The Taxman Cometh

COMP225 S1 2016 – Assignment 1

Joseph Lewis
SID: 43656242

29/04/2016

Table of Contents

1. Introduction **2**

2. Program Code **2**

Coin **2**

Collection **3**

Game **6**

Human **7**

HumanTest **12**

Player **14**

Table **15**

Taxman **17**

TaxmanError **18**

TaxmanOperator **19**

3. Results **22**

3.1 During the Game **23**

3.2 Stage I **23**

3.3 Stage II **25**

3.3.1 Naive 25

3.3.2 Greedy and Optimal 26

4. Discussion **27**

1. Introduction

This report is a supplementary document for COMP225 assignment, The Taxman Cometh. The objective of the report is to provide the program code with brief descriptions of each class, presented in section 2, list outputs from each stage of the assignment and provide a discussion for the design and approaches taken in implementing the solution to the problem.

The goal of the programming aspect of the assignment was to create a project in Java to replicate the 'The Taxman'; a game where given an upper limit, N, the round is played with integers 1 to N. The player chooses a number that has at least one factor still in play and the Taxman gets the numbers that are divisors for the player's choice. If there are no valid choices left, the Taxman then gets the rest of the coins in the collection. It is difficult to determine if there is any simple method of determining the optimal solution given the rules of the game and this report will explore different strategies to optimise the results of the game for the player.

Stage I was to implement basic functionality in the project and allow the user to come up with the strategy for each round of the game by accepting input displayed on the console in a turn-based configuration. Some restrictions were also enforced such as the requirement for a Player class to be present, only one Main (main()) method can be called and the primary data structure must be a 'set', e.g. the java.util.LinkedHashSet used for creating the Collection of coins.

In Stage 2, the requirement was to devise a greedy strategy called as a method with a coin selecting algorithm for each move. A naive strategy was first implemented, choosing the largest remaining number with divisors each turn; this was proved ineffective. The greedy strategy was then implemented to be the successor strategy. As it will be shown in later sections, the greedy strategy could not be proven to choose the optimal coins for each game. The requirement to create a brute-forcing algorithm was then tasked to students in Stage III. Recursive backtracking was the recommended primary method for developing this algorithm.

The results of each Stage will be presented in the section 3 and the strategies discussed in section 4.

2. Program Code

The purpose of this section is to provide the source code used in the program and give short descriptions of each class. Each class and method in the project contains Javadoc and comments that will explain most of the design and functionality for each part.

Coin

This class is used to create coin objects, with instances created using the constructor with a given value as a parameter.

```
package taxman;

/**
```

```
 * A simple class that is used to create Coin objects.
 *
 * @author Joseph Lewis
 *
 */
public class Coin {
	// Declare the value variable for a Coin instance
	private int value;

	/**
	 * The Coin constructor, used to create an instance and assign a value to a
	 * coin.
	 *
	 * @param val
	 *            is the value the coin has
	 */
	public Coin(int val) {
		value = val;
	}

	/**
	 * A getter method for the value of the chosen coin
	 *
	 * @return the value of the coin
	 */
	public int getValue() {
		return value;
	}

	/**
	 * Used to compare the value of the current coin to a specified coin.
	 *
	 * @param c
	 *            is the given coin to compare
	 * @return a boolean value to determine if the coin values are equal
	 */
	public boolean equals(Coin c) {
		return this.getValue() == c.getValue();
	}
}
```

