Team43OSD

1644703/1 Petr Philipp
1644841/1 Samuel Godwin
1644952/1 Petko Daskalov
1645121/1 Mohammed Al Kawsar
1645208/1 Prawee Wongphayabal

5CCS2OSD Coursework, 2017

November 28, 2017

Introduction

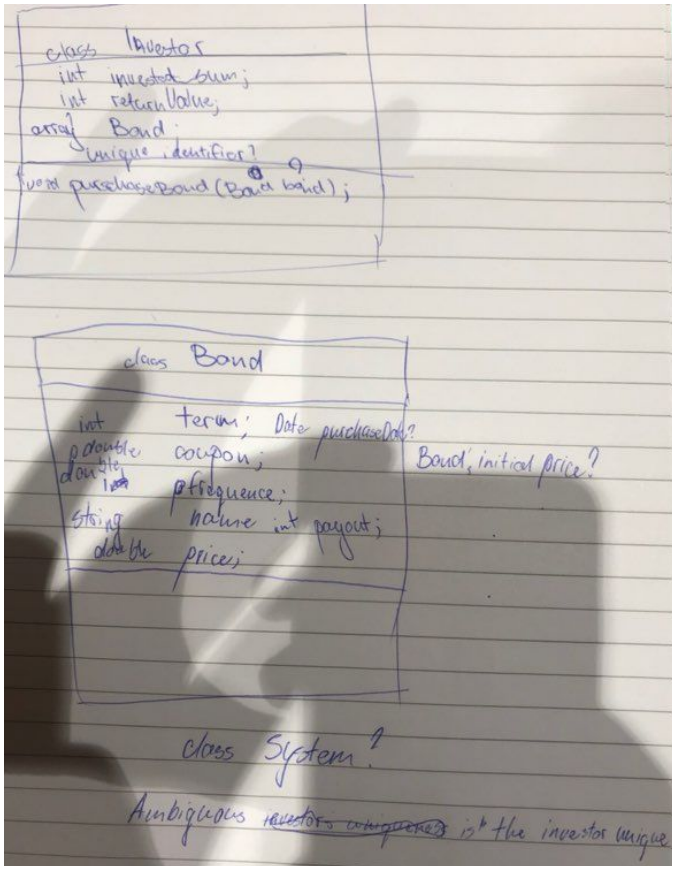
We began our project by carrying out requirements elicitation and documented system functionalities as use cases, all according to the brief. During this process however, we came across a number of ambiguous or incomplete requirements. To overcome this, we made a couple of assumptions to be able to progress with the coursework and create our system.

We initially agreed to have at the very least two classes: *Investor* and *Bond*. We knew that an investor would need to conceptually have ownership of a collection of bonds. Due to the aforementioned ambiguous/incomplete requirements we had to decide what collection we would use for this bonds collection (i.e. array or ArrayList. We decided on the latter, for ease of implementation). We also discussed and decided that we would have a unique identifier for investor class, as well as a date 'purchaseDate' for objects of *Bond*.

We additionally discussed having Bond's *initial* price as a field in the class Bond. On top of this we pondered having a class *System* for the overall system.

What we eventually agreed on was a third class, called *Brain* in our final implementation, containing an instance of *Investor* and an ArrayList of Bonds. In concept, it acts as a portfolio of a particular investor.

From the first meeting we had, we came up with the following mock class diagram. This does not include details of operations:

