5CCS2OSD Coursework 2017

Due: November 30th, 2017

Group 52

Junaid Mahmood (1680371) | Iris Radoi (1756682) | Mia King (1767801) | Thomas Nemeh (176921) | Philip Alexandrov (1771590)

Table of Contents

Requirements Analysis & Initial Specification

Ambiguous or Incomplete Requirements

1. Unclear how to identify unique bonds

   Research indicates that bond names like “UK government bond” are not unique titles and that an additional ID would need to be attributed to each bond for a unique reference. We have included this as an identity attribute within the bond class of a type String.

2. Unclear how to identify unique investors

   Research indicates that a given investor will often be one amongst many managed by a single trader / company and will need a unique identifier. We have included this as an identity attribute within the investor class of a type String.

3. Unclear how payments are processed

   For the sake of our model, we are assuming that the third party is responsible for ensuring the investor receives the payments they are due at the proper time. For this purpose we have included a sendPayment use case. Within this use case, we have also added the capability of the system to remove bonds from an investor's portfolio once they have received all necessary payments which we felt to be implied but not explicitly stated in the project outline.

4. Unclear how interest rate (r) behaves throughout the course of a bond’s life

   We are working under the assumption that r remains constant from the point the bond is purchased until when it expires.

5. Unclear how bond price is set / given

   Research indicated that the price of a bond is a result of movement in the market. Bonds with higher demand and lower supply will be priced higher and bonds with lower demand and higher supply will be priced lower. The price of a bond is therefore set by the market and cannot always be intrinsically calculated. Though in some cases you might say that investment is a part of price, for the sake of our model, we are treating them as two separate entities.

6. Unclear how time is measured

Research indicates that payments on bonds, based upon their coupon amount, are typically executed on an annual basis. Therefore we have taken time to be measured in years. We are interpreting a term value to be a number of years and a frequency of 1 to mean that an investor would be paid once per year for the number of years equal to the bond's term.

7. Unclear how coupon is valued

   We were not sure whether we should treat coupon as a decimal or an integer that should then be converted to a percentage. We decided to use the professor's test cases as an example and have our coupon be an integer input that we then interpret as a percentage as if we were to divide it by 100.

8. Missing: actor who will perform the operations on the bond like calculating internal rate of return, etc.

   Research indicates that the investor simply owns the portfolio, but does not always have the proper certification to trade and manage the bonds and financial entities in their portfolio. This is where a third party, like a wealth manager or a trader comes into the equation. It is these individuals who do all of the calculations specified by the system to inform their clients, the investors, of what they can expect from their bonds. This actor is reflected in our case diagram as the second actor although they do not appear in the class diagram.

9. Missing: operation to compute the value of a bond's payment

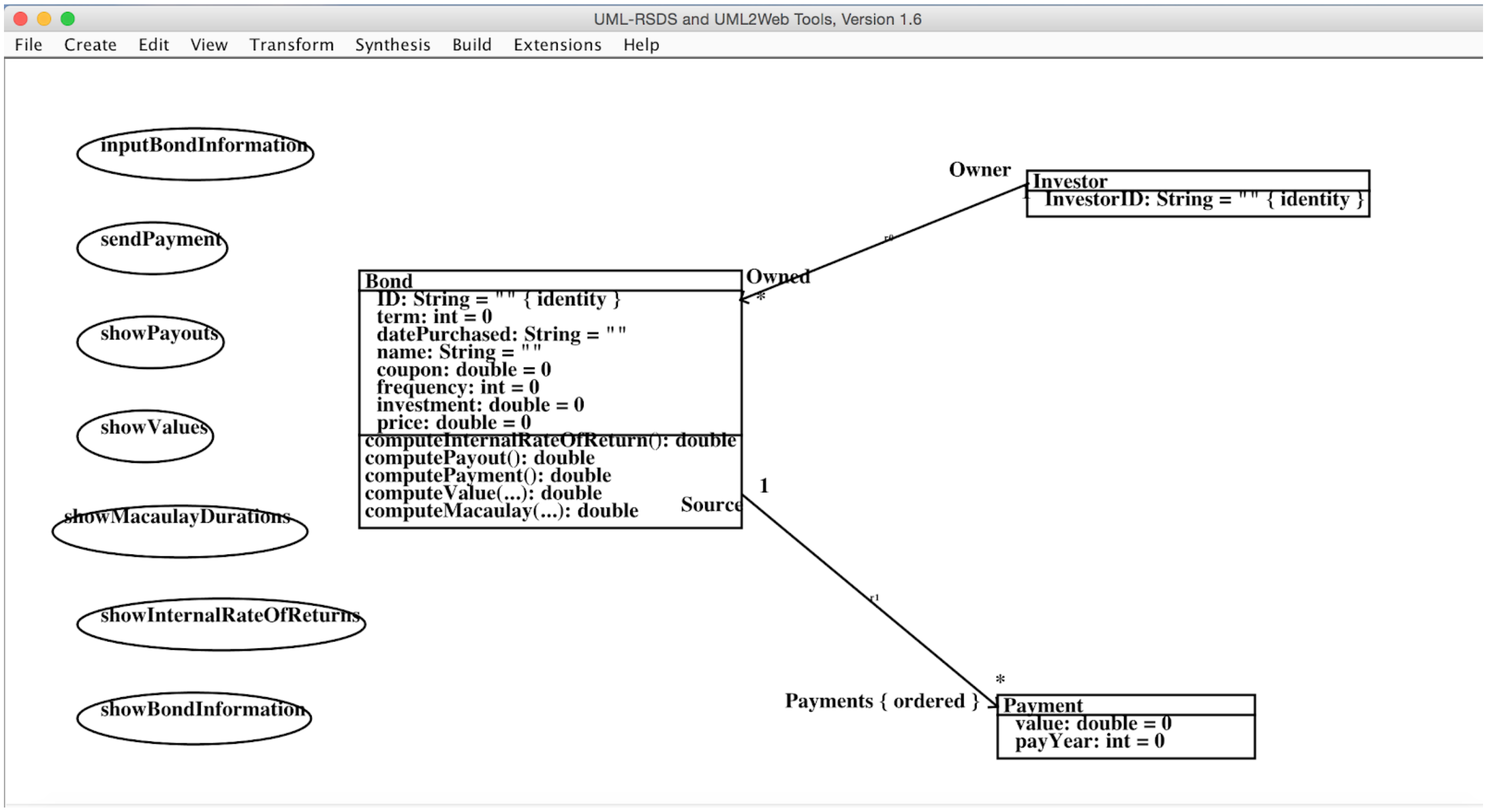
   We are given all of the necessary pieces, coupon, value, etc. but no operation for actually calculating individual payments. We have added this as an operation under the bond class.

