Математические структуры и моделирование
2009, вып. 20, с. 178–190

УДК 004.045+372.862

ОТ ИМПЕРАТИВНОГО К ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПРОГРАММИРОВАНИЮ ВМЕСТЕ С JAVA И NETBEANS: ОБЪЕКТНАЯ ДЕКОМПОЗИЦИЯ И ИНКАПСУЛЯЦИЯ

Д.Н. Лавров

Основной проблемой при переходе от императивного к объектно-ориентированному программированию является изменение стиля мышления. Как изменить этот стиль за одно-два занятия и при этом продемонстрировать основные подходы, используемые при разработке, возникающие сложности и способы их решения — это цель данной статьи. На примере решения простейшей задачи мы ведем учащегося по дороге от умения составлять алгоритмы к способностям строить масштабируемую архитектуру приложения.

Введение

Объектно-ориентированное программирование (ООП) — это один из способов бороться с увеличивающейся сложностью программного обеспечения. Реальные ощутимые плоды этот подход дает на больших проектах, а на небольших задачах применение подхода приводит к увеличению объема кода. Учащиеся, видя увеличения объема, часто теряют мотивацию в использовании ООП. Эту мотивацию нужно создавать, объясняя, как важно в больших программных проектах правильно провести объектную декомпозицию.

К сожалению, приводить примеры больших проектов на лабораторных работах и выдавать задания больших объемов на самостоятельную разработку невозможно. Из опыта преподавания известно, что такие проекты, как правило, не завершаются удачно.

Учиться нужно на небольших примерах. Разобрав пример, необходимо дать похожее по уровню сложности задание учащемуся и проконтролировать его выполнение.

Основной критерий — правильная объектная декомпозиция задачи. К сожалению или к счастью, не существует единственно верного решения. Каждое из

Омский государственный университет.
E-mail: lavrov@omsu.ru

решений будет иметь свои достоинства и недостатки. Существуют откровенно неверные решения и решения, верные при рассмотрении с каких-либо позиций.

Пусть необходимо создать приложение, вычисляющее действительные корни квадратного уравнения с действительными коэффициентами.

Из каких соображений будем исходить при решении этой задачи:

1. Необходимо скрыть реализацию классов друг от друга и от программиста, пользователя класса. Другими словами, обеспечить инкапсуляцию.

2. Необходимо построить приложение, в котором логика выполняемой работы не смешивалась бы с интерфейсной частью. Здесь можно также говорить об инкапсуляции, но уже не на уровне класса, а на уровне всего приложения. Скрывается реализация логики работы приложения от интерфейсных классов.

1. Решение в императивном стиле

С чего начать? Попросим студентов создать программу на языке Java, которая решает задачу нахождения корней квадратного уравнения. Стиль интерфейса не важен, пусть это будет консольный ввод-вывод. Скорее всего получиться код, подобный следующему:

```
import static java.lang.Math.*;
import java.io.*;
public class SolverSquareEquation {
    public static void main(String[] args) throws IOException {
        int numberOfRoots;
        double a, b, c, x1=0, x2=0;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.print("a=");
        a=Double.parseDouble(br.readLine());
        System.out.print("b=");
        b=Double.parseDouble(br.readLine());
        System.out.print("c=");
        c=Double.parseDouble(br.readLine());
        if (a!=0) {
            double d=b*b-4*a*c;
            if        (d<0) {numberOfRoots=0;}
            else if  (d==0) {numberOfRoots=1;  x1=-b/(2*a); }
            else /*(d>0)*/  {numberOfRoots=2;  x1=(-b-sqrt(d))/(2*a);
                                               x2=(-b+sqrt(d))/(2*a);}
        } else /*(a==0)*/ {
            if      (b==0&&c==0)  {numberOfRoots=-1;}
            else if (b==0&&c!=0)  {numberOfRoots= 0;}
            else  /*(b!=0&&c!=0)*/{numberOfRoots= 1; x1=-c/b;}
        }
        switch (numberOfRoots){
```

```
            case -1: {System.out.println("X-любое число.");break;}
            case  0: {System.out.println("Корней нет.");break;}
            case  1: {System.out.println("X1=");break;}
            case  2: {System.out.println("X1="+x1+" X2="+x2);break;}
        }
    }
}
```

В этом примере разобраны все случаи входных данных. Конечно, не разобравший всех случаев учащийся должен исправиться, ведь программа должна работать на всех типах входных данных – и правильных, и неправильных.