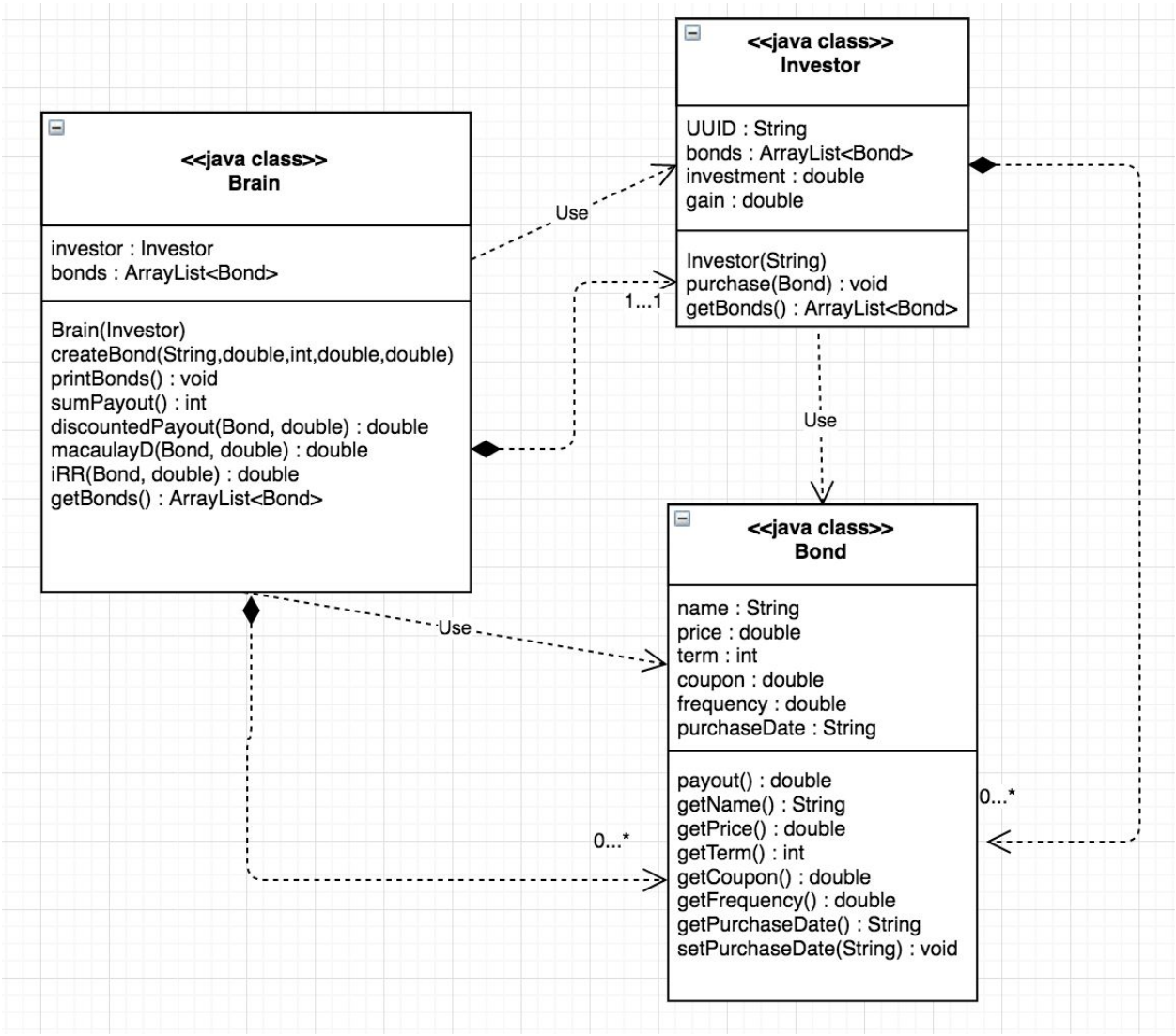


Description of Team Member Roles and Contributions to Project

Across our meetings, we decided on a number of roles for the members in our team. As team leader I (Samuel Godwin) assigned class diagrams to Prawee. I assigned the role of writing pseudocode and also the report writing to myself. The use case and architecture diagrams were done by Prawee and Mohammed. The majority of the coding was done by Petr; some of the coding was also done by Petko, who built our function for internal rate of return (irr). However, much of the programming would be taken with a mob programming approach in our meetings, as significant discussion and a close following of our initial designs would be needed.