Our Model Overview

In light of the above requirement specifications and clarifications, we have decided to approach our model in the following way:

Our software manages the relationship between a third party such as a a wealth manager or trader and the clients of the third party, investors who are interested in obtaining bonds. An investor will use this software to indicate what bond he/she is interested in buying. A representative from the third party will receive this information and, acting on behalf of the investor, buy the bond. Whenever, the third party representative receives a payment from the bond, he/she will relay that payment to the investor. The representative's role of buying a bond and relaying payments is accomplished outside of any help from the software. The function of the software is to store information about what bonds are owned by each client-investor and what payments have been relayed to each investor. Moreover, the software contains use cases accessible to representatives from the third party that display information about each bond in an investor's portfolio.

Class Diagram



Based upon the coursework requirements overview, that we further detailed in the preceding documents, we have generated the preceding class diagram. We decided to include a Bond class, a Payment class, and an Investor class. While the Bond class has a relation to Payment and one to Investor, Payment and Investor do not have a relationship to each other because it is unnecessary. Every investor can own multiple Bonds (or none) and every Bond has multiple Payments associated with it. The Payments associated with a given bond are ordered, as each corresponds with a sequential year of the Bond's term that it is paid in. This Payment year and the actual monetary value of the Payment are the only attributes in the Payment class. The Investor class has still fewer attributes and no operations. It's only attribute, InvestorID is a unique identifier so it has the identity trait. The Bond class has a number of attributes that were specified in the coursework as well as a unique identifier that we added, ID. The Bond class also includes a number of operations that will be operated on instances of the Bond class. Although it is not easily apparent from the class diagram, several of those operations do take inputs which we will see under the following section, class operations.

Class Operations

Query computePayout() : double
Pre-condition: true
Post-condition: result = coupon * term + investment

The computePayout() function will return the payout (sum of all payments) for a specific bond. To calculate this function, we have multiplied the coupon (which represents the payment that the investor receives every year since the frequency for this coursework is considered to be 1 payment per year) and we have added investment to the result (because the last coupon would give back the investment as well). Note: we add 100 every time as the investment, because in the coursework instructions it is specified that the investment is always 100.

****

Query computeValue(r: double): double
Pre-condition: true
Post-condition: result = (investment+coupon)/(1+r).pow(term)+ sequence{1…term-1} → iterate(n: Integer, sum: Real=0| (coupon)/(1+r).pow(n));

The computeValue() function will return bond values (sum of the discounted payments for that bond). To calculate each discounted payment, we have designed a function that loops for "term-1" times (the years in which the investor will receive payments for that specific bond).The function does this from year one until year term-1. Because the bond value at the term is calculated differently we have not included it in the loop (the last payment gives back the investment as well). To calculate each value, we have used the formula provided in the coursework description. Then in the final result we have added the discounted bond payment at the term added with the discounted payment values of the bond from year 1 until year term-1.

****

Query computeMacaulay(r: double): double
Pre-condition: true
Post-condition: result = ((Sequence{1...term}→ iterate(n: Integer, sum: Real=0| ((n*coupon)/(1+r).pow(n)))+(term*investment)/(1+r).pow(term))/Bond.computeValue(r)

The computeMacaulay() function will calculate the weighted number of years an investor must keep a bond until the present value of the bond's payments equals the amount paid for the bond. The function we have implemented loops for “term” times and calculates the value of each coupon throughout the lifespan of a bond in an investor’s portfolio, multiplied by the year in which we are calculating the value of the coupon. We then add the value of the initial investment that is included in the last payment * the term to the sum. And then we divide the sum by the bond value (we are calling the computeValue() function to find out the value of the bond) to solve for the Macaulay duration.