Обратите внимание на то, что в данной программе сложное действие не разбито на более простые, и можно сказать, что программа на объектно-ориентированном языке скорее написана даже не в императивном, а в структурном стиле. Плохо ли это? Смотря для чего. Если цель просто решить квадратное уравнение, то нет. Но наша цель — освоить новый способ мышления, поэтому не будем останавливаться на достигнутом.

Что характеризует полученное решение:

- Это проектное решение невозможно тестировать, кроме как в «ручном» режиме, путем последовательного ввода данных и визуальной проверке полученных ответов.
- Это решение трудно поддерживать, оно не поддается стандартным методам рефакторинга [2].
- Это решение фактически не использует объектов, а значит, невозможно получить преимущества повторного использования и расширяемости без переписывания кода.
- Ни о каком разделении интерфейсной части кода и кода, представляющего решение основной задачи, речи не идет: код взаимодействия с консолью и код решения основной задачи приложения сплетены между собою, а следовательно, в будущем возникнут трудности при попытке смены интерфейса с консольного на оконный или web-интерфейс.

В таких случаях говорят, что приложение немасштабируемое.

Прежде чем исправить указанные недостатки, обсудим преимущества объектно-ориентированного подхода.

2. Суть объектно-ориентированного подхода

Основная идея заключается в построении модели на основе выделения понятий предметной области с последующим распределением обязанностей между ними. Программа, решающая нашу задачу, строится так, как задача решалась бы в жизни. По сути, строится модель предметной области, понятия предметной области представляются в виде классов, а их конкретные проявления в виде объектов. Распределяя обязанности между классами, проектировщик приложения

«оживляет» объекты, наделяет их поведением. Оживление (Animation) является основным принципом, используемым при построении архитектуры объектно-ориентированного приложения [1]. Проектировщик (программист) уподобляется богу, создающему искусственный мир по образу и подобию настоящего мира.

Моделирование осуществляется декомпозицией не по действиям (глаголам), как было принято в императивном программировании, а по сущностям (существительным). Сущности представляют собой, как правило, понятия предметной области, но иногда искусственные понятия, которых нет в явном виде в предметной области, но без которых невозможно создать масштабируемый код.

При оживлении пользователю объектов совсем не обязательно знать внутреннюю структуру объекта, так же как в реальной жизни пользователь обычно не знает, как в точности устроена микроволновая печь или телевизор, что не мешает ему разогревать еду и просматривать телепередачи. Скрытие внутренней реализации объекта от пользователя – это обычная практика, которая называется инкапсуляцией. Важна только функциональность объекта, его интерфейс. Такой подход позволяет ограничить распространение изменений в коде класса на все приложение. Это один из важнейших принципов объектно-ориентированного проектирования.

Необходимо сломать установку учащегося, которая скрывается за высказыванием: «*Но ведь программа работает правильно*». «Работает правильно» - это необходимое условие хорошей программы, но, к сожалению, недостаточное.

3. Объектная декомпозиция задачи

Класс представляет собой некую сущность, которая задает некоторое общее поведение для объектов. Разработка или, точнее, моделирование приложения состоит из двух этапов: анализа и проектирования. В большинстве современных процессов разработки результаты этих двух этапов описываются на графическом языке UML [3] в виде диаграмм.

Цель анализа — выявление объектов и описывающих их классов, представляющих важные, с точки зрения разработчика и заказчика, понятия предметной области. Обычно выделенные понятия формируют так называемую концептуальную модель, которая становится источником большинства классов и объектов разрабатываемого приложения [1].

Выделение объектов и понятий предметной области (будущих классов) часто основывается на выделении существительных из текстовых описаний работы будущего приложения. В нашем случае можно выделить три таких понятия (класса):

SquareEquation – квадратное уравнение;

Solution – решение квадратного уравнения;

SolverSquareEquation – тот кто, пользуется услугами решения квадратного уравнения. Это будет интерфейсный класс, который осуществляет взаимодействие с пользователем через консоль.