Collection

A Collection is a set of Coin objects used in the game. An instance of a collection is used for the game. The Taxman and player both have their own collections that increase in size each turn of the game. This class includes methods to add, remove or remove coins, along with certain getter methods.

```
package taxman;

import java.util.Iterator;
import java.util.LinkedHashSet;

/**
 * The Collection class is used to store a set of Coin instances to be used for
 * the game. The human and taxman both have collections as well as the table
 * with the coins still available.
 *
 * @author Joseph Lewis
 *
 */
public class Collection {
	// Declare the variables to be used throughout the class'
```

```
// LinkedHashSet is the primary data structure
private LinkedHashSet<Coin> collection = new LinkedHashSet<Coin>();
// An iterator is declared for moving through the LinkedHashSets
private Iterator<Coin> it;

/**
 * The Collection constructor used to create an instance for the players.
 */
public Collection() {

}

/**
 * The Collection constructor used for the table and fills a collection with
 * all specified <code>Coin</code> instances for the specified upper limit.
 *
 * @param lim
 *            is the upper limit.
 */
public Collection(int lim) {
	Coin c;
	for (int i = 0; i < lim; i++) {
		c = new Coin(i + 1);
		collection.add(c);
	}
}

/**
 * Used to add a coin to the collection.
 *
 * @param c
 *            is the specified coin
 */
public void add(Coin c) {
	collection.add(c);
}

/**
 * Used to remove a given coin from the collection if it exists.
 *
 * @param c
 *            is the specified coin
 */
public void remove(Coin c) {
	// Iterator is used to scan through the set
	it = collection.iterator();
	while (it.hasNext()) {
		Coin tC = it.next();
		// If the coin is found, remove it
		if (tC.equals(c)) {
			it.remove();
		}
	}
}

/**
 * A selected coin is moved from the current Collection instance to another
 * specified Collection.
 *
 * @param col
 *            is the collection for the coin to be moved to
 * @param c
 *            the given coin
 */
public void move(Collection col, Coin c) {
```

```
        if (collection.contains(c)) {
                collection.remove(c);
                col.add(c);
        }
}

/**
 * Returns the Coin with the value parsed through the method
 *
 * @param val
 *            is the value of the desired Coin
 * @return the Coin
 */
public Coin getCoin(int val) {
        it = collection.iterator();
        while (it.hasNext()) {
                Coin c = it.next();
                // When the Coin is found, return it
                if (c.getValue() == val) {
                        return c;
                }
        }
        // Return null if the Coin is not found
        return null;
}

/**
 * A getter method that returns the collection when called.
 *
 * @return
 */
public LinkedHashSet<Coin> getCollection() {
        return collection;
}

/**
 * Returns the number of Coins in the collection
 *
 * @return
 */
public int getNoCoins() {
        return collection.size();
}

/**
 * Gets the combined value of all the coins in the collection.
 *
 * @return val, the total value of the collection
 */
public int getValue() {
        int val = 0;
        it = collection.iterator();
        // While there are Coins in the collections, add the value to the value
        // variable
        while (it.hasNext()) {
                Coin tC = it.next();
                val += tC.getValue();
        }

        return val;
}

/**
 * Gets the number of coins in the collection.
 *
```

```
         * @return the number of coins
         */
        public int getSize() {
                return collection.size();
        }

        /**
         * Verifies that a Collection contains a given Coin.
         *
         * @param c
         *             the coin to be tested for
         * @return the boolean answer to the coin being found
         */
        public boolean contains(Coin c) {
                return collection.contains(c);
        }

}
```

Game

A single Game instance is created each round from the TaxmanOperator Java Class. An instance of the table, human and Taxman are declared and instantiated for the game. This method is also used to determine when the game is over and getter methods are utilised to return the instance of each desired object.

```
package taxman;

/**
 * Used to create an instance of 'The Taxman' game. Instances of the table and
 * players are created in this class.
 *
 * @author Joseph Lewis
 *
 */
public class Game {
        // Declare variables
        private static Table table;
        private static Human human;
        private static Taxman taxman;
        private int coins;

        /**
         * The constructor that creates an instance of the table, human and taxman.
         *
         * @param uLim
         *             the upper limit of the game
         * @throws TaxmanError
         *              if something does not go to plan
         */
        public Game(int uLim) throws TaxmanError {
                table = new Table(uLim);
                human = new Human();
                taxman = new Taxman();
                coins = uLim;
        }

        /**
         * A getter method for the game <code>Table</code>.
         *
         * @return the table
         */
        public Table getTable() {
```

```
                return table;
        }

        /**
         * A getter method for the game <code>Human</code>.
         *
         * @return the human
         */
        public Human getHuman() {
                return human;
        }

        /**
         * A getter method for the game <code>Taxman</code>.
         *
         * @return the taxman
         */
        public Taxman getTaxman() {
                return taxman;
        }

        /**
         * Gets the number of coins left in the game.
         *
         * @return the number of coins
         */
        public int getNoCoins() {
                return coins;
        }

        /**
         * Determines if the game is over depending on the amount of taxable coins
         * in play.
         *
         * @return a boolean to describe if the game is over
         */
        public boolean gameOver() {
                return table.getTaxableCoins().isEmpty();
        }
}
```