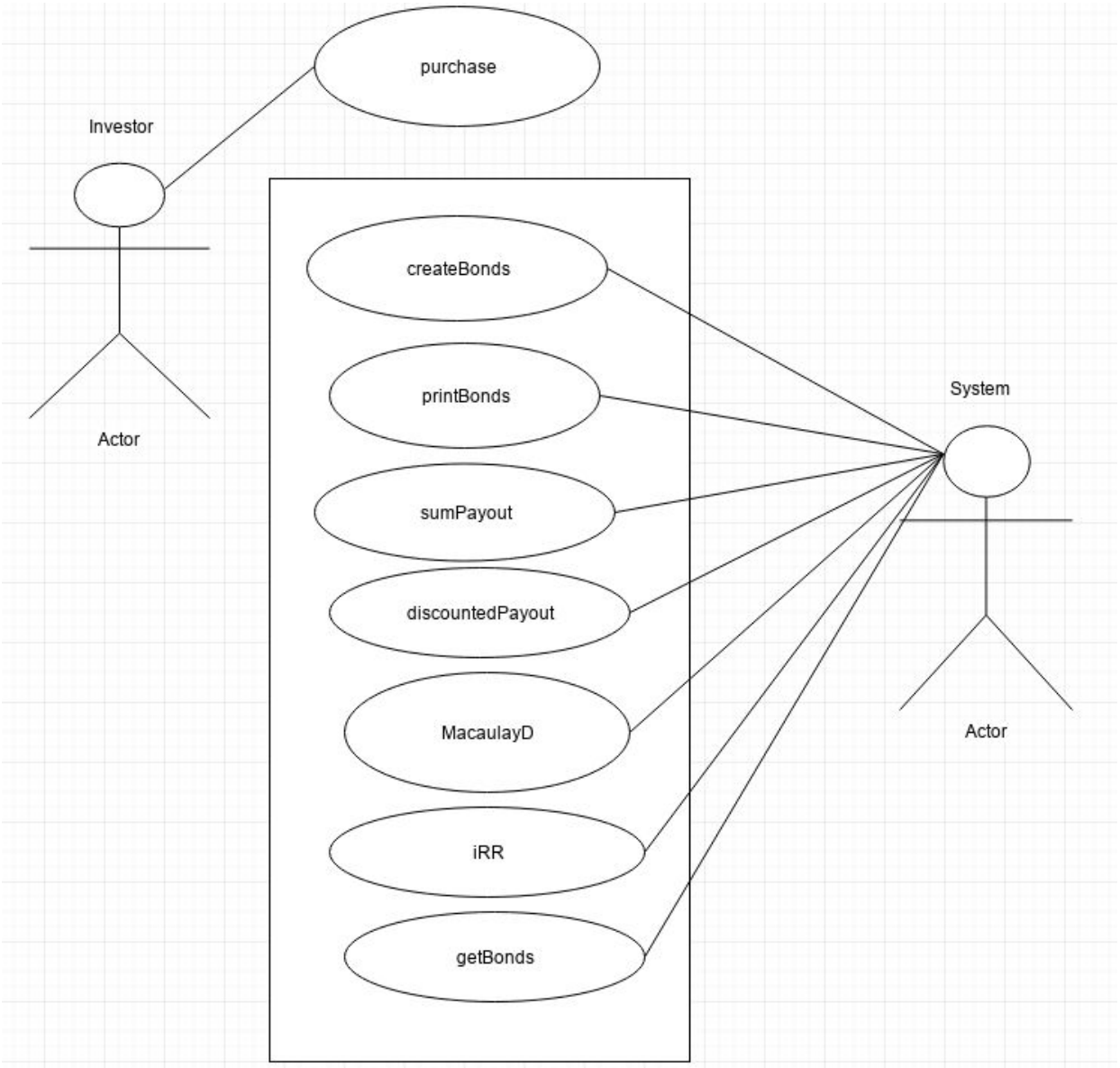
Class Diagram

Below is our final class diagram.



Use Case Diagram

Below is our final use case diagram.