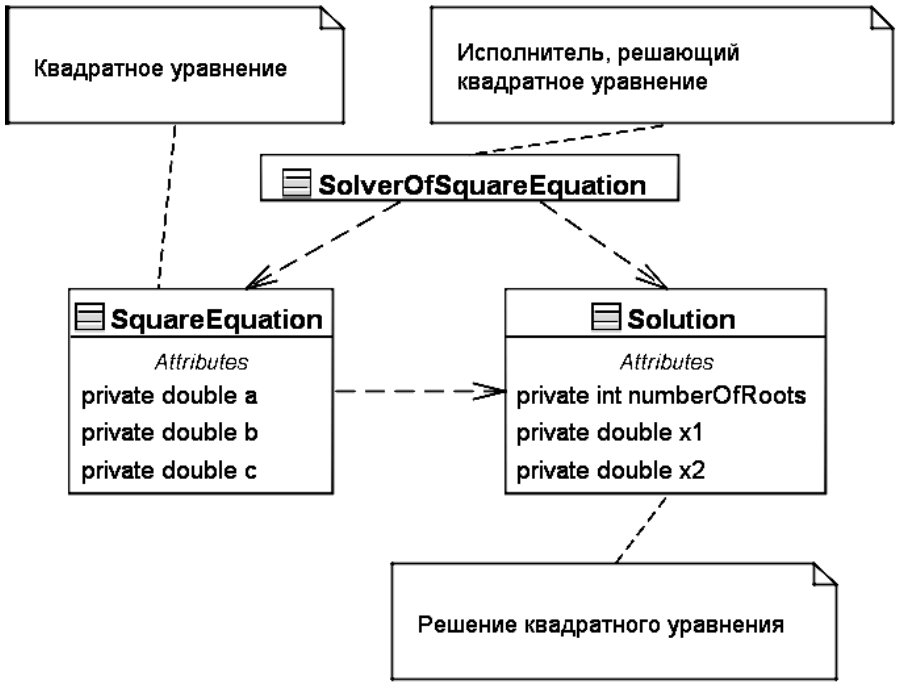

Рис. 1. Концептуальная модель задачи решения квадратного уравнения, представляется на диаграмме классов

Следующая задача этапа анализа — определение наиболее важных атрибутов и ассоциаций (связей) между понятиями. В нашем случае атрибуты достаточно очевидны, а постоянных ассоциаций нет.

Все полученные результаты сводятся на диаграмму классов, представляющую концептуальную модель приложения (рис. 1). На этой диаграмме пунктирные стрелки означают использование одного класса другим. Такие стрелки применяют, если необходимо указать, что экземпляр класса является локальной переменной, параметром или возвращаемым значением метода класса, от которого берет начало стрелка.

Цель проектирования — это распределение обязанностей между объектами для удовлетворения всех требований решаемой задачи. Результаты такого распределения представляются в виде UML-диаграмм взаимодействий, описывающих последовательность сообщений между объектами и UML-диаграммы классов, представляющих архитектуру будущего приложения. На этом этапе понятия предметной области превращаются в программные классы.

Итак, в нашей задаче на этапе проектирования необходимо решить, какому объекту (классу) поручить решать квадратное уравнение. Существует несколько вариантов решения этой задачи, но если необходимо сократить объем кода приложения, то нужно передать эту обязанность тому классу, который обладает наибольшим количеством информации для выполнения задачи. Такой способ решения поставленной задачи носит название шаблона распределения обязанностей Expert, относящегося к семейству шаблонов GRASP [1]. В нашем случае

