The Taxman cometh!

Simon Dawson
40983129

October 28, 2009

Contents

1 The Problem **2**

2 Designing the Solution **3**

3 Implementing the Solution **5**
3.1 Class Summary 5
3.1.1 Coin 5
3.1.2 Wad 5
3.1.3 Purse 6
3.1.4 Kitty 6
3.1.5 Person 7
3.1.6 Taxman 7
3.1.7 Player 7
3.1.8 Game 8

4 Testing the solution **10**
4.1 Coin 10
4.2 Wad 10
4.3 Game 11

5 Conclusion **14**

Chapter 1

The Problem

To produced an object-oriented computer application written in java that a person can use to play the game of *The taxman cometh*, hereafter described.

Taxman is a number game that is played against the computer. The human player chooses an upper limit, **N**, and the game is played with the integers from **1** to **N**. During the course of the game, the human and computer each accumulate a total. The object of the game is for the human to accumulate a larger total than the computer, hereafter referred to as the Taxman.

The player's total accumulates by selecting one of the numbers left in the game. The Taxman then gets all the numbers left in the game that are among the divisors of the player's chosen number. Once numbers are used (by either the player or the Taxman), they are removed from the game.

There is one major restriction on the numbers that the player may select. As in real life, the Taxman must always get something, so the player can never select a number unless at least one proper divisor remains in the game. Once no numbers with divisors remain (at the end of the game), the Taxman gets all the numbers left and the game is over.

The following requirements are to be met by the application:

- Player chooses some N which defines the set $S = \{1, 2, 3 \cdots N\}$
- Players chooses x such that $\exists x \in S \wedge \exists k \in S \wedge S\%k = 0$. x is added to players total, x is removed from S.
- $\forall k \in S$ such that $x\%k = 0$ the $\sum k$ is added to the taxmans total all such $k \in S$ are removed from the set
- Once the player can no longer choose some x matching the condition, $\forall k \in S \sum k$ is added to the taxmans total.
- The winner is then declared to be the player if his total is larger than that of the taxmans.

Chapter 2

Designing the Solution

Designing a solution was a matter of abstracting logical classes for the game. One of the very first things that you notice is that **coins** move around the

