An = An + 3 * Ad;
      // считаем вторую скобку (приводим к общему
      // знаменателю и складываем числитель)
      Bn = 3 * Bn - Bd;
      Bd = 3 * Bd;
      // считаем деление скобок
      An = An * Bd;
      Ad = Ad * Bn;
System.out.println("Результат:");
      // печатаем в десятичном виде
      System.out.println(1.0 * An / Ad);
      if (An / Ad == 0) {
      // печатаем в обычном виде
      System.out.println(An);
      System.out.println("---");
      System.out.println(Ad);
      } else {
      // печатаем правильную дробь
      System.out.println("   " + An % Ad);
      System.out.println(An / Ad + "-----");
      System.out.println("   " + Ad);
      }
```

Результат работы программы:

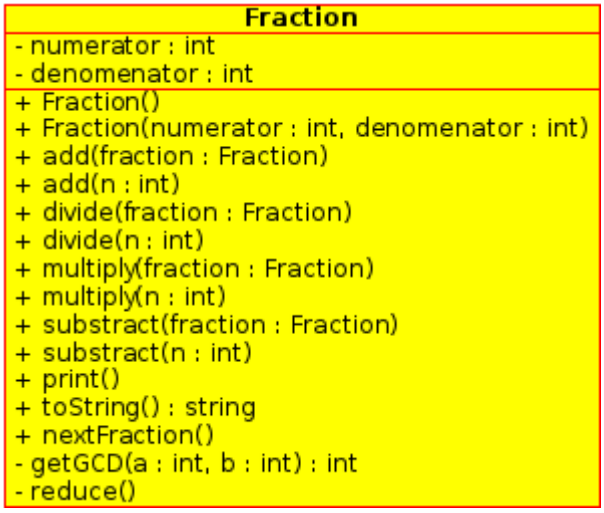
```
Введите первую дробь
1
4
Введите вторую дробь
3
5
Результат:
12.1875
      3
12-----
      16
```

Как видим решение этой задачи было довольно тривиальным, а полученный код простым и понятным.

1b. ООП решение (1b/Test.java)

Теперь решим эту же задачу с применением ООП. В отличие от предыдущего примера, при разработке в ООП парадигме мы сначала спроектируем класс так, чтобы в полях хранился числитель и знаменатель, чтобы были методы для работы с дробью – сложить, вычесть, умножить, делить, вывести на экран, чтобы был конструктор для удобного создания дроби. Кроме того, перегрузим эти методы, чтобы мы могли вызывать их для аргументов разных типов. И только после того как мы получим этот класс, мы займёмся решением непосредственно самой задачи.

Выше мы словесно описали нужный класс, однако чаще используют более наглядный и компактный способ — UML диаграммы.



UML (аббр. Universal Modeling Language — универсальный язык моделирования), это способ графического описания применяемый для разработки программного обеспечения.
Существует несколько разновидностей UML диаграмм, которые используются в зависимости от стадии разработки и целей описания.
Здесь мы будем говорить о диаграммах классов, которые позволяют дать визуальное графическое описание классов и их связей между собой. Обычно создание диаграммы классов знаменует собой окончание процесса анализа и начало процесса проектирования.
Для построения диаграмм UML существует множество инструментов, например, ArgoUML для всех платформ, NClass для Windows; Umbrello для Linux.

Каждый класс в диаграмме классов UML описывается как состоящий из трех выделенных блоков (см. рисунок): наименование класса, поля класса и методы класса. При этом перед именем поля или метода указываются модификаторы видимости:

Обозначение	Модификатор Java
+	public - открытый доступ
-	private - только из методов того же класса
#	protected - только из методов этого же класса и классов, создаваемых на его основе

Создадим класс Fraction для данной диаграммы:

```
import java.util.Scanner;

public class Fraction {
        private int numerator;
        private int denominator=1;

        public void add(Fraction fraction) {
                numerator = numerator * fraction.denominator + fraction.numerator
*
                                                denominator;
                denominator = denominator * fraction.denominator;
                reduce();
```

```
        }

        public void add(int n) {
                add(new Fraction(n, 1));
        }

        public void substract(Fraction fraction) {
                numerator = numerator * fraction.denominator - fraction.numerator
*
                                                        denominator;
                denominator = denominator * fraction.denominator;
                reduce();
        }

        public void substract(int n) {
                substract(new Fraction(n, 1));
        }

        public void multiply(Fraction fraction) {
                numerator = numerator * fraction.numerator;
                denominator = denominator * fraction.denominator;
                reduce();
        }

        public void multiply(int n) {
                multiply(new Fraction(n, 1));
        }

        public void divide(Fraction fraction) {
                if (fraction.numerator == 0) {
                        System.out.println("На эту дробь делить нельзя!");
                        return;
                }
                multiply(new Fraction(fraction.denominator, fraction.numerator));
        }

        public void divide(int n) {
                divide(new Fraction(n, 1));
        }

        public void nextFraction() {
                Scanner sc = new Scanner(System.in);
                int n = sc.nextInt();
                int d = sc.nextInt();
                if (d == 0) {
                        System.out.println("Знаменатель не может быть нулевым!");
                        return;
                }
                numerator=n;
                denominator=d;
                reduce();
        }

        Fraction(){
        }

        Fraction(int numerator, int denominator) {
                if (denominator == 0) {
```

```
                        System.out.println("Знаменатель не может быть нулевым!");
                        return;
                }
                this.numerator = numerator;
                this.denominator = denominator;
                reduce();
        }

        public String toString(){
                return (numerator*denominator<0?"-":" ")+ Math.abs(numerator)+
                                                "/"+Math.abs(denominator);
        }

        public void print() {
                if(numerator % denominator == 0){
                        System.out.println(numerator/denominator);
                        return;
                }
                if (numerator / denominator == 0) {
                        System.out.println(" " + Math.abs(numerator));
                        System.out.println((numerator*denominator<0?"-":" ")+" ---
- или "+
                                                                        1.0 *
numerator / denominator);
                        System.out.println(" " + Math.abs(denominator));
                } else {
                        System.out.println(" " + Math.abs(numerator %
denominator));
                        System.out.println((numerator*denominator<0?"-":"
")+numerator /
                                                                denominator + "---- или
"+1.0 * numerator / denominator);
                        System.out.println(" " + Math.abs(denominator));
                }
        }

        private int getGCD(int a, int b) { return b==0 ? a : getGCD(b, a%b); }

        private void reduce(){
                int t=getGCD(numerator,denominator);
                numerator/=t;
                denominator/=t;
        }