Human

The Human class is a subclass of Player and includes extra functionality only needed by the player. The Stage I strategy that takes user input used methods from this class and the naive, greedy and optimal methods are implemented in this class to be called from the operator class during the game.

```
package taxman;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedHashSet;

/**
 * The human player in The Taxman game. Human extends the player class and
 * instances inherit their own behaviour. The class includes algorithms to
 * automate naive, greedy and optimal moves for the game when opted.
 *
 * @author Joseph Lewis
 *
 */
public class Human extends Player {
```

```
        // Declare the iterator
        private Iterator<Coin> it;

        private int bestTotal;
        ArrayList<Integer> bestMoves = new ArrayList<Integer>();
        ArrayList<Integer> moves = new ArrayList<Integer>();
        int collectionSize = 0; // Used for original recursion
        int[] path; // OR USE LINKED LIST
        int currentPathLevel = 0;
        int prevScore = 0;

        /**
         * The constructor for the Human class
         */
        public Human() {
        }

        /**
         * The human player used this method to choose a coin to take from the table
         * during the instance of the game.
         *
         * @param game
         *             is the game being played
         * @param val
         *             is the value of the coin to be taken
         * @throws TaxmanError
         *              if the coin chosen does not exist
         */
        public void chooseCoin(Game game, int val) throws TaxmanError {
                boolean coinFound = false;
                it = game.getTable().getTaxableCoins().iterator();
                Coin currentCoin;
                // Iterate through the table's valid coins until the value of the given
                // coin is found
                while (it.hasNext()) {
                        currentCoin = it.next();
                        if (currentCoin.getValue() == val) {
                                // If the coin is found, take it from the game table
                                game.getHuman().take(game.getTable(),
game.getTable().getCoin(val), game.getTaxman());
                                coinFound = true;
                        }
                }
                // If the coin is not present, throw a custom TaxmanError
                if (!coinFound) {
                        TaxmanError te = new TaxmanError("The coin to be selected was not
found.");
                        throw te;
                }
        }

        /**
         * If the chosen coin is part of the taxable Collection, take it from the
         * table. The Taxman then gets the rest of the divisors for that coins
         * value. If there are no other taxable coins in the game, the Taxman gets
         * the rest of the table.
         *
         * @param table
         *             contains the collection of coins in play
         * @param coin
         *             is the chosen coin by the <code>Human</code>
         * @param taxman
         *             is the <code>Taxman</code> in the game
         * @throws TaxmanError
         *              if there are no divisors for the specified coin
```

```
	 */
	public void take(Table table, Coin coin, Taxman taxman) throws TaxmanError {
		if (table.getTaxableCoins().contains(coin)) {
			// The Human takes the coin from the table
			super.take(table, coin);
			for (int i = 1; i < coin.getValue(); i++) {
				// The Taxman takes all divisors for the chosen coin
				if (coin.getValue() % i == 0) {
					taxman.take(table, i);
				}
			}
			table.calTaxable();
			if (table.getTaxableCoins().isEmpty()) {
				// The taxman takes the rest if there are no more choices
				taxman.takeTable(table);
				// Clear the table - not necessary as the game will end
				// table.getCollection().clear();
			}
			// Throw a TaxmanError if the move is invalid due to there being no
			// divisors for the chosen number
		} else {
			TaxmanError te = new TaxmanError("The Taxman must always get
something.");
			throw te;
		}
	}

	/**
	 * Returns the largest taxable coin.
	 *
	 * @param inPlay
	 *            are the coins on the table
	 * @return the last element of the parsed Set
	 * @throws TaxmanError
	 *             if a game logic error occurs
	 */
	public int naiveMove(LinkedHashSet<Coin> inPlay) throws TaxmanError {
		LinkedHashSet<Coin> taxableCoins = Table.getTaxable(inPlay);
		it = taxableCoins.iterator();
		Coin tC = new Coin(0);
		while (it.hasNext()) {
			// This works as the LinkedHashSet is created in order
			tC = it.next();
		}
		return tC.getValue();
	}

	/**
	 * For loops are used to go through each choice of coin to decide what move
	 * leads to the Taxman gaining the most value gap in that turn and returns
	 * the value of that coin.
	 *
	 * @param inPlay
	 *            are the coins on the table
	 * @param taxableCoins
	 * @return the greedy move
	 * @throws TaxmanError
	 */
	public int greedyMove(LinkedHashSet<Coin> inPlay) throws TaxmanError {
		Coin greedyChoice = null;
		Integer bestMoveSoFar = null;
		LinkedHashSet<Coin> taxableCoins = Table.getTaxable(inPlay);
		it = taxableCoins.iterator();
		// Iterates through each of the 'taxable' coins
		while (it.hasNext()) {
```

```
                        Coin tC = it.next();
                        int taxmanTotal = 0;
                        Iterator<Coin> it2 = inPlay.iterator();
                        // Iterates through the coins still in play
                        while (it2.hasNext()) {
                                Coin tC2 = it2.next();
                                if (tC.getValue() > tC2.getValue() && tC.getValue() %
tC2.getValue() == 0) {
                                        taxmanTotal += tC2.getValue();
                                }
                        }
                        // If this move results in the largest gap in points between the
                        // plays, set that as the best move
                        if (bestMoveSoFar == null || tC.getValue() - taxmanTotal >
bestMoveSoFar) {
                                greedyChoice = tC;
                                bestMoveSoFar = tC.getValue() - taxmanTotal;
                        }
                }
                return greedyChoice.getValue();
        }

        /**
         * Used to return a <code>Coin</coin> LinkedHashSet created from an upper
         * limit as a parameter for the method.
         *
         * @param lim
         *            is the upper limit provided
         * @return the created collection of coins
         */
        public LinkedHashSet<Coin> createCollection(int size) {
                LinkedHashSet<Coin> set = new LinkedHashSet<Coin>();
                for (int i = 1; i <= size; i++) {
                        set.add(new Coin(i));
                }
                return set;