****

Query computePayment()
Pre-condition: true
Post-condition: if Payments->last().payYear == term - 1

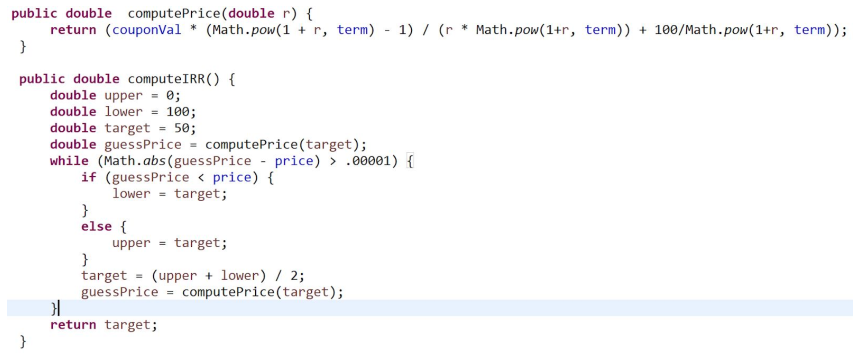
```
    then
        Payment→exists(p | p.value = coupon + investment, p.payYear =
        Payments.size() + 1) : Payments;
    else
    (
       Payment->exists(p | p.value = coupon, p.payYear = Payments.size() + 1) :
    Payments;
    )
```

The computePayment() function calculates the value of the payment in a specific year throughout the bond’s lifespan and then adds it to the Payments attribute in the Bond class which contains a list of all the payments that have been made so far. To generate the payment value, our function checks if the year of the bond in which we currently are calculating the payment is the term-1 year, which means we have to generate a Payment object which holds the last payment which is equivalent to coupon + investment . Otherwise, if we are not in the final year, the function creates a Payment object with the value equal to the coupons value that is then added to the Payments attribute in the Bond class.

****

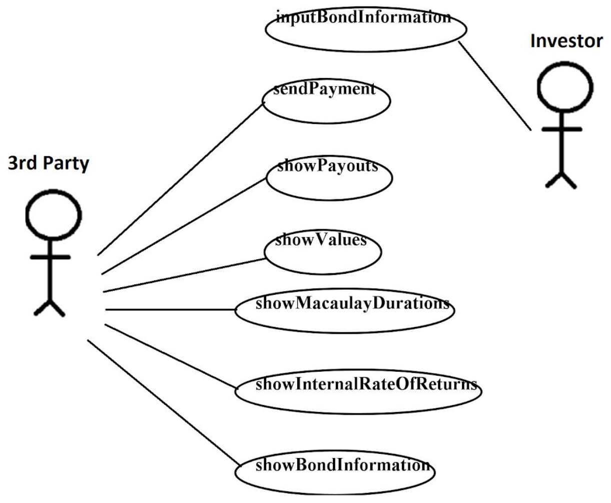
Query computeInternalRateOfReturn()
Pre-condition: price >= 0
Post-condition: price = ((Sequence{1..term} → iterate(n:integer, sum: double=0| coupon/((1+r).pow(n)))) + 100/(1+r).pow(term)) = true

The computeInternalRateofReturn() is used in capital budgeting to measure the profitability of potential investments.To fully understand this function and how it works, our team has done some research on the subject of internal rates of return. We have found out that the internal rate of return is the interest rate (r) at which the value of a bond is equal to the price that the investor has paid for that bond. So our function calculates r so that the value of a bond given the r from our function would be equal to the price of the bond. The post-condition checks to ensure this has been done correctly. Our implementation of this function finds r by taking in a an upper bound and lower bound value for r and converging the upper and lower bound until the correct value for r.



```java
public double  computePrice(double r) {
     return (couponVal * (Math.pow(1 + r, term) - 1) / (r * Math.pow(1+r, term)) + 100/Math.pow(1+r, term));
 }

 public double computeIRR() {
     double upper = 0;
     double lower = 100;
     double target = 50;
     double guessPrice = computePrice(target);
     while (Math.abs(guessPrice - price) > .00001) {
         if (guessPrice < price) {
             lower = target;
         }
         else {
             upper = target;
         }
         target = (upper + lower) / 2;
         guessPrice = computePrice(target);
     }
     return target;
 }
```

Use Case Diagram



Our software has two different types of users, representatives from a third party and the clients of the third party, the investors. The above diagram shows the operations that the two different types of users have access to. In the following page, we have provided pseudocode of these operations along with their written descriptions.