}
```

Мы создали по описанию класс, который содержит нужные поля и методы. По сути мы создали новый тип данных, который мы в дальнейшем можем использовать в своих программах. Поскольку класс готов, то теперь мы можем, наконец, приступить к вычислению задачи:

```
public class Test {

        public static void main(String[] args) {
         Fraction A = new Fraction();
         Fraction B = new Fraction();
         A.nextFraction();
```

```
         B.nextFraction();
         A.add(3);
         B.substract(new Fraction(1,3));
         A.divide(B);
         A.print();
        }
}
```

Запустив на выполнение test.java и введя те же значения, получаем такой же результат как и в прошлый раз:

```
1 4
3 5
    3
12---- или 12.1875
    16
```

Выводы

Мы решили задачу, используя два разных подхода. Сравним (при одинаковом результате):

1. В первой программе порядка 30 строк. Процесс разработки прост и понятен
2. Во второй программе порядка 100 строк. Процесс разработки более громоздкий. Результат стал доступен только в конце.

А теперь вопрос, если результат одинаков, то зачем “платить больше”? Казалось бы первый вариант лучше, первый? Давайте рассмотрим другой пример и уже тогда сделаем вывод.

Более сложные операции с дробями

Ввести 5 правильных дробей и вычислить сумму и произведение дробей. Решить без применения ООП и с применением ООП.

$$\sum_{i=0}^{5} \frac{An_i}{Ad_i} \quad \text{и} \quad \prod_{i=0}^{5} \frac{An_i}{Ad_i}$$

1a.Решение без ООП (1a/Drob2.java)

Решение без ООП – аналогично решению выше. То есть мы удаляем все, что мы делали в первой задаче и решаем заново, последовательно шаг за шагом, четко следуя заданию. Первоначально мы вводим дроби в два массива — числители в массив n, а знаменатели в массив d. Для нахождения суммы мы в программе определяем дробь rez1n/rez1d как 0/1 и в цикле к ней прибавляем все введенные дроби. Для нахождения произведения дробей мы в программе определяем дробь rez2n/rez2d как 1/1 и в цикле умножаем на неё все введенные дроби. В конце мы выводим дробь rez1n/rez1d как результат суммы и rez2n/rez2d как результат произведения.

```
        Scanner sc = new Scanner(System.in);
        System.out.println("Введите дроби:");
        int n[] = new int[5];
        int d[] = new int[5];
        for (int i = 0; i < n.length; i++) {
                System.out.println("==============");
                n[i] = sc.nextInt();
                d[i] = sc.nextInt();
        }
        int rez1n = 0;
        int rez1d = 1;
        int rez2n = 1;
        int rez2d = 1;
        for (int i = 0; i < n.length; i++) {
        rez1n = rez1n * d[i] + rez1d * n[i];
        rez1d = rez1d * d[i];
        rez2n = rez2n * n[i];
        rez2d = rez2d * d[i];
        }
        System.out.println("Результат 1:");
        // печатаем в десятичном виде
        System.out.println(1.0 * rez1n / rez1d);
        if (rez1n / rez1d == 0) {
        // печатаем в обычном виде
        System.out.println(rez1n);
        System.out.println("---");
        System.out.println(rez1d);
        } else {
        // печатаем правильную дробь
        System.out.println("   "+rez1n%rez1d);
        System.out.println(rez1n/rez1d+"-----");
        System.out.println("   "+rez1d);
        }
        System.out.println("Результат 2:");
                // печатаем в десятичном виде
                System.out.println(1.0*rez2n/rez2d);
        if (rez2n / rez2d == 0) {
                // печатаем в обычном виде
                System.out.println(rez2n);
                System.out.println("---");
                System.out.println(rez2d);
                } else {
                // печатаем правильную дробь
                System.out.println("   "+rez2n%rez2d);
                System.out.println(rez2n/rez2d+"-----");
                System.out.println("   " + rez2d);
        }
```

Результат работы программы будет следующий:

```
Введите дроби:
==============
1
2
==============
1
2
```

```
==============
1
2
==============
1
2
==============
1
2
Результат 1:
2.5
   16
2-----
   32
Результат 2:
0.03125
1
---
32
```

1b.ООП решение(1b/Test1.java)

В отличие от решения этой задачи в варианте без ООП, у нас уже имеется класс дроби, и мы его просто используем, как есть, без каких бы то ни было переделок. Переписать же придется только лишь ту часть, где мы использовали дроби, то есть метод main. Поэтому решение выглядит так:

```
public class Test1 {
        public static void main(String[] args) {
         Fraction A[] = new Fraction[5];
         for (int i = 0; i < A.length; i++) {
                 A[i] = new Fraction();
                 A[i].nextFraction();
         }
         Fraction rez1 = new Fraction(0, 1);
         Fraction rez2 = new Fraction(1, 1);
         for (int i = 0; i < A.length; i++) {
                 rez1.add(A[i]);
                 rez2.multiply(A[i]);
         }
         System.out.println("Сумма");
         rez1.print();
         System.out.println("Произведение");
         rez2.print();
        }
}
```

Результат работы программы будет следующий:

```
1 2
1 2
1 2
1 2
1 2
Сумма
```

```
   1
2---- или 2.5
   2
Произведение
   1
 ---- или 0.03125
   32
```

Выводы

Давайте, снова сравним решение в двух парадигмах:

1. В варианте без ООП мы написали ~45 строк кода, и кстати опять думали над реализацией операций (вычислениями).
2. В варианте ООП размер программы ~20 строк кода. Причем мы теперь не задумывались над тем, как именно делаются каждые операции.

Вывод из примеров: По мере усложнения задач, ООП парадигма дает гораздо лучшие результаты в проектировании!

При разработке класса мы описывали все поля и методы, которые нам нужны для работы с нашим объектом. Принцип инкапсуляции дает нам возможность скрывать часть данных и методов, то есть часть реализации класса от потребителя. Публичные же методы по сути представляют собой интрефейс взаимодействия с этим объектом.

Сетепень сокрытия и тип воздействия на поля класса определяются в момент проектирования класса. Например, если вы не создаете методы для доступа к полям класса (сеттеров) и прочие методы не меняют содержимое полей, то полученный класс называется immutable. Ярким примером immutable класса является стандартный класс String. Если в таком классе вы хотите выдать потребителю измененный объект (напимер увлеиченную дробь) - вы просто создаете новый объект с измененным содержимым.

Чаще, все же используются классы, которые позволяют менять свои данные. Для этого вы можете использовать методы сеттеры (setters) либо методы которые меняют поля по какому то закону. В нашем случае - это методы, реализующие математические операции с дробью. Наличие специальных методов для доступа (геттеров и/или сеттеров) необязательно и определяется ТЗ, однако их наличие (если это не противоречит ТЗ) скорее приветствуется, так как есть большое количество полезных фреймворков, которые используют акссессоры объектов. Например спецификация Java Beans требует наличие аксессоров.

Задача 2 (минипроект). Текстовый квест

Задание

Написать игру - текстовый квест «Корпорация». Цель игрока — поднять социальный статус персонажа.

Описание

В процессе игры игроку рассказывают интерактивную историю. История начинается с первой сцены. Игрок выбирает вариант развития событий из ограниченного набора 2-3шт. Выбранная сцена становиться текущей и у нее также есть варианты развития событий. История заканчивается, когда у текущей ситуации нет вариантов развития событий. При выборе варианта у персонажа меняются характеристики: карьерное положение(К), активы(А), репутация(Р).

Пример сюжета

К=1,А=100тр,Р=50%

1я Сцена - «Первый клиент». Вы устроились в корпорацию менеджером по продажам программного обеспечения. Вы нашли клиента и продаете ему партию МС Windows. Ему достаточно было взять версию “HOME” 100 коробок. Вам предлагается на выбор 3и варианта ваших действий:

- вы выпишете ему счет на 120 коробок версии “ULTIMATE” по 50тр(К+0,А-10,Р-10)
- вы выпишете ему счет на 100 коробок версии “PRO” по 10тр (К+1,А+100,Р+0)
- вы выпишете ему счет на 100 коробок версии “HOME” по 5тр (К +0, А +50, Р +1)

2а.Не ООП решение (2а/Quest.java)

Давайте решим эту задачу простым последовательным выполнением поставленной задачи. Для этого будем от пользователя получать номер выбранного действия и в соответствии с ним выбирать ветку дальнейшего сюжета с изменением параметров К,А,Р.

```
Scanner in = new Scanner(System.in);
int K=1,A=100,P=50;
System.out.println("Вы прошли собеседование и вот-вот станете сотрудниуом
Корпорации. \n Осталось уладить формальности - подпись под договором (Введите ваше
имя):");
String name;
name=in.next();
        System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+
"%\n=====");
System.out.println("Только вы начали работать и тут-же удача! Вы нашли клиента и
продаете ему партию \n ПО МС Виндовс. Ему в принципе достаточно взять 100 коробок
версии HOME.");
System.out.println("- (1)у клиента денег много, а у меня нет - вы выпишете ему счет
на 120 коробок \n ULTIMATE по 50тр");
System.out.println("- (2)чуть дороже сделаем, кто там заметит - вы выпишете ему
счет на 100 коробок \n PRO по 10тр");
System.out.println("- (3)как надо так и сделаем - вы выпишете ему счет на 100
коробок HOME по 5тр");
int a1=in.nextInt();
if(a1==1){
        K+=0;A+=-10;P+=-10;
System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====
");
                // Следующие ситуации для этой ветки сюжета
        } else if(a1==2) {
                K+=1;A+=100;P+=0;
System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====
```