        }

        /**
         * Called before using @see {@link Human#optimalMove(LinkedHashSet)} to
         * calculate the optimal moves for the game using recursive backtracking.
         * Used to set the class variables to their correct starting values.
         *
         * @param setSize
         *            to set the size of the path array; used to keep track of the
         *            position for backtracking
         */
        public void preOptimal(int setSize) {
                // Create the integer array for keeping track of the recursion
                path = new int[setSize];
                for (int i = 0; i < path.length; i++) {
                        path[i] = 1;
                }
                prevScore = 0;
                moves.clear();
                bestTotal = 0;
                collectionSize = setSize;
        }

        /**
         * Helper method for backtracking in @see
         * {@link Human#optimalMove(LinkedHashSet)}. This method returns a
         * <code>LinkedHashSet</code> containing the set of coins available for the
         * next path of calculations on the game tree. The current score obtained
         * and the current level of recursion is kept track of here.
```

```
         *
         * @return the next collection to complete the recursion on
         */
        public LinkedHashSet<Coin> nextPath() {
                LinkedHashSet<Coin> full = createCollection(collectionSize);
                LinkedHashSet<Coin> tmp = Table.getTaxable(full);
                boolean somethingLeft = true;
                boolean found = false;
                int sum = 0;
                for (int i = 0; i < path.length; i++) {
                        it = tmp.iterator();
                        if (path[i] != 1) {
                                int tC = 0;
                                if (it.hasNext()) {
                                        while (somethingLeft && tC < path[i]) {
                                                tC = it.next().getValue();
                                                // If at the end and no success
                                                if (!it.hasNext() && tC < path[i]) {
                                                        somethingLeft = false;
                                                        found = false;
                                                } else {
                                                        found = true;
                                                }
                                        }
                                } else {
                                        somethingLeft = false;
                                }
                                // If no more choices in higher level, increment lower
                                if (!found) {
                                        if (i != 0) {
                                                path[i - 1]++;
                                                somethingLeft = true;
                                                // i -= 2;
                                                // Return to the start of the loop to rebuild
collection
                                                tmp = Table.getTaxable(full);
                                                i = -1;
                                        } else {
                                                return null;
                                        }
                                } else { // Otherwise remove found coin for next level
                                        // path[i] = tC;
                                        if (path[i + 1] != 1) {
                                                sum += tC;
                                                full.remove(tC);
                                                // -- Remove coins from table after choice
                                                Iterator<Coin> it2 = full.iterator();
                                                while (it2.hasNext()) {
                                                        Coin c2 = it2.next();
                                                        if (tC % c2.getValue() == 0) {
                                                                full.remove(c2);
                                                                it2 = full.iterator();
                                                        }
                                                } // -- End removal
                                                tmp = Table.getTaxable(full);
                                                found = false;
                                        } else {
                                                currentPathLevel = i;
                                                prevScore = sum;
                                                return tmp;
                                        }
                                }
                        }
                }
                prevScore = sum;
```

```
                return tmp;
        }

        /**
         * Recursive backtracking is used along with some helper methods to
         * calculate all possible combinations of the game tree generated from the
         * given collection. The move set leading to the highest total is kept track
         * of and returned at the end of the recursion.
         *
         * @param inPlay
         *            the collection of coins to find the best moves for
         * @return the best move set
         * @throws TaxmanError
         *             if a game logic related error occurs
         */
        public ArrayList<Integer> optimalMove(LinkedHashSet<Coin> inPlay) throws TaxmanError {
                // Get the valid moves of the given coins
                LinkedHashSet<Coin> choices = nextPath();
                int score = 0;
                // Test end condition
                if (choices == null) {
                        return bestMoves;
                }
                // Take the next coin
                it = choices.iterator();
                int curr = it.next().getValue();
                moves.add(curr);
                score += curr;
                inPlay.remove(curr);
                path[currentPathLevel] = curr;
                // Update best moves if the conditions are met
                if (score > bestTotal) {
                        bestMoves.clear();
                        bestTotal = score;
                        for (int i = 0; i < moves.size(); i++) {
                                bestMoves.add(moves.get(i));
                        }
                }
                // If there are still nodes to be follows, continue recursion
                choices = nextPath();
                if (choices != null) {
                        currentPathLevel++;
                        optimalMove(choices);
                }
                return bestMoves;
        }

}
```