Use Case Operations

inputBondInformation(investorID: String, bondID: String, length: int, date: String, type: String, couponVal: double, frequencyVal: double, investmentVal: double, priceVal: double)

```
activity:
    if Bond->exists(b | b.ID = bondID)
    then
        “A bond with this identifier already exists” -> display() ;
    else if Investor->exists(i | i.ID = investorID)
    then
        “Invalid investor ID” -> display() ;

    else
    (
        Bond->exists(b | b.ID = bondID & b.term = length & b.datePurchased = date &
            b.name = type & b.coupon = couponVal & b.frequency = frequencyVal &
            b.investment = investmentVal & b.price = priceVal) :
            Investor[investorID].Owned;
    )
```

When an investor is interested in investing in a bond, he/she will use the inputBondInformation() use case to create a Bond object containing the essential details of the bond. The Bond object is then added to the Owned attribute in the Investor object corresponding to the investor (the Owned represents the investor’s portfolio). The 3rd party, acting on behalf of the investor, will then be responsible for buying the bond and relaying payments to the investor.

****

```
sendPayment(bondIdentifier: String, investorIdentifier: String)
activity:
    if ! Bond->exists(b | b.ID = bondIdentifier)
    then
        “A bond with this identifier does not exist”->display();
    else if ! Investor→exists(i | i.investorID = investorIdentifier)
    then
        “An investor with this identifier does not exist”->display()
    else if Bond[bondIdentifier] / : Investor[investorIdentifier].Owned
    then
        “This bond does not belong to this investor’s portfolio”->display();
    else if Bond[bondIdentifier].Payments→last().value == investment + coupon
        “The investor has received all payment from this bond. Removing
        bond from investor’s portfolio”->display();
        Investor[investorID].Owned /:  Bond[bondIdentifier]
    else
    (
         Bond[bondIdentifier].computePayment();
    )
```

Since the third party is the one actually in possession of the Bond, it must relay the payments it receives for the Bond to the investor. Once it relays a payment to the investor, the 3rd party updates the software to reflect this fact using the sendPayment() use case. This use case creates a Payment object containing the value of the payment and the year of the payment from the time the bond was issued by calling computePayment on the Bond in question. Within that operation the Payment object is then added to the Payments list in the Bond object to which the payment corresponds to. This use case is also responsible for removing a Bond object from an investor's portfolio. When all the payments belonging to a bond are listed in the Payments list of a Bond object, calling sendPayment() one more time will have the effect of removing the Bond from the list.

****

showPayouts(ID: String, r: double)
activity:

```
    if ! Investor->exists(i | i.investorID = ID)
    then
        "An investor with this identifier does not exist"-> display();
    else
    (
        for b : Investor[ID].Owned
        do
            ("Bond " + b.ID + b.computePayout(r))->display();
    )
```

The showPayouts() use case displays the payouts of all the bonds in the portfolio of an investor. For each Bond contained in the Owned list of an Investor object, the payout is calculated using the computingPayout() operation in the Bond class and displayed.

****

showValues(ID: String, r: double)
activity:

```
    if ! Investor->exists(i | i.investorID = ID)
    then
        "An investor with this identifier does not exist"-> display();
    else
    (
        for b : Investor[ID].Owned
        do
            ("Bond " + b.ID + b.computeValue(r))->display();
    )
```

The showValues() use case displays the discounted value of all the bonds in the portfolio of an investor. For each Bond contained in the Owned list of an Investor object, the discounted value is calculated using the computeValue() operation in the Bond class and displayed.

****

showMacaulayDurations(ID: String, r: double)
activity:

```
        if ! Investor->exists(i | i.investorID = ID)
        then
                “An investor with this identifier does not exist”-> display();
        else
        (
                for b : Investor[ID].Owned
                do
                        (“Bond “ + b.ID + b.computeMacaulay(r))->display();
        )
```

The showMacaulayDuration() use case shows the Macaulay duration of all the bonds in the portfolio of an investor. For each Bond contained in the Owned list of an Investor object, the Macaulay duration is calculated using the computeMacaulay() operation in the Bond class and displayed.

****

showInternalRateOfReturns(ID: String, r: double)

```
activity:
        if ! Investor->exists(i | i.investorID = ID)
        then
                “An investor with this identifier does not exist”-> display();
        else
        (
                for b : Investor[ID].Owned
                do
                        (“Bond “ + b.ID + b.computeInternalRateOfReturn()->display();
        )
```

The showInternalRateOfReturn() use case shows the internal rate of return of all the bonds in the portfolio of an investor. For each Bond contained in the Owned list of an Investor’s object, the internal rate of return is calculated using the computeInternalRateOfReturn() operation in the Bond class and displayed.