```
");
                    // Следующие ситуации для этой ветки сюжета
} else {
                    K+=0; A+=50; P+=1;
System.out.println("=====\nКарьера:"+K+"\tАктивы:"+A+"т.р.\tРепутация:"+P+"%\n=====
");
                    // Следующие ситуации для этой ветки сюжета
}
System.out.println("Конец");
```

Такое решение, даже в этом небольшом примере, выглядит громоздким. А представьте, что будет, если у вас будет по три варианта в каждом под-варианте, а в нем свою очередь еще три и так далее. Эта программа станет совершенно нечитабельной даже для создателя. Еще хуже если вы решите впоследствии изменить сюжет, удалив или добавив несколько сцен. Тут одно неосторожное изменение может вызвать такой дисбаланс скобок, что найти потерянную скобку будет очень сложно.

2b. ООП решение (2b/Game.java)

Прежде чем решать эту задачу в ООП парадигме, давайте разберемся, как это делать правильно.

Объектно-ориентированное проектирование, как и обычное проектирование, проходит стадию анализа и синтеза. Анализ ориентирован на поиск отдельных сущностей (или классов, объектов). Синтез же воссоздаст полную модель предметной области задачи.

Анализ – это разбиение предметной области на минимальные неделимые сущности. Выделенные в процессе анализа сущности формализуются как математические модели объектов со своим внутренним состоянием и поведением.

Синтез – это соединение полученных в результате анализа моделей минимальных сущностей в единую математическую модель предметной области, с учетом взаимосвязей объектов в ней.

Если же сказать простыми словами, то процесс проектирования можно описать так:

- из текста подробного описания предметной области выбираем все подлежащие - это сущности (классы),
- выбираем все глаголы (отглагольные формы, например деепричастия) – это поведение сущностей (методы классов),
- выбираем дополнения, определения, обстоятельства – они, скорее всего, будут определять состояние сущностей (поля классов).

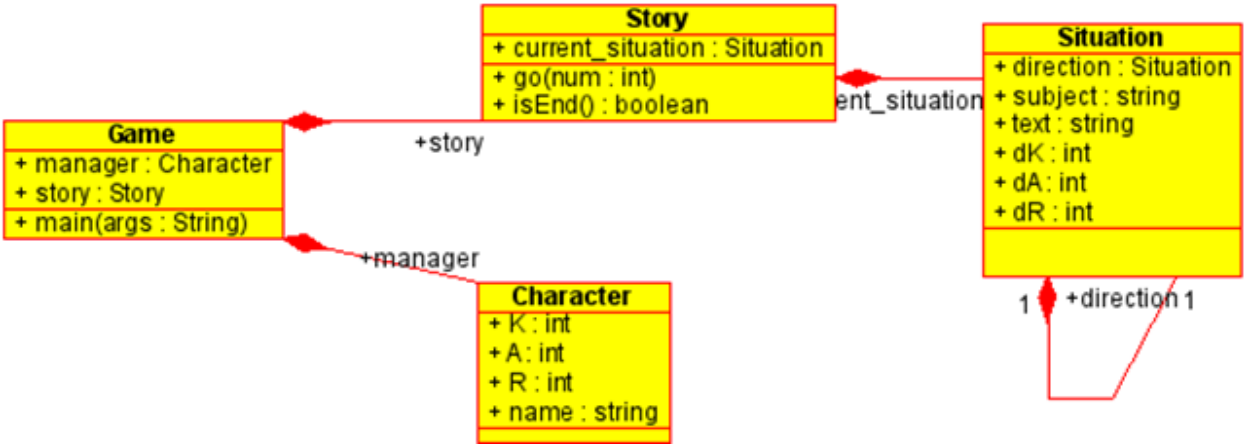
Проведя синтаксический разбор и последующий анализ описания нашей задачи получим следующую таблицу:

Сущности (подлежащие)	Свойства (дополнения)	Действия (глаголы)
Игра	История, персонаж	Начать (основная программа)
Персонаж	Имя, Параметры К, А, Р	
История	Ряд сцен, текущая сцена, начальная сцена	Выбрать следующую сцену, проверить на окончание истории

Сцена	Название, Описание, Возможные варианты развития событий, модификаторы К, А, Р	

После того как мы вычленили из задания сущности, их свойства и действия — построим модели отдельных сущностей (это классы) и объединим их в общую модель, с учетом их взаимосвязей. То есть мы при описании каждой сущности должны учесть, являются ли ее поля простыми типами, или в свою очередь другими сущностями (отношение агрегирования или композиции).

Это и есть разработка приложения. Сделаем это на UML и получим следующую диаграмму классов.



После построения модели вы можете прямо из UML-редактора сгенерировать исходный код (только нужно выбрать язык Java). Однако редактор, зачастую дает правильный, но избыточный для нас код. Мы можем также по диаграмме написать классы сами, тем более что они не большие.

Отдельно отмечу, что классы ***Character*** и ***Situation*** почти пусты; класс ***Story*** в конструкторе создает всю историю как набор ситуаций; класс Game имеет точку входа для консольного выражения он же наш главный метод main, где создается персонаж и история и происходит взаимодействие с пользователем и переключение сцен.

В случае же если приложение разрабатывалось для Android, класс ***Game*** будет расширять класс ***Activity,*** и в методе ***onCreate*** (вместо ***main)*** будет создан объект персонажа и игры, а переключение сцен (метод ***go)*** должен вызываться в методах типа ***onClick*** этой активности.

Кстати если поле ***Direction*** сделать не обычным массивом, а ассоциативным, то написание самой истории (последовательности сцен) сильно упростится, так как вы будете видеть шаги истории не по номерам индексов в массивах вариантов, а по значащим именам сцен.

```
//=======персонаж=======
public class Character {
        public int K;
        public int A;
        public int R;
```

```
        public String name;

        public Character(String name) {
                K = 1;
                A = 100;
                R = 50;
                this.name = name;
        }
}

//=======ситуация=======
public class Situation {

  public Situation[] direction;
  public String subject;
  public String text;
  public int dK;
  public int dA;
  public int dR;

  public Situation (String subject, String text, int variants, int dk,int da,int
dr) {
           this.subject=subject;
           this.text=text;
           dK=dk;
           dA=da;
           dR=dr;
           direction=new Situation[variants];
  }
}

//=======история=======
public class Story {
        private Situation start_story;
        public Situation current_situation;
        Story() {
                start_story = new Situation("первая сделка (Windows)","Только вы
начали работать и тут-же удача! Вы нашли клиента и продаете ему " + "партию ПО MC
Виндовс. Ему в принципе достаточно взять 100 коробок версии HOME.\n"
+ "(1)у клиента денег много, а у меня нет - вы выпишете ему счет на 120 коробок
ULTIMATE по 50тр\n"
+ "(2)чуть дороже сделаем, кто там заметит - вы выпишете ему счет на 100 коробок
PRO по 10тр\n"
+ "(3)как надо так и сделаем - вы выпишете ему счет на 100 коробок HOME по 5тр ",3,
0, 0, 0);
                start_story.direction[0]=new Situation("корпоратив", "Неудачное
начало, ну чтож, сегодня в конторе копроратив! "
+ "Познакомлюсь с коллегами, людей так сказать посмотрю, себя покажу", 0, 0, -10, -
10);
                start_story.direction[1]=new Situation("совещание, босс доволен",
"Сегодня будет совещание, меня неожиданно вызвали,"
+ "есть сведения что \n босс доволен сегодняшней сделкой.", 0, 1, 100, 0);
                start_story.direction[2]=new Situation("мой первый лояльный
клиент", "Мой первый клиент доволен скоростью и качеством "
+ "моей работы. Сейчас мне звонил лично \nдиреткор компании и сообщил что скоро
состоится еще более крупная сделка"
+ " и он хотел чтобы по ней работал именно я!", 0, 0, 50, 1);
                current_situation=start_story;
```