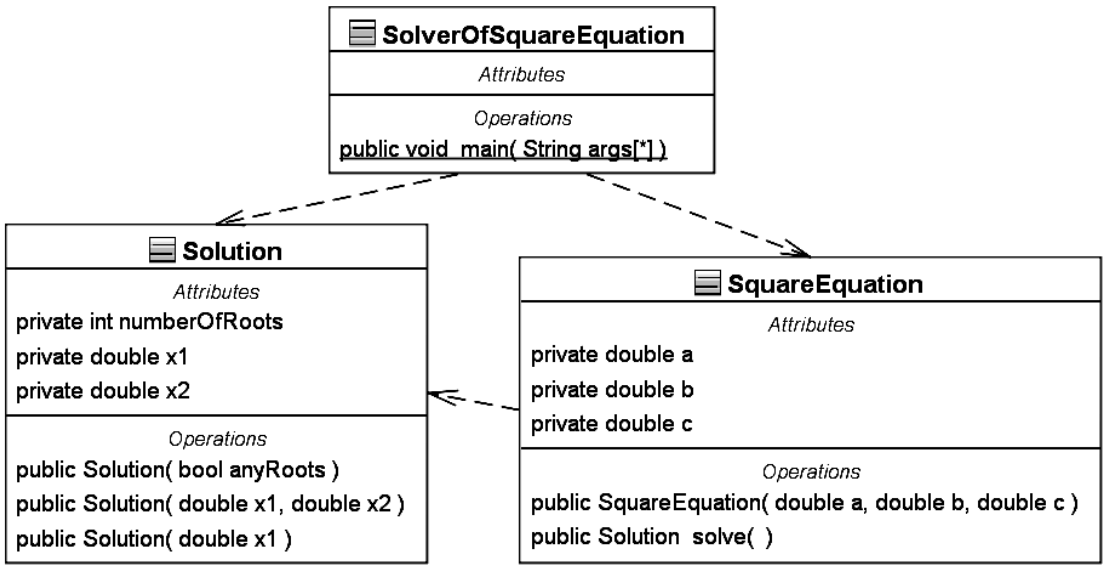

Рис. 2. UML-диаграмма классов, представляющая архитектуру приложения SolverSquareEquation

таким классом является класс SquareEquation, ведь именно он обладает всеми данными для получения решения квадратного уравнения. В результате метод solve() становиться методом SquareEquation.

Полезно разобрать с учащимися другие решения распределения обязанностей и посмотреть, к какому коду и каким проблемам могут эти решения привести.

Далее следует определить, какой класс будет отвечать за консольный интерфейс приложения. Наиболее подходящим для этих целей является класс SolverOfSquareEquation, представляющий все приложение в целом. Реализация ввода/вывода будет осуществляться в статическом методе main() так, что создавать экземпляр этого класса не нужно.

После распределения обязанностей получаем диаграмму классов (рис. 2).

Обратите внимание, что для решения нашей задачи нам не нужны все методы доступа и инициализации (методы типа get/set). Все лишнее должно быть отброшено, чтобы код был более ясным и не загромождался никому не нужными конструкциями.

Интегрированная среда разработки NetBeans[1] позволяет построить такую диаграмму классов, а затем сгенерировать по ней код, в котором останется только дописать реализации методов. Нормально, если в диаграмме классов будут опушены некоторые детали или будут присутствовать несущественные недоработки. В дальнейшем при реализации можно будет исправить эти недоработки в коде, а на следующей итерации разработки методом обратного реинженеринга (также реализован в NetBeans) скорректировать вышеупомянутую диаграмму классов.

[1] Свободно распространяемая среда разработки, доступная по адресу http://www.netbeans.org