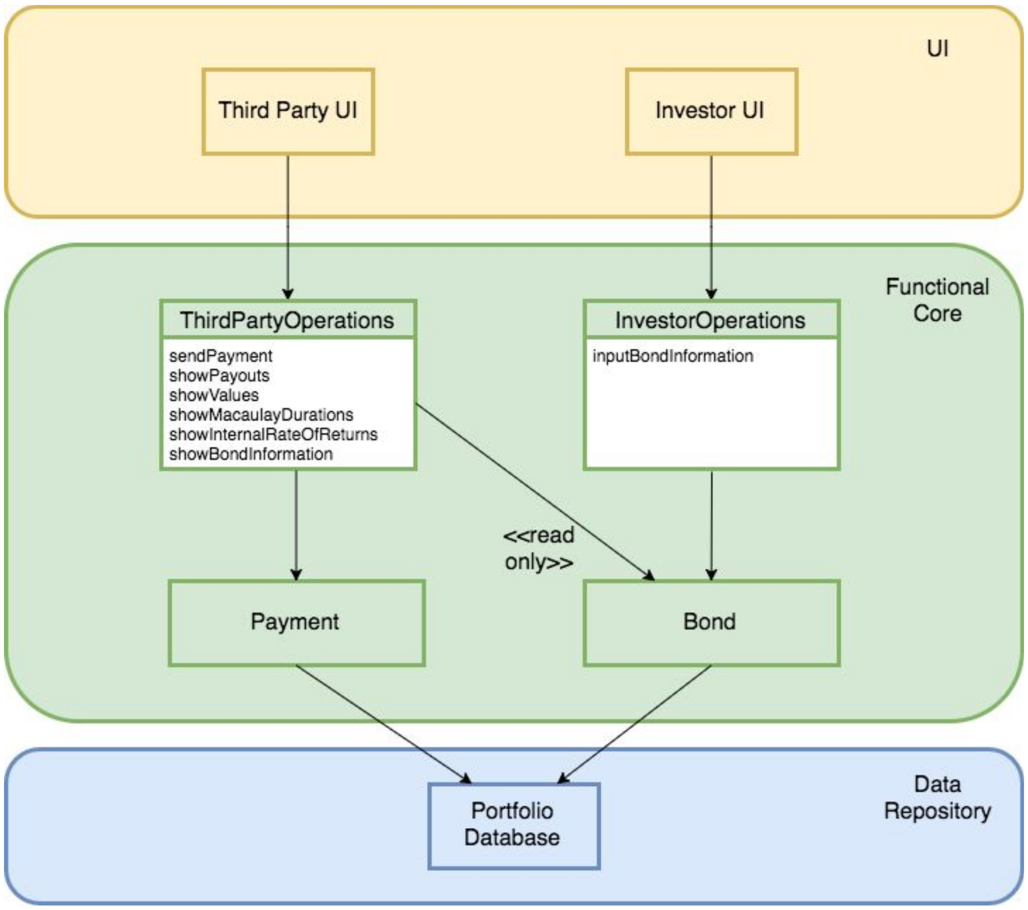
****

showBondInformation(bondIdentifier: String, r: double)

```
activity:
        if ! Bond->exists(b | b.ID =bondIdentifier)
        then
                “A bond with this identifier does not exist”→ display();
        else
        (
                b : Bond := Bond[bondIdentifier];
                (“Term: “ + b.term)->display();
                (“Date Purchased: “ + b.datePurchased)->display();
                (“Name: “ + b.name)->display();
                (“Coupon: “ + b.coupon)->display();
                (“Frequency: “ + b.frequency)->display();
                (“Investment: “ + b.investment)->display();
                (“Price: “ + b.price)->display();
                (“Payments sent so far: “ + b.Payments.size)->display
        )
```

The showBondInformation() use case simply displays the values of all the attributes belonging to an instance of the Bond object, along with the number of payments from that bond that have been sent to the investor.

Architecture Diagram



Future-proofing was an important factor in deciding which software architecture design to use. A 3-tier architecture diagram has allowed us to develop and maintain the user interface, functional core and data repository as independent modules. This gives the potential in the future to upgrade and replace any tier independently, as a change at any tier would not cause changes or unexpected behaviour in other tiers. The presentation tier occupies the top level of our architecture and displays the user interface. This tier communicates with the other tiers in the network by receiving, sending and displaying user data. Since there are two actors in the system, the user interface varies slightly for the third party and the investor and each actor can only view or access certain operations. Exclusivity gives the right view to the respective actor and promotes good software structure.

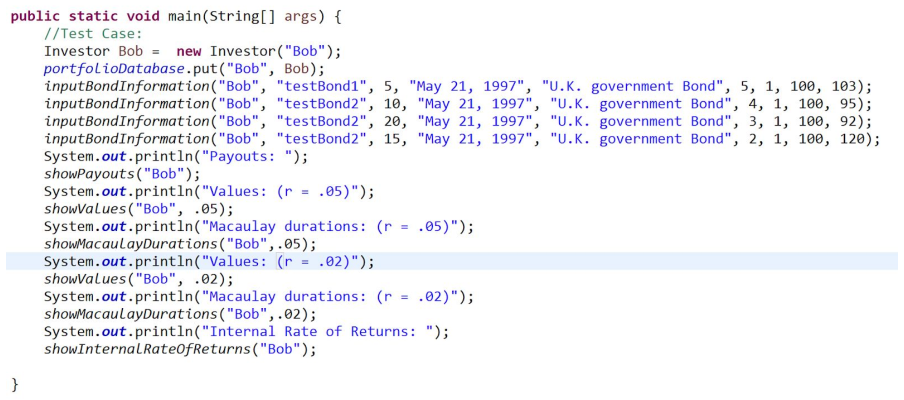
The functional core is the 2nd tier which holds the business and logic section of the software. Relying upon the data it pulls from the user interface, it controls the application functionally by performing operations and processes. We can see that each actor links with a respective set of operations, these operations handle the commands from the UI. The operations are grouped as a component for each actor as they functionally depend on the actor's input. The operations then invoke functionality on the bond and payment components, which are entities. Some or all functions in the operations component may depend on the bond and payment entities to perform their operations. This can be seen by multiple arrows. The third-party operations interact with the Bond entity on a read-only basis as the invoked operations do not directly write or change anything.