Pseudocode For Each Class

Investor Class

```
CLASS 'Investor':
        Create field 'INVESTMENT' of type Integer
        Create field 'gain' of type Double
        Create field 'UUID' of type String
        Create field 'bonds' as ArrayList of Bonds
        CONSTRUCTOR METHOD 'Investor' <UUID>:
                Set field 'UUID' to <UUID>
        END CONSTRUCTOR METHOD.
        METHOD 'purchase' <bond>:
                Add <bond> to 'bonds' collection
                Set 'fraction' in <bond> to result of INVESTMENT/bond.getPrice
                Set 'purchaseDate' in <bond> to Date
        END METHOD.
        METHOD 'payoutF' <b>:
                Return result of calculation b.payout * b.getFraction
        END METHOD.
        METHOD 'getBonds':
                Return bonds
        END METHOD.
END CLASS.
```

Bond Class

CLASS ‘Bond’:

Create field ‘name’ of type String

Create field ‘price’ of type Double

Create field ‘term’ of type Integer

Create field ‘coupon’ of type Double

Create field ‘frequency’ of type Double

Create field ‘purchaseDate’ of type String

Create field ‘fraction’ of type Double

CONSTRUCTOR METHOD ‘Bond’ <name, price, term, coupon, frequency>:

Set field ‘name’ to <name>

Set field ‘price’ to <price>

Set field ‘term’ to <term>

Set field ‘coupon’ to <coupon>

Set field ‘frequency’ to <frequency>

END CONSTRUCTOR METHOD.

METHOD ‘payout’:

Return result of calculation (price + price * coupon * frequency * term)

END METHOD.

END CLASS.

Brain Class

CLASS ‘Brain’:

Create object field ‘investor’ of type Investor

Create field ‘bonds’ as ArrayList of Bonds

CONSTRUCTOR METHOD ‘Brain’ <inv>:

Set field ‘investor’ to <inv>

END CONSTRUCTOR METHOD.

METHOD ‘createBond’ <name, price, term, coupon, frequency>:

Add bond to ‘bonds’ collection with: <name, price, term, coupon, frequency>

END METHOD.

METHOD ‘printBonds’:

Print as output all bonds in collection ‘bonds’

END METHOD.

METHOD ‘sumPayout’:

Create local variable ‘sum’ of type Double, initialise as 0

FOR EACH Bond from ‘bonds’ in ‘investor’: sum += investor.payoutF(bond)

ENDFOR.

Return sum

END METHOD.

METHOD ‘discountedPayout’ <bond, r>:

Create local variable ‘sum’ of type Double, initialise as 0

FOR i = 1; i < bond.getTerm; increment i:

sum += (investor.getINVESTMENT*bond.getCoupon/Math.pow(1 + r, i))