После кодирования и внесения доработок получаем следующий код:

```
// модуль компиляции SquareEquation
import static java.lang.Math.*;
public class SquareEquation {
    private double a;
    private double b;
    private double c;

    public SquareEquation (double a, double b, double c) {
        this.a=a;
        this.b=b;
        this.c=c;
    }

    public Solution solve () {
       if (a!=0) {
            double d=b*b-4*a*c;
            if        (d<0) {return new Solution(false);}
            else if  (d==0) {return new Solution(-b/(2*a)); }
            else /*(d>0)*/  {return new Solution((-b-sqrt(d))/(2*a),
                                                 (-b+sqrt(d))/(2*a));}
        } else /*(a==0)*/ {
            if      (b==0&&c==0)  {return new Solution(true);}
            else if (b==0&&c!=0)  {return new Solution(false);}
            else  /*(b!=0&&c!=0)*/{return new Solution(-c/b);}
        }
    }
}

// модуль компиляции Solution.java
public class Solution {
    private int numberOfRoots;
    private double x1;
    private double x2;

    final static int ANY_NUMBER      = -1;
    final static int NOT_EXIST_ROOTS =  0;
    final static int ONE_ROOT        =  1;
    final static int TWO_ROOTS       =  2;

    public Solution (boolean anyRoots) {
        numberOfRoots=(anyRoots)?ANY_NUMBER:NOT_EXIST_ROOTS;
    }

    public Solution (double x1, double x2) {
        numberOfRoots=TWO_ROOTS;
        this.x1=x1;
```

```
        this.x2=x2;
    }

    public Solution (double x1) {
        numberOfRoots=ONE_ROOT;
        this.x1=x1;
    }

    public int getNumberOfRoots() {
        return numberOfRoots;
    }

    public double getX1() {
        if (numberOfRoots==ONE_ROOT||numberOfRoots==TWO_ROOTS) {
            return x1;
        } else {
            throw new Error("Sorry, X1 IS NOT EXIST OR ANY NUMBER.");
        }
    }

    public double getX2() {
        if (numberOfRoots==TWO_ROOTS) {
            return x2;
        } else {
            throw new Error("Sorry, X2 IS NOT EXIST OR ANY NUMBER.");
        }
    }
}

// модуль компиляции SolverOfSquareEquation.java
import java.io.*;
public class SolverOfSquareEquation {
    public static void main (String[] args) throws IOException {
        int numberOfRoots;
        double a, b, c, x1=0, x2=0;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.print("a=");
        a=Double.parseDouble(br.readLine());
        System.out.print("b=");
        b=Double.parseDouble(br.readLine());
        System.out.print("c=");
        c=Double.parseDouble(br.readLine());
        SquareEquation sq=new SquareEquation(a, b, c);
        Solution sol=sq.solve();
        switch (sol.getNumberOfRoots()){
          case Solution.ANY_NUMBER:
               {System.out.println("X-любое число.");break;}
          case Solution.NOT_EXIST_ROOTS:
```

a	b	c	numberOfRoots	x1	x2
1	2	1	1	-1	–
1	-2	1	1	1	–
2	-7	6	2	1.5	2
1	0	-4	2	-2	2
0	0	0	∞	–	–
0	1	2	1	-2	–
0	0	1	0	–	–
0	1	0	1	0	–
2	-3.5	1	2	$\approx$0.35961	$\approx$1.39039

Таблица 1. Набор тестов для метода solve()

```
                {System.out.println("Корней нет.");break;}
            case Solution.ONE_ROOT:
                {System.out.println("X1="+sol.getX1());break;}
            case Solution.TWO_ROOTS:
                {System.out.println("X="+sol.getX1()+" X2="+sol.getX2());break;}
        }
    }
}
```

Класс Solution обзавелся тремя конструкторами для реализации принципа инкапсуляции. Первый конструктор имеет один аргумент типа double, через который передается значение корня в случае единственного решения. Второй конструктор имеет два параметра для передачи значений двух корней. И третий конструктор необходим для представления двух случаев: отсутствия корней и ситуации, когда корнем является любое число.

В Solution прописаны четыре статические константы, определяющие количество корней и используемые для повышения читаемости кода. Определены три метода доступа, позволяющие классу SolverOfSquareEquation отобразить полученные результаты. Для защиты пользователя класса от случайных ошибок во время разработки в методах доступа стоит проверка корректности операций.

4. Тестирование классов

При тестировании необходимо проверить работу методов классов. Автоматическое тестирование необходимо не столько для проверки правильности работы модулей, сколько для проведения последующего рефакторинга [2]. Кроме того, часто при создании и выполнении тестов становится понятно, что проектирование можно было выполнить еще проще.

Покажем, как это можно сделать на примере метода solve().

Составим таблицу тестов, проверяющую все варианты решения (табл. 1).

Воспользуемся инструментом JUnit (запускается из контекстного меню проекта, раздел Tools). Этот инструмент генерирует набор классов для тестирования классов нашего приложения. После генерации можем использовать семей-

ство набор assert-команд для сравнения ожидаемого результата с вычисленным. Но для правильной работы системы автоматического тестирования необходимо реализовать дополнительный метод equals(), унаследованный от Object.

Метод equals() используется командами типа assertEquals() для сравнения классов по значениям (по содержимому полей атрибутов, а не по ссылкам). Другими словами, equals() реализует сравнение идентичности объектов с точки зрения предметной области, в то время как оператор == с точки зрения программной реализации.