Finally, the database repository forms the last tier of the architecture and is responsible for housing the database server where information is stored and retrieved. The data is kept independent from the logic tier, which is highly important as it ensures data security. The database provides a single point of access to data and governs who is retrieving the data and how the data is updated. As the software deals with sensitive data it is important to establish a loosely coupled system. The entities from the previous tier allow the functional core and database to interact and communicate with each other. The Bond and Payment components may need to read and write data in the database so they depend on the data repository.

Ultimately all the tiers work together to fulfil the requirements of the systems through correct relations. Our 3-tier architecture provides a platform to keep up with the pace of change and new technologies. Not only has modularising our software helped us work efficiently it is ready to adapt to future needs and requirements. The design promotes increased scalability as the structure can run on different hardware and software. Due to the simplicity of our architecture diagram, reviewing requirements and concepts is easier compared to doing so in code.

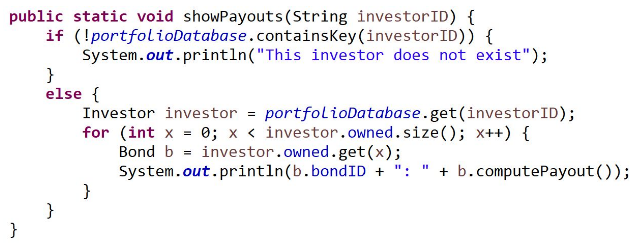
Testing

Our test case results are based on the following code (we have only changed the format of the spacing of the output):



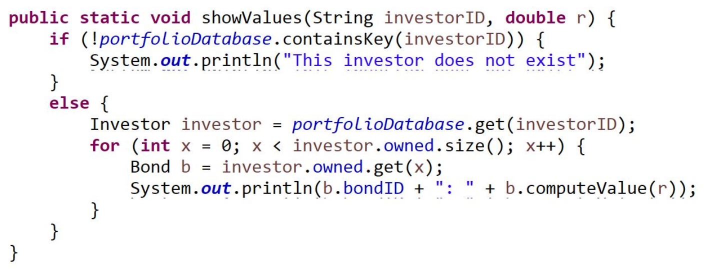
```
public static void main(String[] args) {
    //Test Case:
    Investor Bob =  new Investor("Bob");
    portfolioDatabase.put("Bob", Bob);
    inputBondInformation("Bob", "testBond1", 5, "May 21, 1997", "U.K. government Bond", 5, 1, 100, 103);
    inputBondInformation("Bob", "testBond2", 10, "May 21, 1997", "U.K. government Bond", 4, 1, 100, 95);
    inputBondInformation("Bob", "testBond2", 20, "May 21, 1997", "U.K. government Bond", 3, 1, 100, 92);
    inputBondInformation("Bob", "testBond2", 15, "May 21, 1997", "U.K. government Bond", 2, 1, 100, 120);
    System.out.println("Payouts: ");
    showPayouts("Bob");
    System.out.println("Values: (r = .05)");
    showValues("Bob", .05);
    System.out.println("Macaulay durations: (r = .05)");
    showMacaulayDurations("Bob",.05);
    System.out.println("Values: (r = .02)");
    showValues("Bob", .02);
    System.out.println("Macaulay durations: (r = .02)");
    showMacaulayDurations("Bob",.02);
    System.out.println("Internal Rate of Returns: ");
    showInternalRateOfReturns("Bob");

}
```

Test Case Results



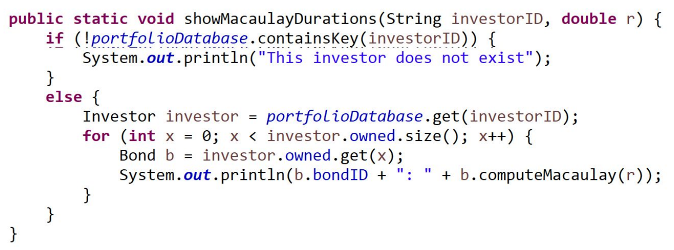
```
public static void showPayouts(String investorID) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computePayout());
        }
    }
}
```

Payouts: testBond1: 125.0, testBond2: 140.0, testBond3: 160.0, testBond4: 130.0



```
public static void showValues(String investorID, double r) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computeValue(r));
        }
    }
}
```

Values: (r = .05): testBond1: 99.99999999999997, testBond2: 92.27826507081517, testBond3: 75.07557931491999, testBond4: 68.86102588545819