Figure 2.1: A game of the taxman in progress

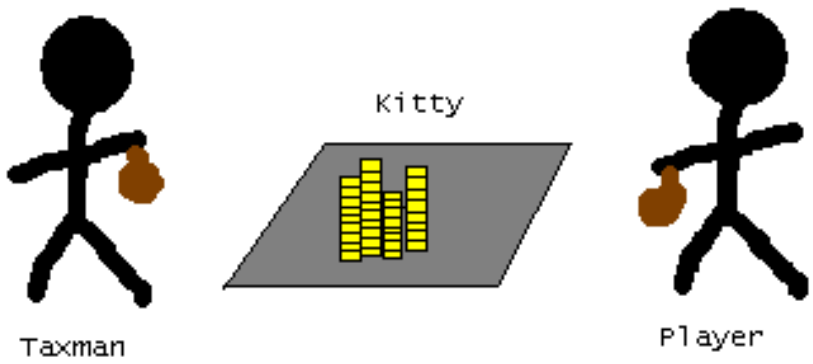
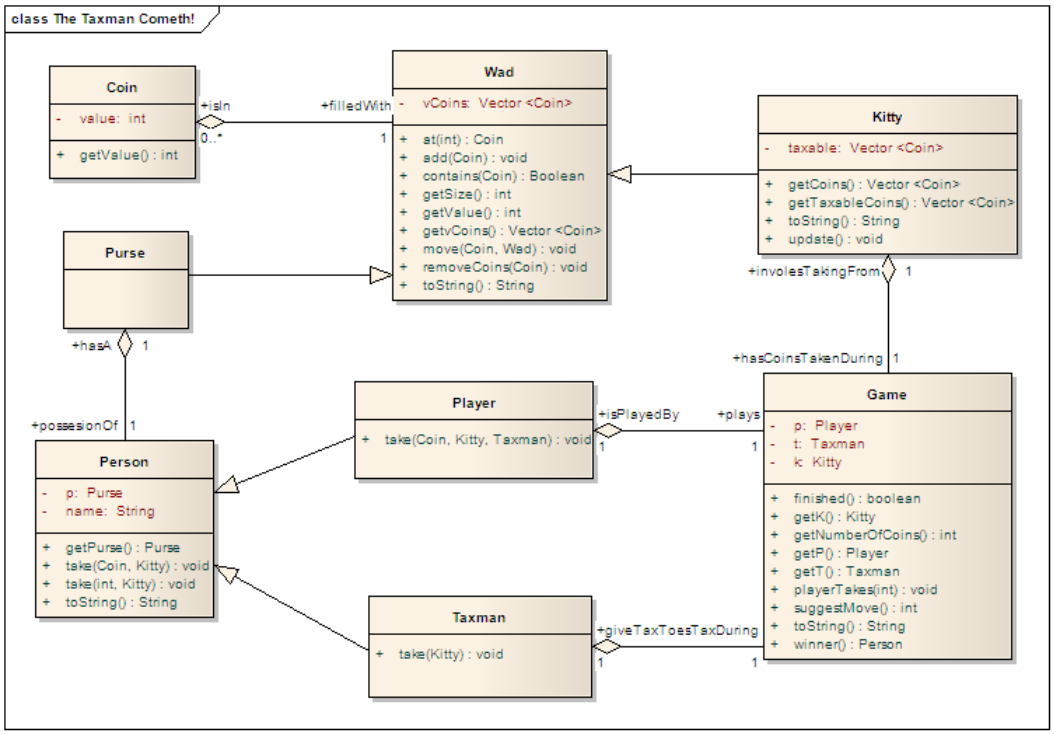


table. There are many collection of coins around the game. One in each the player and taxmans hand as well as in the centre of the table, the **kitty**. A collective noun for money is a **wad**. As in a **wad** of cash.

The other observation to make is that without the label **Taxman** and **Player** both look exactly the same. This infers they have the exact same attributes but may behave slightly differently. Both have a personal collection of coins one that others can't take out of, a **purse**.

Figure 2.2: Taxman Class Diagram



Chapter 3

Implementing the Solution

3.1 Class Summary

3.1.1 Coin

Most basic object in the game The Taxman, a coin.

Attributes

- `private int value;`

  Stores the value of the coin

Methods

- `public Coin(int _value);`

  Constuctor value of the coin is set to _value.

- `public int getValue();`

  Returns the value of this Coin

- `public boolean equals (Coin c);`

  This will return true if this coins value is the same as the coins which is passed in the parameter.

3.1.2 Wad

A wad is the collective noun for cash. This will define methods specific to a collection of coins.

Attributes

- `private Vector <Coin> vCoins;`

  A container to store all the coins in this wad

Methods

- `public Wad (int n) throws TaxmanError;`

  Constructor for a Wad filled with coins from value 1 to n inclusive. Throws an error on negative n.

- `public Wad (int n) throws TaxmanError;`

  Constructor for a Wad filled with coins from value 1 to n inclusive. Throws an error on negative n.

- `public void move (Coin _c, Wad _w);`

  Removes Coin _c from this Wad and puts it into Wad _w

- `public Coin at(int n) throws IndexOutOfBoundsException;`

  Return the coin at the n'th position in the vector. Throws error if n is not with the bounds of indexes of the vector.

- `public int getSize();`

  Returns the number of coins in this wad

3.1.3 Purse

A collection of coins is assembled into a purse object. Inherits from the Wad class.

3.1.4 Kitty

A collection of coins in the game of the taxman from which both the taxman and player take coins. Inherites from the Wad class.

Attributes

- `Vector <Coin> taxable;`

  Stores handles to the coins that still have divisors in the game.

Methods

- `public void update();`

  At any given point at execution this method will add all the Coin objects in this Kitty that are still taxable to the taxable vector.