Если переписать подходящим образом метод toString() в классе Solution, также унаследованный от Object, то при ошибке тестирования можно будет увидеть не хэш-значения классов, на которых произошел сбой, а сами значения. Кроме того, можно хорошо сэкономить в SolverOfSquareEquation: код с оператором выбора switch переносится в toString(), а в классе Solution отпадает необходимость в реализации методов доступа и отслеживании корректности их вывода. Код становится более простым и прозрачным.

Последний вариант изменений приведен ниже.

```
// Изменения в Solution
public class Solution {
    ...
    // Описание атрибутов и конструкторов осталось без изменений
    // Константы можно сделать закрытыми
    // Удалены все методы доступа типа getX()
    // Для повышения читабельности кода вводим две константы
    public static boolean ROOT_IS_ANY_NUMBER=true;
    public static boolean ROOT_ISNT_EXIST=false;

@Override
    public boolean equals(Object obj) {
        double eps = 1e-5;
        if (obj == null) { return false; }
        if (getClass() != obj.getClass())  { return false; }
        final Solution other = (Solution) obj;
        if (this.numberOfRoots != other.numberOfRoots) { return false; }
        if (this.numberOfRoots==ONE_ROOT) {
               return abs(this.x1-other.x1)<eps
        }
        if(this.numberOfRoots==TWO_ROOTS){
               return abs(this.x1-other.x1)<eps && abs(this.x2-other.x2)<eps;
        }
        return true;
    }

    @Override
    public String toString() {
        String s=null;
          switch (getNumberOfRoots()){
```

```
            case Solution.ANY_NUMBER:
                 {s="X is any number.";break;}
            case Solution.NOT_EXIST_ROOTS:
                 {s="Roots not exist";break;}
            case Solution.ONE_ROOT:
                 {s="X="+x1;break;}
            case Solution.TWO_ROOTS:
                 {s="X1="+x1+" X2="+x2;break;}
        }
        return s;
    }
...
// Изменения в SolverOfSquareEquation
public class SolverOfSquareEquation {
    public static void main (String[] args) throws IOException {
        int numberOfRoots;
        double a, b, c;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.print("a=");
        a=Double.parseDouble(br.readLine());
        System.out.print("b=");
        b=Double.parseDouble(br.readLine());
        System.out.print("c=");
        c=Double.parseDouble(br.readLine());
        System.out.println(new SquareEquation(a, b, c).solve());
    }
}
```

В этом варианте описание атрибутов и конструкторов осталось без изменений. Статические константы: ANY_NUMBER, ONE_ROOT, TWO_ROOTS, NOT_EXIST_ROOTS можно сделать закрытыми, окончательно скрыв внутреннее устройство класса. Удалены все методы доступа типа getX(), теперь в них нет необходимости — их действия заменяет один метод toString().

Далее приведен один из возможных способов реализации метода тестового класса.

```
    public void testSolve() {
        System.out.println("solve");
        SquareEquation[] instance =
            {new SquareEquation( 1, 2, 1),
             new SquareEquation( 1,-2, 1),
             new SquareEquation( 2,-7, 6),
             new SquareEquation( 1, 0,-4),
             new SquareEquation( 0, 0, 0),
             new SquareEquation( 0, 1, 2),
             new SquareEquation( 0, 0, 1),
             new SquareEquation( 0, 1, 0),
             new SquareEquation( 2,-3.5,1),
```

```
        };
    Solution[] expResult = {
        new Solution(-1),
        new Solution(1),
        new Solution(1.5,2),
        new Solution(-2,2),
        new Solution(Solution.ROOT_IS_ANY_NUMBER),
        new Solution(-2),
        new Solution(Solution.ROOT_ISNT_EXIST),
        new Solution(0),
        new Solution(0.35961,1.39039),
    };
    Solution result = null;
    for (int i=0;i<instance.length;i++) {
        result=instance[i].solve();
        assertEquals(expResult[i], result);
    }
}
```

Запуск теста должен производиться при каждом внесении изменения в код реализации метода solve(), для того чтобы убедиться в том, что функциональность не пострадала от внесенных в код поправок. В NetBeans это легко осуществить с помощью комбинации Alt+F6.