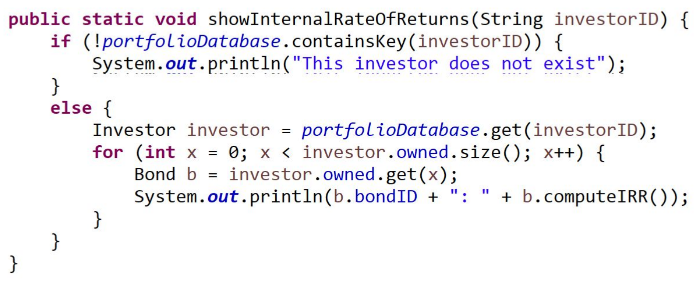


```
public static void showMacaulayDurations(String investorID, double r) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computeMacaulay(r));
        }
    }
}
```

Macaulay durations (r = .05): testBond1: 4.54595050416236, testBond2: 8.359589164010508, testBond3: 14.473825547456943, testBond4: 12.617602115465148

Values: (r = .02): testBond1: 114.1403785255126, testBond2: 117.96517001248446, testBond3: 116.35143334459708, testBond4: 99.99999999999997

Macaulay durations (r = .02): testBond1: 4.5788709836748644, testBond2: 8.579642968268926, testBond3: 15.717880872158787, testBond4: 13.106248770585527



```
public static void showInternalRateOfReturns(String investorID) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computeIRR());
        }
    }
}
```

Internal Rate of Returns: testBond1: 0.04320046864449978, testBond2: 0.04636130761355162, testBond3: 0.035662692971527576, testBond4: 0.006015982944518328

To see full implementation, including use cases and classes not shown here, refer to end of the document where the complete code implementation is available in an appendix.

Team Member Roles and Contributions

Junaid Mahmood (1680371): Architecture diagram and description. Implementing operations in Java.

Iris Radoi (1756682): class operations pseudo code and descriptions.

Mia King (1767801): requirements and initial specification, class diagram. Helped with class operations. Compilation, editing and submission of coursework deliverable.

Thomas Nemeh (176921): use case diagram, use case pseudocode, and use case descriptions. Helped with operations and implementation.

Philip Alexandrov (1771590): Implementation.

Sources

- www.Dmo.gov.uk
- www.investopedia.com
- https://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf
- https://www.investopedia.com/terms/i/irr.asp
- http://www.investinganswers.com/financial-dictionary/investing/internal-rate-return-irr-2130

Appendix

Full Java Implementation

Main class

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;

public class Main {
	public static HashMap<String, Investor> portfolioDatabase = new
HashMap<String,Investor>();
	public static HashMap<String, Bond> bondDatabase = new HashMap<String,Bond>();

  public static void main(String[] args) {
	//Test Case:
	Investor Bob =  new Investor("Bob");
	portfolioDatabase.put("Bob", Bob);
	inputBondInformation("Bob", "testBond1", 5, "May 21, 1997", "U.K. government Bond",
5, 1, 100, 103);
	inputBondInformation("Bob", "testBond2", 10, "May 21, 1997", "U.K. government Bond",
4, 1, 100, 95);
	inputBondInformation("Bob", "testBond2", 20, "May 21, 1997", "U.K. government Bond",
3, 1, 100, 92);
	inputBondInformation("Bob", "testBond2", 15, "May 21, 1997", "U.K. government Bond",
2, 1, 100, 120);
	System.out.println("Payouts: ");
	showPayouts("Bob");
	System.out.println("Values: (r = .05)");
	showValues("Bob", .05);
	System.out.println("Macaulay durations: (r = .05)");
	showMacaulayDurations("Bob",.05);
	System.out.println("Values: (r = .02)");
	showValues("Bob", .02);
	System.out.println("Macaulay durations: (r = .02)");
	showMacaulayDurations("Bob",.02);
	System.out.println("Internal Rate of Returns: ");
	showInternalRateOfReturns("Bob");

  }
```

```
    public static void inputBondInformation(String investorID, String bondID, String length, String
date, String type,
                                String couponVal, String frequencyVal, String investmentVal, String
priceVal) {
            if (!portfolioDatabase.containsKey(investorID)) {
                    System.out.println("This investor does not exist");
            }
            else {
                    Bond b = new Bond(bondID, date, Integer.parseInt(frequencyVal), type,
Double.parseDouble(couponVal), Double.parseDouble(investmentVal),
Double.parseDouble(priceVal), Integer.parseInt(length));
                    portfolioDatabase.get(investorID).owned.add(b);
                    bondDatabase.put(bondID, b);
            }
            }

    public static void showPayouts(String investorID) {
            if (!portfolioDatabase.containsKey(investorID)) {
                    System.out.println("This investor does not exist");
            }
            else {
                    Investor investor = portfolioDatabase.get(investorID);
                    for (int x = 0; x < investor.owned.size(); x++) {
                            Bond b = investor.owned.get(x);
                            System.out.println(b.bondID + ": " + b.computePayout());
                    }
            }
    }

    public static void showValues(String investorID, double r) {
            if (!portfolioDatabase.containsKey(investorID)) {
                    System.out.println("This investor does not exist");
            }
            else {
                    Investor investor = portfolioDatabase.get(investorID);
                    for (int x = 0; x < investor.owned.size(); x++) {
                            Bond b = investor.owned.get(x);
                            System.out.println(b.bondID + ": " + b.computeValue(r));
                    }
            }
```