```
        }
        public void go(int num) {
                if(num<=current_situation.direction.length)
                current_situation=current_situation.direction[num-1];
                else System.out.println("Вы можете выбирать из
"+current_situation.direction.length+" вариантов");
        }
        public boolean isEnd(){
                return current_situation.direction.length==0;
        }
}
//=======игра=======
public class Game {
  public static Character manager;
  public static Story story;
  public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        System.out.println("... подпись под договором (ваше имя):");
        manager =new Character(in.next());
        story = new Story();
        while (true){
                        manager.A+=story.current_situation.dA;
                        manager.K+=story.current_situation.dK;
                        manager.R+=story.current_situation.dR;
System.out.println("=====\nКарьера:"+manager.K+"\tАктивы:"+manager.A+"\tРепутация:"
+manager.R+"\n=====");
System.out.println("=============="+story.current_situation.subject+"==============="
);
                        System.out.println(story.current_situation.text);
                        if(story.isEnd())
{System.out.println("====================the end!====================");return;}
                        story.go(in.nextInt());
        }
}
}
```

Выводы

В предыдущем примере мы произвели проектирование игры-квеста. Обратите внимание, что при росте количества сцен — сложность восприятия кода сильно возрастает и программа быстро становится нечитабельной. В то же время при использовании ООП, описание сцен довольно компактное и код получается простой и прозрачный.

Это даёт на сделать вывод об очевидном преимуществе ОО подхода при разработки ПО. Однако мы не задействовали всего арсенала ООП. По сути, мы использовали лишь инкапсуляцию.

Давайте теперь рассмотрим пример, в котором мы сможем задействовать все преимущества ОО.

Задача 3. Электронный журнал

Необходимо разработать электронный журнал для школы. Разработку будем проводить только в ОО парадигме.

Детализируем постановку задачи. Итак есть школа как учебное заведение, находящаяся по адресу 344000, г. Ростов-на-Дону, …., в которой происходит обучение детей по стандартной программе среднего образования (11 классов). В школе работают учителя, которые преподают по несколько дисциплин, причем некоторые имеют дополнительную нагрузку в виде классного руководства либо факультативных предметов, кружков. Помимо преподавателей, в школе есть прочий персонал, директор, завуч, охранники, повара, уборщики. Вход в школу осуществляется при предъявлении магнитной карты с уникальным ID. Есть множество документов, регламентирующих процесс образования.

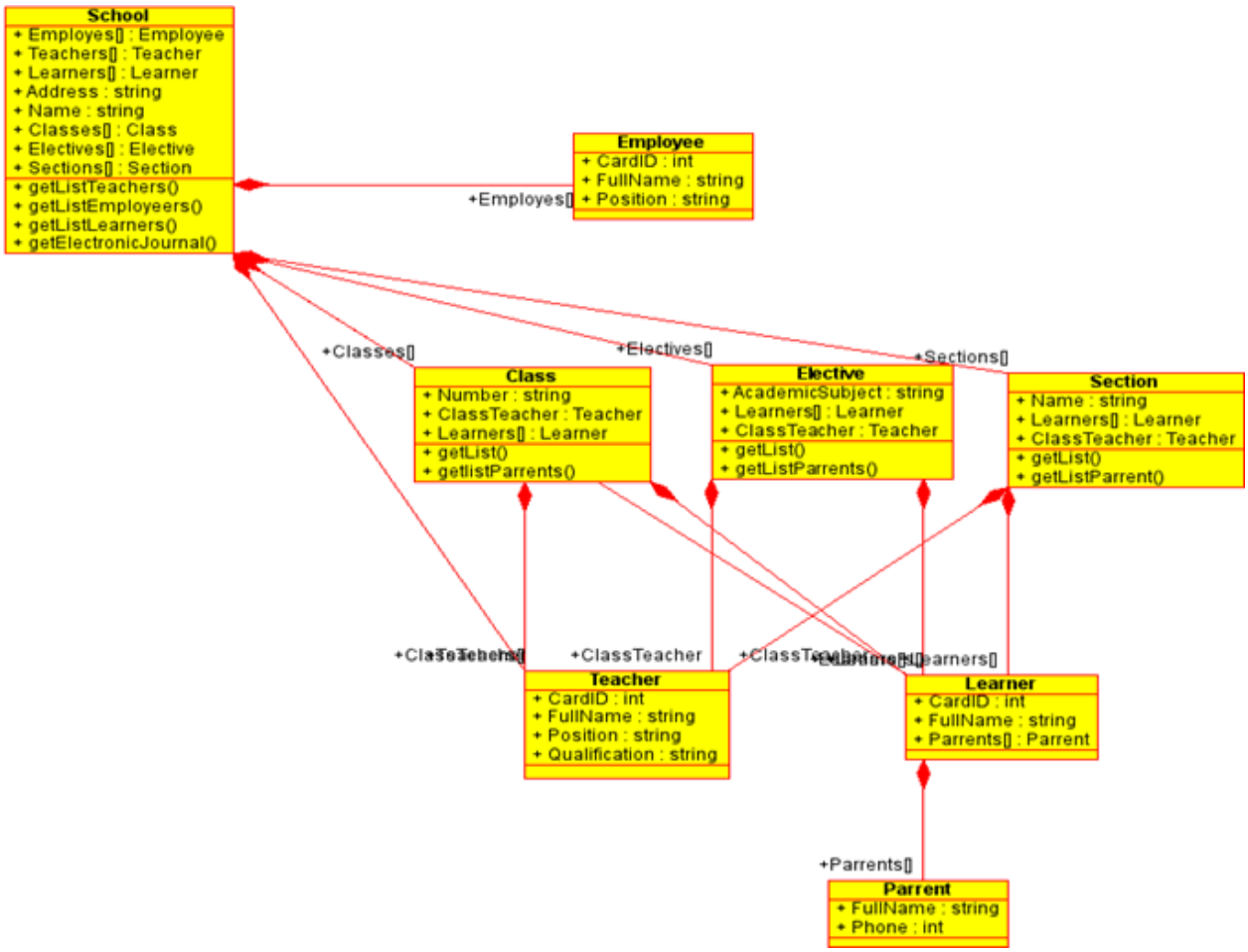
Для реализации в этом задании мы выберем следующие документы:

- общий список преподавательского состава с указанием квалификации для ведения отчётности,
- общий список школьников с указанием возраста для ведения отчётности,
- общий список всех людей имеющих доступ в школу, для выдачи магнитных карт,
- список учеников класса вместе с родителями для организации собраний,
- Электронный журнал, каждая страница которого связывает отчетность о посещении/оценках школьников определённого класса по датам, с учебным предметом и преподавателем.

Описывать процесс формирования журнала не будем – вы и так знаете. Давайте проведем анализ технического задания (сущность, свойства, действия):

Сущности (подлежащие)	**Свойства (дополнения)**	**Действия (глаголы)**
школа	сотрудники, список классов, список кружков, список факультативов	принять на работу сотрудника, принять ученика, выдать общий список имеющих доступ в школу (CardID, ФИО), общий список учителей (ФИО, квалификация), общий список школьников (ФИО, класс, возраст), ЭлЖур
сотрудник	CardID, ФИО, должность	
школьник	CardID, ФИО, родители	
учитель	CardID, ФИО, должность, квалификация	
родитель	ФИО, номер телефона	
класс	номер, список учеников, классный руководитель	выдать список учеников, список учеников с родителями
кружок	название, список учеников, ведущий учитель	выдать список учеников, список учеников с родителями
факультатив	предмет, список учеников, ведущий учитель	выдать список учеников, список учеников с родителями

В соответствии с анализом построим UML диаграмму.



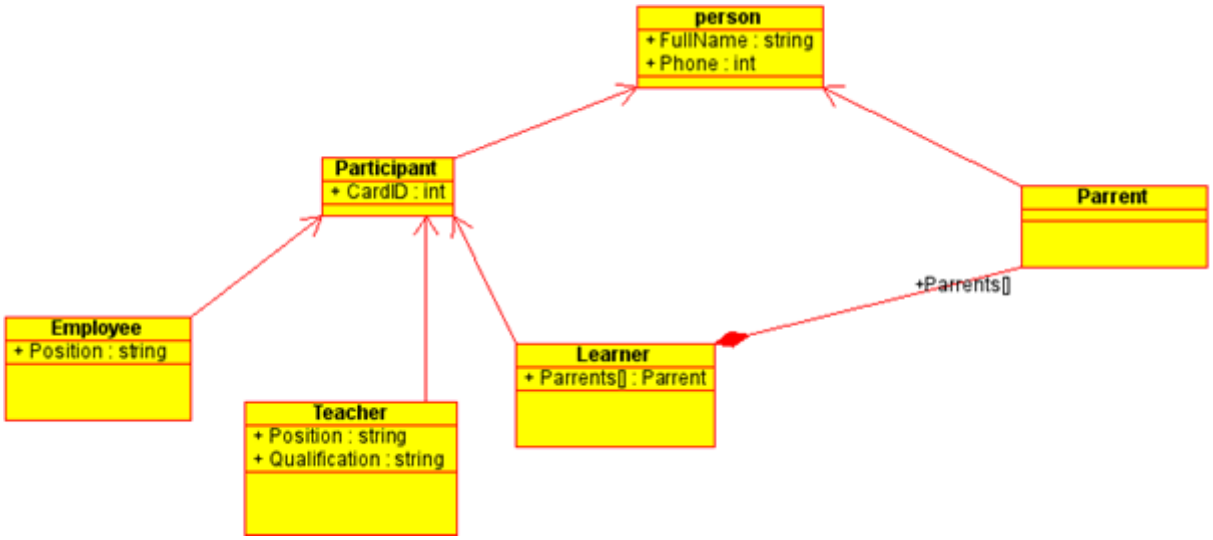
Итак, мы получили диаграмму классов нашей будущей программы. Однако в ней имеется ряд вопросов.

Во-первых, в некоторых классах есть повторяющиеся свойства (например ФИО,CardID). Это наводит на мысль об общности этих классов.

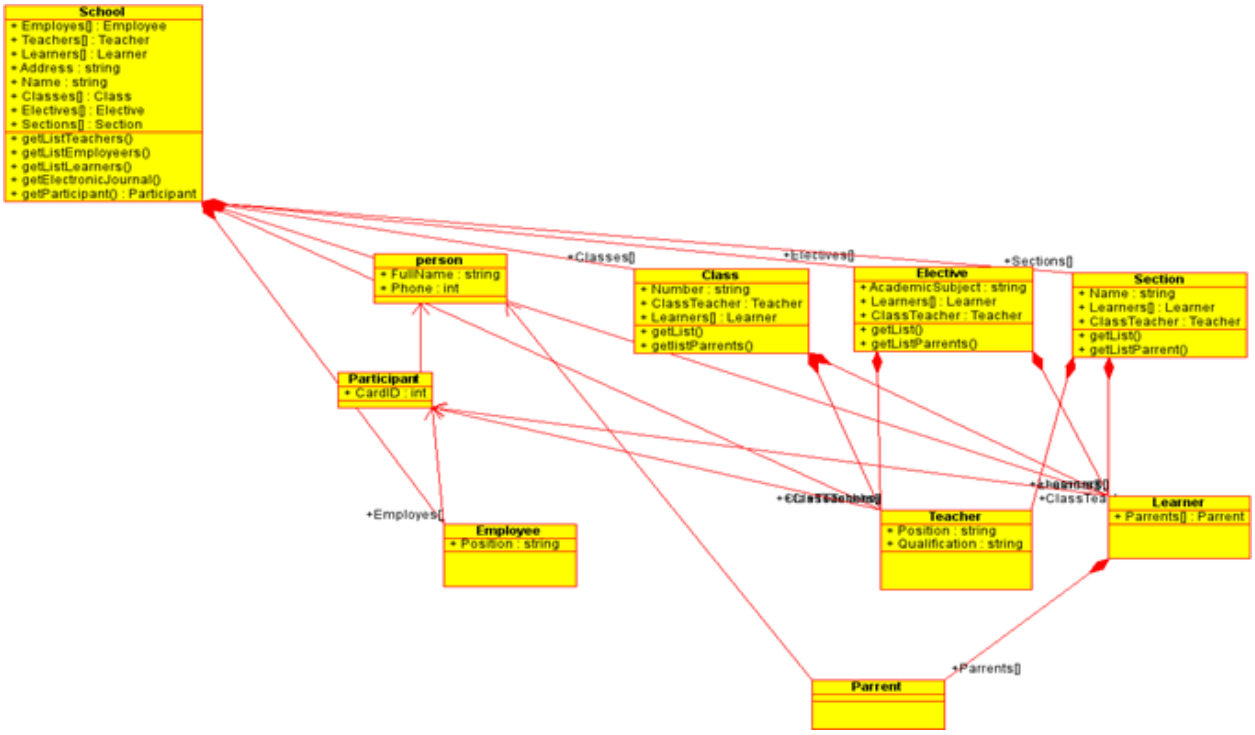