HumanTest

A JUnit test class that ensures that the Stage II strategy algorithms are returning the desired values. The diagram in Figure 1 shows the results of these tests. It can be seen that the naive strategy does indeed pick the highest valid choice and the greedy strategy is an improvement:

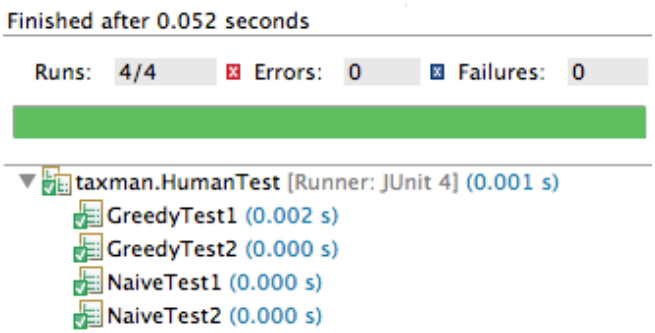

Figure 1 – HumanTest JUnit Results

```
package taxman;

import static org.junit.Assert.*;
import java.util.LinkedHashSet;
import org.junit.Test;

public class HumanTest {
	Human player = new Human();

	/**
	 * The first test for @see {@link Human#naiveMove(LinkedHashSet)} to assert
	 * that the largest valid choice is taken.
	 */
	@Test
	public void NaiveTest1() {
		Collection col = new Collection(6);
		LinkedHashSet<Coin> set = col.getCollection();

		try {
			// Assert largest valid choice is first chosen
			assertEquals(player.naiveMove(set), 6);

		} catch (TaxmanError e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	/**
	 * The second test for the @see {@link Human#naiveMove(LinkedHashSet)} method,
asserting
	 * that the largest valid choice is taken.
	 */
	@Test
	public void NaiveTest2() {
		Collection col = new Collection(25);
		LinkedHashSet<Coin> set = col.getCollection();

		try {
			assertEquals(player.naiveMove(set), 25);
		} catch (TaxmanError e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	/**
	 * The first test for @see {@link Human#naiveMove(LinkedHashSet)} to assert
	 * that a greedy choice is made, smarter than the naive move of '6'.
	 */
	@Test
	public void GreedyTest1() {
```

```
                Collection col = new Collection(6);
                LinkedHashSet<Coin> set = col.getCollection();

                try {
                        // Assert a smarter 'greedy' choice is made
                        assertEquals(player.greedyMove(set), 5);

                } catch (TaxmanError e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

        /**
         * The first second for @see {@link Human#naiveMove(LinkedHashSet)} to assert
         * that a greedy choice is made, smarter than the naive move of '25'.
         */
        @Test
        public void GreedyTest2() {
                Collection col = new Collection(25);
                LinkedHashSet<Coin> set = col.getCollection();

                try {
                        // Assert a smarter 'greedy' choice is made
                        assertEquals(player.greedyMove(set), 23);

                } catch (TaxmanError e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

}
```

Player

Player is the superclass that the Human and Taxman classes extend. The class contains methods to take coins, return the respective player's collection and get the score statistics of the specified instance of the player.

```
package taxman;

import java.util.Iterator;

/**
 * Player is the class used to define the behaviour to be inherrited by the
 * <code>Human</code> and <code>Taxman</code>.
 */
public class Player {
        // Declare and instantiate the Collection each player has
        private Collection collection = new Collection();
        private Iterator<Coin> it;

        /**
         * The constructor for the player.
         */
        public Player() {
        }

        /**
         * Used to move a coin from the table to the player's collection.
         *
         * @param table
```

```
	 *            containing the collection of coins in play
	 * @param coin
	 *            that will be taken
	 */
	public void take(Table table, Coin coin) {
		table.move(collection, coin);
	}

	/**
	 * Moves the coin from the table that has the value from the method
	 * parameter to the player's collection.
	 *
	 * @param table
	 *            containing the collection of coins in play
	 * @param cVal
	 *            is the value of the coin
	 */
	public void take(Table table, int cVal) {
		// Iterate through the table of coins
		it = table.getCollection().iterator();
		while (it.hasNext()) {
			Coin tC = it.next();
			// If the coin is found, move it to the player's collection
			if (tC.getValue() == cVal) {
				table.move(collection, tC);
				it = table.getCollection().iterator();
			}
		}
	}

	/**
	 * Return the player's <code>Collection</code> of coins obtained during the
	 * game.
	 *
	 * @return the collection of the player
	 */
	public Collection getCollection() {
		return collection;
	}

	/**
	 * Gets the number of coins the player has and the combined value of those
	 * coins.
	 *
	 * @return the number of coins and thei total value in an int array
	 */
	public int[] getScore() {
		return new int[] { collection.getNoCoins(), collection.getValue() };
	}

}
```