```
}

public static void showMacaulayDurations(String investorID, double r) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computeMacaulay(r));
        }
    }
}

public static void showInternalRateOfReturns(String investorID) {
    if (!portfolioDatabase.containsKey(investorID)) {
        System.out.println("This investor does not exist");
    }
    else {
        Investor investor = portfolioDatabase.get(investorID);
        for (int x = 0; x < investor.owned.size(); x++) {
            Bond b = investor.owned.get(x);
            System.out.println(b.bondID + ": " + b.computeIRR());
        }
    }
}

public static void showBondInformation(String bondID) {
    if (!bondDatabase.containsKey(bondID)) {
        System.out.println("This bond does not exist");
    }
    else {
        Bond b = bondDatabase.get(bondID);
          System.out.println("Date: " + b.date);
          System.out.println("Frequency " + b.freq);
          System.out.println("Source: " + b.type);
          System.out.println("Term: " + b.term);
          System.out.println("Coupon: " + b.couponVal);
          System.out.println("Investment: " + b.investmentVal);
          System.out.println("Price: " + b.price);
          System.out.println("Payments sent: " + b.payments.size());
    }
    }

    public static void sendPayment(String bondID, String investorID) {
            if (!bondDatabase.containsKey(bondID)) {
                    System.out.println("This bond does not exist");
            }
            else if (!portfolioDatabase.containsKey(investorID)) {
                    System.out.println("This investor does not exist");
            }
            else {
                    Investor p = portfolioDatabase.get(investorID);
                    Bond b = bondDatabase.get(bondID);
                    if (!p.owned.contains(b)) {
                            System.out.println("This investor does not own this bond");
                    }
                    else {
                            if (b.payments.size() == b.term) {
                                    System.out.println("The investor has recieved all the payments for
this bond. This bond will be removed.");
                                    p.owned.remove(b);
                            }
                            else {
                                    b.computePayment();
                                    System.out.println("Payment sent");
                            }
                    }
            }

    }
}
```

Investor Class

```
import java.util.ArrayList;
public class Investor {
        private String investorID;
        public ArrayList<Bond> owned = new ArrayList<Bond>();

        public Investor(String ID) {
                investorID = ID;
        }

}
```

Bond Class

```
import java.text.DecimalFormat;
import java.util.ArrayList;

public class Bond {

   public String bondID;
   public String date;
   public int freq;
   public String type;
   public int term;
   public double couponVal, investmentVal, price;
   public ArrayList<Payment> payments;

   public Bond(String bondID, String date, int freq, String type, double couponVal, double
investmentVal, double price, int term){
       this.bondID = bondID;
       this.date = date;
       this.freq = freq;
       this.type = type;
       this.couponVal = couponVal;
       this.investmentVal = investmentVal;
       this.price = price;
       this.term = term;
       payments = new ArrayList<Payment>();
   }

   public double computePayout(){

           double sum = 0;
        for (int i = 0; i < term; i++){
           sum += couponVal;
        }
        sum += investmentVal;
        return sum;

   }

   public double computeValue(double r){

       double result = 0;
```

```
        for (int i = 1; i <= term-1; i++){
            result += couponVal/Math.pow(1+r, i);
        }
        result += (investmentVal + couponVal)/Math.pow(1+r, term);
        return result;

    }

    public double computeMacaulay(double r){

        double result = 0;

        for (int i = 1; i <= term; i++){
            result += i*couponVal/Math.pow(1+r, i);
        }

        result = (result+(term*investmentVal)/Math.pow(1+r, term))/computeValue(r);
        return result;
    }

    public double  computePrice(double r) {
                return (couponVal * (Math.pow(1 + r, term) - 1) / (r * Math.pow(1+r, term)) +
100/Math.pow(1+r, term));
        }

        public double computeIRR() {
                double upper = 0;
                double lower = 100;
                double target = 50;
                double guessPrice = computePrice(target);
                while (Math.abs(guessPrice - price) > .00001) {
                        if (guessPrice < price) {
                                lower = target;
                        }
                        else {
                                upper = target;
                        }
                        target = (upper + lower) / 2;
                        guessPrice = computePrice(target);
                }
                DecimalFormat numberFormat = new DecimalFormat("#.0000");
                return Double.parseDouble(numberFormat.format(target));
```

```
	}

	public void computePayment() {
		if (payments.size() == term - 1) {
			Payment p = new Payment(term, couponVal + investmentVal);
			payments.add(p);
		}
		else {
			Payment p = new Payment(term, couponVal);
			payments.add(p);
		}

	}
}
```

Payment Class

```
public class Payment {
	public double value;
	public int year;

	public Payment(int paymentYear, double amnt) {
		value = amnt;
		year = paymentYear;
	}
}
```