Во-вторых, в постановке задачи был отчёт – общий список всех людей имеющих доступ школу. Однако добавить такой метод в класс ***School*** не получается, так как в нашей программе нет обобщённого типа для всех участников учебного процесса, а следовательно не возможно вернуть массив объектов неодинаковых типов.

Для решения этих вопросов выделим суперклассы для всех участников, описанных в анализе ТЗ. Небольшой анализ показывает, что самый общий класс для всех – ***Person*** (персона, человек) с полями ФИО и тел.

Также выделим две разновидности персон – имеющих доступ в школу (с карточкой – персонал, учителя, ученики) и не имеющих доступа (родители)



Изменяем схему с учётом наследования классов



В приведённой диаграмме поднятые вопросы разрешены. Теперь в классе School есть метод, который нам вернёт массив участников учебного процесса. Обратите внимание, что вследствие полиморфизма вы получите список именно участников (поля ФИО, phone, CardID) независимо от того, какие роли будут у каждого конкретного участника – ученик, учитель или сотрудник. Однако общие методы этих объектов при вызове буду вызваны именно в соответствии фактическому классу.

Конечно, мы только начали проектировать систему в вышеприведённом UML, но уже видим, что использование полиморфизма и наследования даёт дополнительную гибкость в проектировании.

*Шаблоны и принципы проектирования

Парадигма ООП давно и устойчиво утвердилась как основная при разработке ПО. В представленном материале автор старался рассказать просто о сложном, так как проектирование ПО - это очень сложный процесс, которым обычно занимаются самые высококвалифицированные специалисты – системные архитекторы. При проектировании профессионалы помимо известных нам основных принципов ООП (инкапсуляция, наследование, полиморфизм) опираются на:

- глубокое знание computer science, опыт программирования, построения и управления структурами данных,
- на знание всевозможных фреймворков, программных платформ, сред разработки, серверов баз данных и серверов приложений,
- опыт внедрения и сопровождения ПО.

для того, чтобы созданный ими проект мог быть не только реализован в кратчайшие сроки наиболее эффективным способом, но и чтобы у заказчика он был наиболее эффективен, то есть выдавал заказанную функциональность с наименьшими затратами.

Конечно лучшие практики были систематизиравны и выработаны шаблоны и принципы проектирования.

Первоначально были выработаны так называемые шаблоны проектирования. Впервые шаблоны были описаны в книге “Шаблоны проектирования: Элементы повторно используемого объектно-ориентированного программного обеспечения”, написанной Эриком Гамма, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом, которых называли бандой четырех (Gang of Four). В книге GOF были представленные готовые “рецепты”, для типовых задач. Если в процессе проектирования у вас встречается ситуация описанная в шаблонах - вы можете не “изобретать велосипед”, а взять готовую структуру классов и алгоритмы работы из шаблонов. В свое время очень большое внимание развитию шаблонов проеткирования уделяла компания SUN Microsystem. Они разработали наборы шаблонов Java Blueprint. Сегодня известны десятки широкоупотребительных шаблонов. Классическая литература о шаблонах:

- Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995), .
- Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML by Mark Grand (Wiley, 1998)
- Core J2EE Patterns: Best Practices and Design Strategies by Deepak Alur, John Crupi, and Dan Malks (Prentice Hall, 2001)
- UML Distilled: Applying the Standard Object Modeling Language by Martin Fowler with Kendall Scott (Addison-Wesley, 2000)
- The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, and James Rumbaugh (Addison-Wesley, 1998) .

Однако, в процессе проектирования классов важно не только создать правильную архитектуру приложения, но крайне важно чтобы сами классы были разработаны правильно. Свод принципов проектирования классов появился совсем недавно, и на сегодня оформлен в в виде пяти принципов, которые сокращенно называют SOLID.

	Название сокр.	Описание принципа
S	SRP	Принцип единственной обязанности (Single responsibility principle)
O	OCP	Принцип открытости/закртости (Open/closed principle)
L	LSP	Принцип подстановки Барбары Лисков (Liskov substitution principle)
I	ISP	Принцип разделения интерфейса (Interface segregation principle)
D	DIP	Принцип инверсии зависимостей (Dependency inversion principle)

Принцип единственной обязанности SRP

На каждый объект должна быть возложена одна единственная обязанность. Обязанность - это набор методов, служащих одному действующему лицу. Для обязанности действующее лицо - единственный источник изменений (Роберт Мартин). Например неправильный код:

```
public class MainActivity extends Activity implements View.OnClickListener {
	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_main);
		Button btnSend = (Button) findViewById(R.id.btnSend);
		btnSend.setOnClickListener((View.OnClickListener) this);
	}

	@Override
	public void onClick(View v) {
		// Разослать рассылку
	}
}
```

Здесь MainActivity помимо своей основной работы создания интрефейса занимается рассылкой. Исправим разделив на два класса:

```
public class MainActivity extends Activity {
	private TestButtonHandler testButtonHandler;

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_main);
		testButtonHandler = new TestButtonHandler(this, R.id.btnTest);
	}
}
```

```
public class TestButtonHandler implements View.OnClickListener {
	private Button btnSend;
```

```
	public TestButtonHandler(Activity activity, int btnID) {
		btnSend = (Button) activity.findViewById(R.id.btnSend);
		btnSend.setOnClickListener((View.OnClickListener) this);
	}

	@Override
	public void onClick(View v) {
		// Сделать рассылку
	}
}
```

Принцип открытости/закртости OCP

Программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменения. Достоинством применения такого подхода следующие:

- Не нужно пересматривать уже существующий код, не нужно менять уже готовые для него тесты при доработке проекта.
- Если нужно ввести какую-то дополнительную функциональность, то это не должно коснуться уже существующих классов или как-либо иначе повредить уже существующую функциональность.

Например мы имеем класс, который выполняет необходимые операции, и класс логера который записывает в файл журнала отметки о выполнении операций:

```
public class Logger {
	void Log(String logText)
	{
		// сохранить лог в файле
	}
}
```

```
public class AnyStuff {
	 private Logger logger;
	 public AnyStuff() {
		logger = new Logger();
	 }
	 public void startAnyOperation()
	 {
		logger.Log("Operation started");
	 }
}
```

Теперь изменим наш проект - будем сохранять записи журнала в базу данных:

```
public class DatabaseLogger {
	void Log(String logText)
	{
		// сохранить лог в БД
	}
}
```

Тогда нам прийдется заменить класс с основной логикой программы!