Table

A Table instance is created to contain the Collection that is still in play. Along with the Human and Taxman, these are one of the objects that contain collections of coins. Getter methods return the current state of the table and the coins that are valid for taking by the player or ‘taxable’ according to the rules of the game are calculated in methods inside this class.

```
package taxman;

import java.util.Iterator;
```

```java
import java.util.LinkedHashSet;

/**
 * Table is a subclass to the Collection class and is used to manage the
 * <code>Collection</code> of coins in play.
 *
 * @author Joseph Lewis
 *
 */
public class Table extends Collection {
	// Declare variables for the table
	private LinkedHashSet<Coin> taxable = new LinkedHashSet<Coin>();
	private static Iterator<Coin> it;

	/**
	 * Constructor for Table that fill the table with a collection of coins to
	 * the upper limit given by the human when an instance is created.
	 *
	 * @param uLim
	 * @throws TaxmanError
	 */
	public Table(int uLim) throws TaxmanError {
		// Sends uLim back to the superclass
		super(uLim);
		// Adds coins to the taxable Collection
		// Note that 1 can never be chosen as it has no divisors
		for (int i = 1; i < this.getSize(); i++) {
			taxable.add(this.getCoin(i + 1));
		}
	}

	/**
	 * Gets the <code>Collection</code> of 'taxable' coins on the table.
	 *
	 * @return the collection of taxable coins
	 */
	public LinkedHashSet<Coin> getTaxableCoins() {
		return taxable;
	}

	/**
	 * Used to return a string representation of the coins left in the game.
	 * Coins that are valid for taking by the player are surrounded with pipes.
	 *
	 * @return the game state representation for a command line UI
	 */
	public String getTableState() {
		String tableStr = "";
		it = this.getCollection().iterator();
		Coin currentCoin;
		// Iterate through the collection on the table
		while (it.hasNext()) {
			currentCoin = it.next();
			// If the coin is taxable, represent it with pipes surrounding it
			if (taxable.contains(currentCoin)) {
				tableStr += "|" + String.valueOf(currentCoin.getValue()) + "|,
";
			} else {
				if (currentCoin != null) {
					tableStr += String.valueOf(currentCoin.getValue()) + ",
";
				}
			}

		}
```

```
                // Return the table coin values as a string for visualisation
                return tableStr.substring(0, tableStr.length() - 2);
        }

        /**
         * Used to calculate the coins on the table that are taxable, storing them
         * in a private <code>LinkedHashSet</code> for the <code>Table</code> class.
         */
        public void calTaxable() {
                taxable = new LinkedHashSet<Coin>();
                it = this.getCollection().iterator();
                // Iterates through each coin on the table
                while (it.hasNext()) {
                        Coin c1 = it.next();
                        Iterator<Coin> it2 = this.getCollection().iterator();
                        while (it2.hasNext()) {
                                Coin c2 = it2.next();
                                // Calculates if a coin has a divisor
                                if (c1.getValue() > c2.getValue() && c1.getValue() %
c2.getValue() == 0) {
                                        taxable.add(c1);
                                }
                        }
                }
        }

        /**
         * This method is used to calculate which coins are valid for taking or
         * 'taxable' from a set given as a parameter. The valid coins are returned
         * in the same format.
         *
         * @param coinsInPlay
         *            is the given set of coins
         * @return the valid coins to be chosen in that set
         */
        public static LinkedHashSet<Coin> getTaxable(LinkedHashSet<Coin> coinsInPlay) {
                LinkedHashSet<Coin> taxableInPlay = new LinkedHashSet<Coin>();
                it = coinsInPlay.iterator();
                // Iterates through each coin on the table
                while (it.hasNext()) {
                        Coin c1 = it.next();
                        Iterator<Coin> it2 = coinsInPlay.iterator();
                        while (it2.hasNext()) {
                                Coin c2 = it2.next();
                                // Calculates if a coin has a divisor
                                if ((c1.getValue() > c2.getValue() && c1.getValue() %
c2.getValue() == 0)) {
                                        taxableInPlay.add(c1);
                                }
                        }
                }
                return taxableInPlay;
        }
}
```

Taxman

Taxman is a subclass of Player and plays the role of the Taxman. The only specific method specific to this Player is to take the remaining coins on the table when there are no valid choices left for the Human Player to make.

```
package taxman;
```

```java
import java.util.Iterator;

/**
 * The Taxman, a subclass of Player used in the game.
 *
 * @author Joseh Lewis
 *
 */
public class Taxman extends Player {
	private Iterator<Coin> it;

	/**
	 * The empty Taxman constructor used to create <code>Taxman</code> instances
	 */
	public Taxman() {
	}

	/**
	 * Adds coins from the given table to the <code>Collection</code> of the
	 * taxman. This method is called when there are no 'taxable' coins left on
	 * the table.
	 *
	 * @param table
	 *            is the specified table
	 */
	public void takeTable(Table table) {
		it = table.getCollection().iterator();
		// Iterates through the table's coin collection
		while (it.hasNext()) {
			Coin tC = it.next();
			// Add's a coin to the taxman's collection for each coin on the
			// table
			this.getCollection().add(table.getCoin(tC.getValue()));
		}
	}
}
```

TaxmanError

A custom Error class that handles exceptions specifically related to game rule logic. TaxmanErrors are thrown from other classes and are set to be handled so the program does not end unexpectedly, with information usually printed to the console.

```java
package taxman;

/**
 * TaxmanError is a custom Exception subclass, used to throw and catch
 * exceptions specific to the game.
 *
 * @author Joseph Lewis
 *
 */
public class TaxmanError extends Exception {
	// Generated serialVersionUID
	private static final long serialVersionUID = -1001216979081256204L;

	/**
	 * TaxmanError constructor that takes an <code>TaxmanError</code> and calls
	 * the parents constructor to display the error.
	 *
	 * @param te
	 *            is the description of the <code>TaxmanError</code>
	 */
```

```
	public TaxmanError(String te) {
		super(te);
	}
}
```