3.1.5 Person

This defines some the behaviour inherent to both the taxman and the player.

Attributes

- `private Purse p;`

  Stores all of the persons Coin objects.

- `String name;`

  Used to print out on screen who this person is.

methods

- `public void take (Coin _c, Kitty _k);`

  Remove any Coin from the Kitty who's value is the same as Coin _c.

- `public void take (int _n, Kitty _k);`

  Removes first coin found that has value _n in _k and puts it into this purse.

3.1.6 Taxman

The taxman in the game. The person collecting all the tax.

Methods

- `public void take(Kitty _k);`

  Takes all the coins in Kitty _k and puts them into the Purse of this taxman

3.1.7 Player

The player of the game of the taxman.

Methods

- `public void take(Coin _c, Kitty _k, Taxman _t) throws TaxmanError;`

  Takes _c checks if that coin is part of the taxable coins in the Kitty _k. If it is taxable. Puts Coin _c in this Players Purse. Takes all coins in the kitty that were divisors of Coin _c and puts them into the Taxman _t's purse. Calls for the Kitty to update. If that was the last move of the game takes all the coins in the kitty and puts them into the taxmans purse.

3.1.8 Game

Game has a Player, Taxman and a Kitty. This class defines how these three interact.

Attributes

- `private Player p;`

  The player in this game.

- `private Taxman t;`

  The taxman in this game.

- `private Kitty k;`

  The kitty object in this game.

- `private int numberOfCoins;`

  Stores how many coins will be avalible to choose from at the first turn.

Methods

- `public boolean finished();`

  Return true if and only if there are no more taxable coins in the kitty.

- `public void playerTakes(int n) throws TaxmanError;`

  If this games kitty contains a Coin with value n that Coin is removed from this games Kitty and put in the Purse of the Player. All the divisors of that Coin left in the game are put into this games taxmans purse.

- `public int suggestMove() throws TaxmanError;`

  A greedy algorithm to decide the best move available in the game. Checks for the largest difference between the value the taxman will get and the value the player will get. Suggests that best one for the player. Throws an error should the game be in a finished state.

- `public Person winner();`

  Returns null should the game not be over but if it is. It returns the Player if the player has a larger total value of coins than the taxman. If the player has the same value or lower than the taxman it returns the taxman.

Chapter 4

Testing the solution

4.1 Coin

- ```
  public void testConstructorAndGetValue () {
      for (int i=-10;i<10;i++) {
          Coin c = new Coin(i);
          Assert.assertTrue(c.getValue()==i);
      }//end for
  }//end testConstructorAndGetValue
  ```

  Creates coins with values from -10 to 10 and ensures the value is correct.

4.2 Wad

- ```
  public void testDefaultConstructor () {
      Wad w = new Wad();
      Assert.assertTrue(w.getSize()==0);
      Assert.assertTrue(w.getValue()==0);
      try { w.at(0); Assert.fail(); }
      catch (Exception e) {}
  }//end testDefailtConstructor
  ```

  Create a empty wad. Ensure that it is empty. Checks if an exception is thrown if the first element is attempted to be accessed.

- ```
  public void testConstructorWithParam () {
      Wad w;
      int sum;
      for (int i=0;i<100;i++) {
          try {
              w= new Wad(i);
              Assert.assertTrue(w.getSize()==i);
  ```
  ```

```
            sum=0;
            for (int x=1;x<=i;x++){sum+=x;}
            Assert.assertTrue(w.getValue()==sum);
        } catch (TaxmanError e) {e.printStackTrace();}
    }//end for
}//end testConstructorWithParam
```

Test to see that the number of coins in a wad matches that of the constructor and that $\sum_{k=1}^{i} k$ is the same as the Coin.getValue().