```
public class AnyStuff {
        private DatabaseLogger logger;
        public AnyStuff() {
               logger = new DatabaseLogger();
        }
        public void startAnyOperation()
        {
               logger.Log("Operation started");
        }
}
```

Но ведь по принципу единственности ответственности не AnyStuff отвечает за логирование, почему изменения дошли и до него? Потому что нарушен наш принцип открытости/закрытости. AnyStuff не закрыт для модификации. Нам пришлось его изменить, чтобы поменять способ хранения его логов.

Защитить от изменений класс AnyStuff поможет выделение абстракции.

```
public interface ILogger
{
   void Log(String logText);
}

public class Logger implements ILogger
{
   public void Log(string logText)
  {
     // сохранить лог в файле
  }
}

public class DatabaseLogger implements ILogger
{
   public void Log(string logText)
   {
     // сохранить лог в базе данных
   }
}
```

```
public class AnyStuff {
       private ILogger logger;

       public AnyStuff(ILogger logger) {
              this.logger = logger;
       }

       public void startOperation() {
              logger.Log("Operation started");
       }
}
```

Теперь смена логики логирования уже не будет вести к модификации нашего класса AnyStuff.

Принцип подстановки Барбары Лисков LSP

Объекты в программе могут быть заменены их наследниками без изменения свойств программы. Иными словами поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа. Классический пример класс прямоугольник и его наследник квадрат:

```
public class Rectangle {
	private double height;
	private double width;

	public double area();

	public void setHeight(double height);

	public void setWidth(double width);
}
```

Класс квадрата просто наследуется от класса прямоугольника с той лишь разницей, что поддерживает ширину и высоту эквивалентными.

```
public class Square extends Rectangle {
	public void setHeight(double height) {
		super.setHeight(height);
		super.setWidth(height);
	}

	public void setWidth(double width) {
		setHeight(width);
	}
}
```

Однако если внимательно посмотреть, то ниже приведенныей пример продемонстрирует некорректность такого подхода:

```
Rectangle rect = new Square();
rect.setWidth(3);
rect.setHeight(10);
System.out.println(rect.area());
```

Полученный результат будет 100 вместо 30. Конечно здесь мы видим, что создан был все таки квадрат, а затем использован в качестве прямоугольника. А если использование объекта класса Rectangle будет совсем не в том месте, где объект был создан? Выход из такой ситуации только один - отказаться от наследования. Варианты исправить ситуацию:

- Rectange и Square будут разными типами
- Square агрегирует объект типа Rectangle.

Принцип разделения интерфейса ISP

Много специализированных интерфейсов лучше, чем один универсальный.

Принцип разделения интерфейсов состоит в том, что слишком универсальные интерфейсы необходимо разделять на более маленькие и специфические, чтобы при использовании маленьких интерфейсов потребителю необходимо было бы реализовать только методы, необходимы им в работе. Например интерфейс созданный с нарушением принципа ISP:

```
public interface Messenger {
	askForCard();
	tellInvalidCard();
	askForPin();
	tellInvalidPin();
	tellNotEnoughMoneyInAccount();
	tellAmountDeposited();
	tellBalance();
}
```

Разделим его на два, которые могут быть использованы независимо:

```
public interface LoginMessenger {
	askForCard();
	tellInvalidCard();
	askForPin();
	tellInvalidPin();
}
```

и

```
public interface WithdrawalMessenger {
	tellNotEnoughMoneyInAccount();
	askForFeeConfirmation();
	tellBalance();
}
```

Принцип инверсии зависимостей DIP

Рависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Иными словами, следует разрабатывать ПО таким образом, чтобы различные модули были автономными, и соединялись друг с другом с помощью абстракции. Пример неправильной реализации:

```
public Keyboard {
	char getchar();
}

public Printer {
	void putchar(char c);
}

class CharCopier {
	void copy(Keyboard reader, Printer writer) {
```

```
            int c;
            while ((c = reader.getchar()) != EOF) {
                  writer.putchar();
            }
      }
}
```

Исправим так чтобы класс CharCopier не зависел от классов Printer и Keyboard:

```
public interface Reader {
      char getchar();
}
public interface Writer {
      void putchar(char c)
}

public Keyboard implements Reader {...}
public Printer implements Writer {…}

class CharCopier {
      void copy(Reader reader, Writer writer) {
            int c;
            while ((c = reader.getchar()) != EOF) {
                  writer.putchar();
            }
      }
}
```

Задание на выполнение минипроекта

Придумайте тему своего минипроекта - текстового квеста. Например это могут быть квест по мотивам Скайрим или симулятор выживания или квест из какой либо области знаний, например информатики. По приведенному в разделе два образцу для своего квеста проведите проектирование. То есть:

- проведите анализ задания,
- синтез моделей,
- постройте диаграмму классов UML,
- создайте все нужные классы,
- продемонстрируйте работу вашего квеста.

В следующих занятиях вы будете развивать этот минипроект, и возможно у вас получится полноценная игра для Android.

Заключение

Даже если вы делаете маленькую программку, которой кроме вас, возможно, никто и не воспользуется, все равно не стоит пренебрегать проектированием. Этот этап поможет вам заранее устранить возможные противоречия в ТЗ, поможет уменьшить количество повторных разработок, которые возникают, когда выясняется, что изначально был выбран неверный/неэффективный способ решения задачи. Кроме того, вспомните, очень многие известные проекты начинались как маленькие программки для личного пользования! И именно грамотное проектирование позволило им вырасти во что-то серьезное.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Дмитрию Владимировичу Яценко.