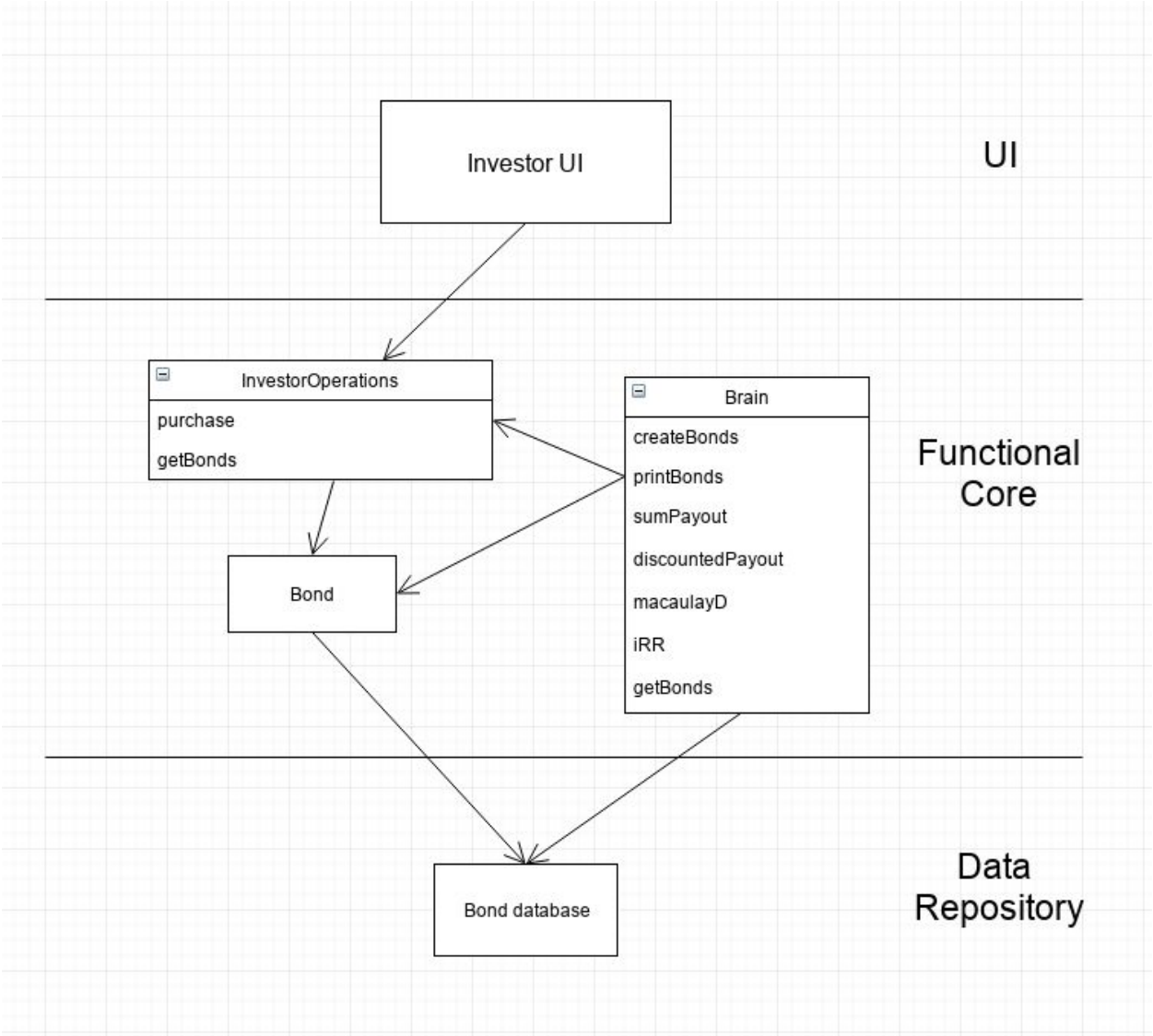
ENDFOR.

sum += ((investor.getINVESTMENT*bond.getCoupon + investor.getINVESTMENT)/Math.pow(1 + r, bond.getTerm))

```
        Return sum
    END METHOD.
    METHOD 'macaulayD' <bond, rate>:
        Create local variable 'temp' of type Double, initialise as 0
        FOR i = 1; i < bond.getTerm; increment i:
            temp += (i*bond.getCoupon*100/Math.pow(1 + rate,i))
        ENDFOR.
        Return result of (temp + ((bond.getTerm*100)/Math.pow(1 + rate,bond.getTerm))) /
        discountedPayout(bond, rate)
    END METHOD.
    METHOD 'iRR' <bond, r>:
        Initialise local variable 'rate', of type Double, as <r>
        Initialise local variable 'price' as result of 'getPrice' in 'bond'
        Initialise local variable 'i', of type Integer, as 0
        WHILE i < 50000:
            IF result of 'discountedPayout(bond, rate)' < price THEN:        rate -= 0.00001
            ELSE:    rate += 0.00001
            ENDIF.
            Increment i
        ENDWHILE.  Return rate
    END METHOD.
    METHOD 'getBonds':
        Return bonds
    END METHOD.
END CLASS.
```

Architecture Diagram

Below is our architecture diagram, which also assumes potentially future implementation of a more aesthetic user interface (for this, we currently use a basic command line interface).