- ```
  public void ConstructorWithInvalidParam () {
      try {Wad w= new Wad(-1); Assert.fail(); }
      catch (Exception e) {}
  }//end ConstructorWithInvalidParam
  ```

  Check to see that there is an error thrown when a negative is given.

4.3 Game

- ```
  public void testAI () {
      float countWins=0;
      for (int i=3;i<100;i++) {
          if (testAI(i)){
              countWins=countWins+1;
          }//end if
      }//end for
      float winPercentage=countWins/97;
      Assert.assertTrue(winPercentage>0.90);
  }//end testAI

  public boolean testAI (int i) {
      Game g=null;
      try {
          g = new Game(i);
          while (!g.finished()) {
              int j = g.suggestMove();
              g.playerTakes(j);
          }//end while
      } catch (TaxmanError e) {e.printStackTrace();}//end try/catch
      return (g.winner().getName()=="Player");
  }//end GameTest
  ```

  These two tests combined get the computer the play the game of taxman by itself. If the computer is able to beat the taxman countWins is incremented. By the end 97 games of the Taxman have been played.

The percentage of games that have been won by the computers AI functionality alone is calculated. The test passes if and only if the computer can in 90% of the games with coins raning from 3 to 99 coins.

- ```
  public void testGameConstruction () {
      Random r = new Random();
      int i = Math.abs(r.nextInt()) % 300;
      i+=3;
      Game g = null;
      try {
          g=new Game(i);
          //All coins are able to be chosen bar 1
          Assert.assertTrue(g.getK().getTaxableCoins().size()==i-1);
          //Game is not finished
          Assert.assertFalse(g.finished());
          //Game has no winner
          Assert.assertNull(g.winner());
      } catch (TaxmanError e) {e.printStackTrace();}
  }//end testGameConstruction
  ```

  Creates a random number N such that $(N > 4) \wedge (N < 302)$ and uses that to create a game with N coins. Ensures that there are $N - 1$ chooses for the player the first game. Seeing as 1 has no divisors. Asserts a game with chooses is not finished. Asserts while the game is not over yet there is no winner yet.

- ```
  public void testOutOfBoundMove () {
      Random r = new Random();
      int i = Math.abs(r.nextInt()) % 300;
      i+=3;
      Game g = null;
      try {
          g=new Game(i);
          g.playerTakes(i+1);
          g.playerTakes(-1);
          Assert.fail();
      } catch (TaxmanError e) {e.printStackTrace();}
  }//end testMove
  ```

  Creates a game of a random size. Player attempts to take a coins that do not exist. -1 and i+1. The test will fail if no exception is thrown.

- ```
  public void testNoDivsorMove () {
      Random r = new Random();
      int i = Math.abs(r.nextInt()) % 300;
  ```
  ```

```
    i+=3;
    Game g = null;
    try {
        g=new Game(i);
        g.playerTakes(1);
        Assert.fail();
    } catch (TaxmanError e) {e.printStackTrace();}
}//end testMove
```

Creates a game of a random size. Player attempts to take a coin that has no divisors in the game. Will fail if exception is not thrown.

- ```
  public void testLegalMove () {
      Random r = new Random();
      int i = Math.abs(r.nextInt()) % 300;
      i+=3;
      Game g = null;
      try {
          g=new Game(i);
          g.playerTakes(2);
          Assert.assertTrue(g.getK().getvCoins().size()==i-2);
          Assert.assertTrue(g.getP().getPurse().getValue()==2);
          Assert.assertTrue(g.getT().getPurse().getValue()==1);
      } catch (TaxmanError e) {e.printStackTrace();}
  }//end testMove
  ```

  Creates a game of a random size. Player takes coin 2. Taxman will therefore get coin 1. Test will fail if coin was not removed from games kitty. Test will fail if coin is not added to players purse. Test will fail if coin is not added to taxmans purse.
  ```

Chapter 5

Conclusion

Throughout the problem definition, design and implementation the need to ensure object simplicity was keep paramount. Classes were written to behave as close to their real world counterparts. With the addition of Javas class inheritance technology the solution moved swiftly from UML design to code. The classes relating the logic and function of the game mechanics were kept from the user via visage classes that dealt with the interactions between objects and users. What the product of all of these became was an object-oriented application that met the requirements.