TaxmanOperator

An instance of Game is created in this class working as the operator for the program. It contains the Main method for the project and uses logic to direct the game through the desired flow while printing feedback for the user to look at while entering keyboard input to control parts of the game.

```
package taxman;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * TaxmanOperator is used run the control of the game. A <code>Game</code>
 * instance is created after the initial setup has complete.
 *
 * @author Joseph Lewis
 *
 */
public class TaxmanOperator {
	// Declare variables
	public static Game game;
	public static Human inputPlayer;
	private static int upperLimit;
	private static BufferedReader keyboard;

	/**
	 * The use of BufferedReader is a way to capture user input. This method is
	 * used to capture the next line of input a user enters on their keybpard.
	 *
	 * @return the string of 3
	 * 7the entered information
	 */
	private static String getInput() {
		String in = "";
		try {
			in = keyboard.readLine();
		} catch (IOException e) {
			System.out.println(e.getMessage());
		}
		return in;
	}

	/**
	 * This method is used to capture the users input to select both the game
	 * mode and upper limit for the round.
	 *
	 * @throws TaxmanError
	 *		if input invalid to the choices made available is entered
	 */
	public static void setupGame() throws TaxmanError {
		System.out.println("------ Let's play! ------");
		int mode = 0;
		boolean t = true;
		while (t) {
			try {
				// User selects the game mode
```

```
                                System.out.println("1 - Human       2 - Naive\n3 - Greedy
4 - Optimal");
                                System.out.print("> To select a game mode, enter the number
corresponding: ");
                                mode = Integer.valueOf(getInput());
                                // If the the input is out of range, throw a TaxmanError
                                if (!(mode > 0) || !(mode < 5)) {
                                        TaxmanError te = new TaxmanError("Choose a mode from 1-
4.");
                                        throw te;
                                }
                                // User chooses an upper limit
                                System.out.print("> Choose the upper limit for the round: ");
                                String inputNum = getInput();
                                if (Integer.valueOf(inputNum) >= 2) {
                                        upperLimit = Integer.valueOf(inputNum);
                                        t = false;
                                        // Throw a TaxmanError if the limit is out of range
                                } else {
                                        TaxmanError te = new TaxmanError("Select a legal upper
limit >= 2.");
                                        throw te;
                                }
                                // Catch any errors and re-attempt to setup the game
                        } catch (NumberFormatException e) {
                                System.out.println("> Input needs to be a valid number. Let's
try again.");
                                setupGame();
                        } catch (TaxmanError te) {
                                System.out.println("> " + te + " Let's try again.");
                                setupGame();
                        }
                }
                startGame(mode);
        }

        /**
         * Used to start the game by creating an instance of <code>Game</code> with
         * the specified upper limit. While the game hasn't finished, moves will be
         * made by the human or computer, depending on the game mode. When the game
         * has ended the winner will be printed in the console.
         *
         * @param mode
         *            is the game mode (Stage) to play with
         * @throws TaxmanError
         *             if the move selected in invalid due to the rules of the game
         */
        public static void startGame(int mode) throws TaxmanError {
                // Create a Game instance
                try {
                        game = new Game(upperLimit);
                } catch (TaxmanError e) {
                        System.out.println(e);
                }
                String stateOfPlay;
                // Loop until game has ended
                while (!game.gameOver()) {
                        // Print the current coin collection of the table
                        stateOfPlay = game.getTable().getTableState();
                        stateOfPlay += "\nHuman has " + game.getHuman().getScore()[0] + " coins
totaling "
                                        + game.getHuman().getScore()[1] + ".";
                        stateOfPlay += "\nTaxman has " + game.getTaxman().getScore()[0] + "
coins totaling "
                                        + game.getTaxman().getScore()[1] + ".";
```

```
                        System.out.println(stateOfPlay);
                        // Select a move determined by the game mode

                        try {
                                String moveInput = null;
                                int moveInt = -1;
                                // If mode '1', get user input
                                if (mode == 1) {
                                        System.out.print("> Enter the number of the coin to
take: ");
                                        moveInput = getInput();
                                        if (moveInput == null) {
                                                System.out.println("> Null input detected.
Terminating program...");
                                                return;
                                        }
                                        moveInt = Integer.parseInt(moveInput);
                                } else if (mode == 2) { // Modes 2, 3 and 4 used algorithms to
                                                                                        // compute the
move to choose
                                        moveInt =
game.getHuman().naiveMove(game.getTable().getTaxableCoins());
                                        System.out.println("> The computer has chosen to take
coin '" + moveInt + "'.");
                                } else if (mode == 3) {
                                        moveInt =
game.getHuman().greedyMove(game.getTable().getCollection());
                                        System.out.println("> The computer has chosen to take
coin '" + moveInt + "'.");
                                } else if (mode == 4) {
                                        // Prepare for the recursive back

                game.getHuman().preOptimal(game.getTable().getCollection().size());
                                        moveInt =
game.getHuman().optimalMove(game.getTable().getCollection()).get(0);
                                        System.out.println("> The computer has chosen to take
coin '" + moveInt + "'.");
                                }
                                // Play the selected move in the game
                                game.getHuman().chooseCoin(game, moveInt);
                                // Catch invalid input or errors to be handled
                        } catch (NumberFormatException e) {
                                System.out.println("> The choice must be a valid number
displayed above.");
                        } catch (TaxmanError te) {
                                System.out.println("> " + te.getMessage());
                        }
                }
                // Once the game has ended print the game state one last time
                stateOfPlay = "----- There are no more coins left -----";
                stateOfPlay += "\nHuman has " + game.getHuman().getScore()[0] + " coins
totaling "
                                + game.getHuman().getScore()[1] + ".";
                stateOfPlay += "\nTaxman has " + game.getTaxman().getScore()[0] + " coins
totaling "
                                + game.getTaxman().getScore()[1] + ".";
                System.out.println(stateOfPlay);
                // Determine the winner to be printed in the console
                String outcome;
                if (game.getTaxman().getCollection().getValue() >
game.getHuman().getCollection().getValue()) {
                        outcome = "Taxman as the winner";
                } else if (game.getTaxman().getCollection().getValue() <
game.getHuman().getCollection().getValue()) {
                        outcome = "Human as the winner";
```

```
                } else {
                        outcome = "a draw";
                }
                // Prompt the user to play again if they enter 'y'
                System.out.print("----- The game has ended with " + outcome + ". -----\n>
Enter 'y' to play again: ");
                if (getInput().equals("y")) {
                        setupGame();
                } else {
                        System.out.print("Thanks for playing!");
                }
        }

        /**
         * A getter used to return the current game when called.
         *
         * @return the game
         */
        public static Game getGame() {
                return game;
        }

        /**
         * Getter used to return the upper limit of the current game.
         *
         * @return the upper limit
         */
        public static int getUpperLimit() {
                return upperLimit;
        }

        /**
         * The Main method of the program that starts the Main daemon. The
         * BufferedReader is initialised and the @see {@link TaxmanOperator#setupGame()}
method is called to
         * start the game.
         *
         * @param args
         *             is input that can be entered from a command prompt / terminal
         * @throws TaxmanError
         *             if an Exception specific to the game arises
         */
        public static void main(String[] args) {
                keyboard = new BufferedReader(new InputStreamReader(System.in));
                System.out.println("----- The Taxman Cometh -----\nFollow the prompts to play
the game.");
                try {
                        setupGame();
                } catch (TaxmanError te) {
                        System.out.println("> " + te);
                }
        }

}
```