Test Cases and Results

Here are the test cases we tested, as can be seen on KEATS. Our tests cover each of a value of 0.05 for coupon and also a value of 0.02 for coupon.

```
1) term = 5, coupon = 5, price = 103
payout = 125
value(0.05) = 99.999
value(0.02) = 114.14
macaulayDuration(0.05) = 4.54
macaulayDuration(0.02) = 4.578
irr = 0.0432

2) term = 10, coupon = 4, price = 95
payout = 140
value(0.05) = 92.278
value(0.02) = 117.965
macaulayDuration(0.05) = 8.359
macaulayDuration(0.02) = 8.579
irr = 0.0463

3) term = 20, coupon = 3, price = 92
payout = 160
value(0.05) = 75.075
value(0.02) = 116.35
macaulayDuration(0.05) = 14.47
macaulayDuration(0.02) = 15.71
irr = 0.0356

4) term = 15, coupon = 2, price = 120
payout = 130
value(0.05) = 68.86
value(0.02) = 99.99
macaulayDuration(0.05) = 12.61
macaulayDuration(0.02) = 13.1
irr = 0.006
```

Here is a *Main* class to act as our driver and also used primarily for the above test cases. The test cases themselves are highlighted in yellow.

```
import java.util.ArrayList;
import java.util.Scanner;
public class Main {
public static void main(String[] args) {
        Investor petr = new Investor("PH_97");
        Brain sys = new Brain(petr);
        sys.createBond("UKbond", 100, 5, 0.10, 1);
        sys.createBond("USAbond", 103, 5, 0.05, 1);
        sys.createBond("CZbond", 95, 10, 0.04, 1);
        sys.createBond("DEbond", 92, 20, 0.03, 1);
        sys.createBond("RUbond", 120, 15, 0.02, 1);
        System.out.println("Hello, here is a list of commands");
        System.out.println(
                "exit - to close the application \n" +
                "bonds - to show all available bonds and purchase\n" +
                "my bonds - to show all purchased bonds \n" +
                "create - to create bond \n"+
                "back - to navigate to main menu \n" +
                "payout - to show values of your bonds with r 0.05");

        Scanner scan =  new Scanner(System.in);
        String inp = scan.next();
```

```
        while (!inp.equals("exit")) {
        if (inp.equals("bonds")) {
        System.out.println("choose your bond by number");
        sys.printBonds();
        inp = scan.next();
        while (!inp.equals("back")) {
        petr.purchase(sys.getBonds().get(Integer.parseInt(inp)));
        System.out.println("your bonds: " + petr.getBonds() +"\n");
        sys.printBonds();
        inp = scan.next();
        }
    } else if (inp.equals("mybonds")) {
        if (petr.getBonds().size() == 0) {
        System.out.println("you have no bonds");
        }
        else System.out.println(petr);
        } else if (inp.equals("create")) {
        System.out.println("please enter info in format x,y, .. \n " + "name,price,term,coupon,frequency");
        inp = scan.next();
        String[] a = inp.split(",");
        sys.createBond(a[0],Double.parseDouble(a[1]),Integer.parseInt(a[2]),
                                Double.parseDouble(a[3]), Double.parseDouble(a[4]));
        System.out.println("new bond created : " + sys.getBonds().get(sys.getBonds().size()-1));
        }
```

```
else if (inp.equals("payout")) {
if (petr.getBonds().size() == 0) {
System.out.println("you have no bonds");
}
else {
for (Bond b : petr.getBonds()) {
            System.out.println("payout is : " + petr.payoutF(b));
            System.out.println("discounted payout is: " + sys.discountedPayout(b, 0.05));
            System.out.println("macaulayD is : " + sys.macaulayD(b, 0.05));
            System.out.println("IRR is : " + sys.irr(b, 0.1) + "\n");
}}}
inp = scan.next();
}}}
```