Последний вариант диаграммы классов приведен на рисунке 3.

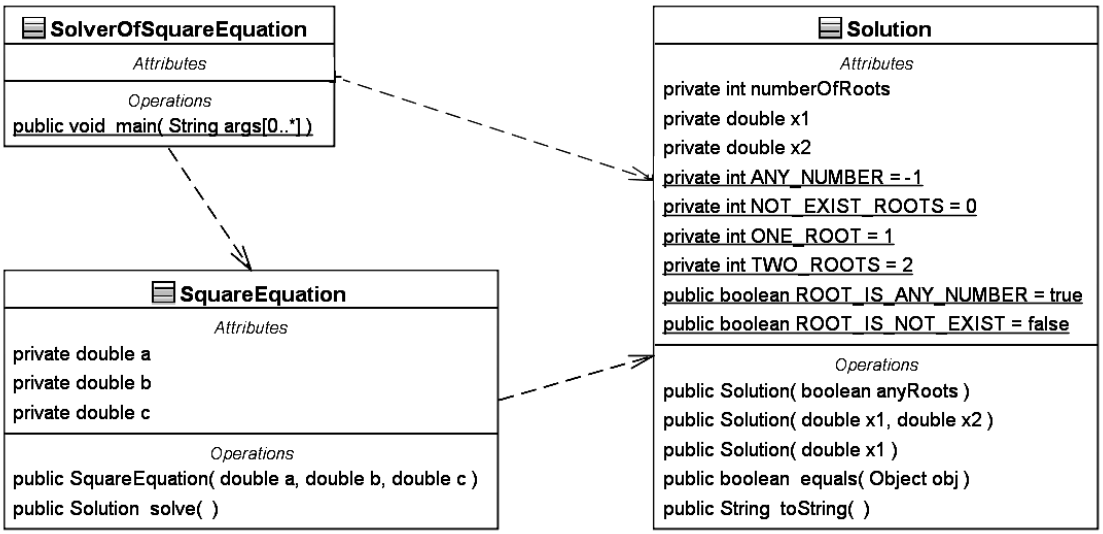

Рис. 3. Итоговая UML-диаграмма классов, представляющая архитектуру приложения SolverSquareEquation. Получена с помощью функции обратного реинженеринга в NetBeans

5. Заключение

Теперь пришло время взглянуть на диаграммы классов, которые были получены в процессе разработки (рисунки 1, 2, 3). Видим, что процесс проектирования — творческий процесс. Понятия предметной области превратились в классы и обросли обязанностями. Учащийся должен взглянуть еще раз на весь процесс, чтобы увидеть все промежуточные решения, увидеть цель — простой и ясный код, оптимальную с точки зрения нашей постановки задачи архитектуру. Необходимо еще раз указать на все преимущества и недостатки этих решений.

Наш пример позволяет сконцентрироваться на двух важных моментах: принципе инкапсуляции и автоматизации тестирования модулей. Кроме того, удалось отделить классы уровня интерфейса от классов уровня логики работы приложения [1]

Возможны и другие решения данной задачи, исходящие из других принципов проектирования. Не существует единственно верного проектного решения.

Можно продолжать развитие этого примера, постепенно усложняя задачу. Например, потребовать, чтобы наше приложение работало с любыми типами многочленов, тогда можно показать необходимость введения наследования и реализации полиморфного поведения классов нахождения корней. Интегрированная среда разработки NetBeans, имея встроенные средства рефакторинга, может помочь преобразовать наше приложение в расширяемый вид и в этом случае.

В заключение, хочу поблагодарить доцента кафедры информационной безопасности Надежду Федоровну Богаченко и ведущего программиста фирмы Luxsoft Максима Потанина, за обсуждение примера, приведенного в данной статье.

Литература

1. Ларман, К. Применение UML и шаблонов проектирования / К. Ларман. 2-е издание. – М.: Издательский дом «Вильямс», 2004. – 624 с.
2. Фаулер, М. Рефакторинг: улучшение существующего кода / М. Фаулер. – СПб: Символ-Плюс, 2003. – 432 с.
3. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. – М.: ДМК Пресс. Питер, 2004. – 430 с.