Below are the results of the above test cases, in screenshot form. The screenshots also show the rest of the output in our program, including our use of scanner for user input and interaction.

```
Hello, here is a list of commands
exit - to close the application
bonds - to show all available bonds and purchase
my bonds - to show all purchased bonds
create - to create bond
back - to navigate to main menu
payout - to show values of your bonds with r 0.05
mybonds
you have no bonds
payout
you have no bonds
bonds
choose your bond by number
0) Bond: name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0
1) Bond: name : USAbond price: 103.0 term: 5 coupon: 0.05 frequency: 1.0
2) Bond: name : CZbond price: 95.0 term: 10 coupon: 0.04 frequency: 1.0
3) Bond: name : DEbond price: 92.0 term: 20 coupon: 0.03 frequency: 1.0
4) Bond: name : RUbond price: 120.0 term: 15 coupon: 0.02 frequency: 1.0
back
create
please enter info in format x,y, ..
 name,price,term,coupon,frequency
newBond,111,2,3,4
new bond created : name : newBond price: 111.0 term: 2 coupon: 3.0 frequency: 4.0
```

```
bonds
choose your bond by number
0) Bond: name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0
1) Bond: name : USAbond price: 103.0 term: 5 coupon: 0.05 frequency: 1.0
2) Bond: name : CZbond price: 95.0 term: 10 coupon: 0.04 frequency: 1.0
3) Bond: name : DEbond price: 92.0 term: 20 coupon: 0.03 frequency: 1.0
4) Bond: name : RUbond price: 120.0 term: 15 coupon: 0.02 frequency: 1.0
5) Bond: name : newBond price: 111.0 term: 2 coupon: 3.0 frequency: 4.0
0
your bonds: [name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0]

0) Bond: name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0
1) Bond: name : USAbond price: 103.0 term: 5 coupon: 0.05 frequency: 1.0
2) Bond: name : CZbond price: 95.0 term: 10 coupon: 0.04 frequency: 1.0
3) Bond: name : DEbond price: 92.0 term: 20 coupon: 0.03 frequency: 1.0
4) Bond: name : RUbond price: 120.0 term: 15 coupon: 0.02 frequency: 1.0
5) Bond: name : newBond price: 111.0 term: 2 coupon: 3.0 frequency: 4.0
1
your bonds: [name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0, name : USAb

0) Bond: name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0
1) Bond: name : USAbond price: 103.0 term: 5 coupon: 0.05 frequency: 1.0
2) Bond: name : CZbond price: 95.0 term: 10 coupon: 0.04 frequency: 1.0
3) Bond: name : DEbond price: 92.0 term: 20 coupon: 0.03 frequency: 1.0
4) Bond: name : RUbond price: 120.0 term: 15 coupon: 0.02 frequency: 1.0
5) Bond: name : newBond price: 111.0 term: 2 coupon: 3.0 frequency: 4.0
2
your bonds: [name : UKbond price: 100.0 term: 5 coupon: 0.1 frequency: 1.0, name : USAb
```

Shown first in the screenshot below is the results for test case 1, followed by tests 2, 3 and 4 respectively. As can be seen from the screenshots, the output of our program for these tests presents the exact same results as expected, if truncated to 4 decimal places. Keeping potentially more accuracy, we chose not to truncate these output values for extra clarity in our screenshots while we discuss this, while it is something we are very much aware of.

```
mybonds
UUID: PH_97 Bonds: [name : UKbond price: 100.0 te
payout
payout is : 150.0
discounted payout is: 121.64738335315407
macaulayD is : 4.2534989519347235
IRR is : 0.1

payout is : 125.0
discounted payout is: 99.99999999999997
macaulayD is : 4.54595050416236
IRR is : 0.043200000000008634

payout is : 140.0
discounted payout is: 92.27826507081517
macaulayD is : 8.359589164010508
IRR is : 0.0463600000000096

payout is : 159.99999999999997
discounted payout is: 75.07557931491999
macaulayD is : 14.473825547456943
IRR is : 0.035660000000006326

payout is : 130.0
discounted payout is: 68.86102588545819
macaulayD is : 12.617602115465148
IRR is : 0.006020000000006003

exit
```

Code Listing of Implemented Classes and Use Case Code

Bond Class

```
public class Bond {


private String name;

private double price;

private int term; //number of years untill expiry

private double coupon; // interest

private double frequency;

private String purchaseDate;

private double fraction;


public Bond(String name, double price, int term, double coupon, double frequency){

        this.name = name;

        this.price = price;

        this.term = term;

        this.coupon = coupon;

        this.frequency = frequency;

}

double payout() {

        return price + price * coupon * frequency * term;

}
```

```
public String getName() {
        return name;
}
public double getPrice() {
        return price;
}
public int getTerm() {
        return term;
}
public double getCoupon() {
        return coupon;
}
public double getFraction() { return fraction; }
public double getFrequency() {
        return frequency;
}
public String getPurchaseDate() { return purchaseDate; }
public void setFraction(double fraction) { this.fraction = fraction; }
public void setPurchaseDate(String purchaseDate) {
        this.purchaseDate = purchaseDate;
}
@Override
public String toString() {
        return "name : " + name + " price: " + price + " term: " + term + " coupon: " + coupon + " frequency: " +
frequency;
}}
```

Investor Class

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;

public class Investor {
private String UUID;
private ArrayList<Bond> bonds = new ArrayList<Bond>();
private  final int INVESTMENT = 100;
private double gain;

public Investor(String UUID) {
        this.UUID = UUID;
}

public int getINVESTMENT() {
        return INVESTMENT;
}

public void purchase(Bond bond) {
        bonds.add(bond);
    bond.setFraction(INVESTMENT/bond.getPrice());
        bond.setPurchaseDate(new SimpleDateFormat("yyyy-MM-dd").format(new Date()));
}
```

```
double payoutF(Bond b) {
        return b.payout() * b.getFraction();
}
public ArrayList<Bond> getBonds() {
        return bonds;
}
@Override
public String toString() {
        return "UUID: " + UUID + " Bonds: " + bonds.toString();
}}
```

Brain Class

```
import java.util.ArrayList;
import java.math.*;

public class Brain {
Investor investor;
ArrayList<Bond> bonds = new ArrayList<>();

public Brain(Investor inv) {
        this.investor = inv;
}

public void createBond(String name, double price, int term, double coupon, double frequency) {
        bonds.add(new Bond(name, price, term, coupon, frequency));
}
```

```
public void printBonds() {
        for (int i = 0; i < bonds.size(); i++) {
        System.out.println(i + ") " + "Bond: " + bonds.get(i));
        }
}

int sumPayout() {
        int sum = 0;
        for (Bond i : investor.getBonds()) {
        sum += investor.payoutF(i);
        }
        return sum;
}
double discountedPayout(Bond bond, double r) {
        double sum = 0;
        for (int i = 1; i < bond.getTerm(); ++i) {
        sum += (investor.getINVESTMENT()*bond.getCoupon()/Math.pow(1 + r, i));
        }
        sum += ((investor.getINVESTMENT()*bond.getCoupon() + investor.getINVESTMENT())/Math.pow(1 + r,
bond.getTerm()));
        return sum;
}

public double macaulayD(Bond bond, double rate) {
        double temp = 0;
```

```
        for (int i = 1; i <= bond.getTerm(); i++) {
        temp += (i*bond.getCoupon()*100/Math.pow(1 + rate,i));
        }
        return (temp + ((bond.getTerm()*100)/Math.pow(1 + rate,bond.getTerm()))) / discountedPayout(bond,
rate);
}
public double irr (Bond bond, double r) {
        double rate = r;
        double price = bond.getPrice();
        int i = 0;

        while( i < 50000) {
        if (discountedPayout(bond,rate) < price) {
        rate -= 0.00001;} else { rate += 0.00001; }
        i++;
        }
        return rate;
}
public ArrayList<Bond> getBonds() {
        return bonds;
}
@Override
public String toString() {
        return "All Bonds: " + bonds.toString() + " Investor: " + investor;
}}
```