3. Results

The output for the game stages implemented in the program are printed or tubulised below, for given upper limits, N. Some output printed during the game is also shown.

3.1 During the Game

When the program first runs, text is printed in the console welcoming the player, prompting them to choose a game mode and upper limit. Pasted below is the console output for a game equivalent to Stage I with the upper limit 6.

```
----- The Taxman Cometh -----
Follow the prompts to play the game.
------ Let's play! ------
1 - Human       2 - Naive
3 - Greedy      4 - Optimal
> To select a game mode, enter the number corresponding: 1
> Choose the upper limit for the round: 6
1, |2|, |3|, |4|, |5|, |6|
Human has 0 coins totaling 0.
Taxman has 0 coins totaling 0.
> Enter the number of the coin to take:
```

3.2 Stage I

The output of the following Stage I game rounds will be displayed below, starting with N = 6.

N = 6

```
1, |2|, |3|, |4|, |5|, |6|
Human has 0 coins totaling 0.
Taxman has 0 coins totaling 0.
> Enter the number of the coin to take: 5
2, 3, |4|, |6|
Human has 1 coins totaling 5.
Taxman has 1 coins totaling 1.
> Enter the number of the coin to take: 4
3, |6|
Human has 2 coins totaling 9.
Taxman has 2 coins totaling 3.
> Enter the number of the coin to take: 6
----- There are no more coins left -----
Human has 3 coins totaling 15.
Taxman has 3 coins totaling 6.
----- The game has ended with Human as the winner. -----
```

N = 11

```
1, |2|, |3|, |4|, |5|, |6|, |7|, |8|, |9|, |10|, |11|
Human has 0 coins totaling 0.
Taxman has 0 coins totaling 0.
> Enter the number of the coin to take: 11
2, 3, |4|, 5, |6|, 7, |8|, |9|, |10|
Human has 1 coins totaling 11.
Taxman has 1 coins totaling 1.
> Enter the number of the coin to take: 9
2, |4|, 5, |6|, 7, |8|, |10|
Human has 2 coins totaling 20.
Taxman has 2 coins totaling 4.
> Enter the number of the coin to take: 6
4, 5, 7, |8|, |10|
Human has 3 coins totaling 26.
Taxman has 3 coins totaling 6.
> Enter the number of the coin to take: 10
4, 7, |8|
Human has 4 coins totaling 36.
Taxman has 4 coins totaling 11.
```

```
> Enter the number of the coin to take: 8
----- There are no more coins left -----
Human has 5 coins totaling 44.
Taxman has 6 coins totaling 22.
----- The game has ended with Human as the winner. -----
```

N = 15

```
1, |2|, |3|, |4|, |5|, |6|, |7|, |8|, |9|, |10|, |11|, |12|, |13|, |14|, |15|
Human has 0 coins totaling 0.
Taxman has 0 coins totaling 0.
> Enter the number of the coin to take: 13
2, 3, |4|, 5, |6|, 7, |8|, |9|, |10|, 11, |12|, |14|, |15|
Human has 1 coins totaling 13.
Taxman has 1 coins totaling 1.
> Enter the number of the coin to take: 9
2, |4|, 5, |6|, 7, |8|, |10|, 11, |12|, |14|, |15|
Human has 2 coins totaling 22.
Taxman has 2 coins totaling 4.
> Enter the number of the coin to take: 15
2, |4|, |6|, 7, |8|, |10|, 11, |12|, |14|
Human has 3 coins totaling 37.
Taxman has 3 coins totaling 9.
> Enter the number of the coin to take: 10
4, 6, 7, |8|, 11, |12|, |14|
Human has 4 coins totaling 47.
Taxman has 4 coins totaling 11.
> Enter the number of the coin to take: 14
4, 6, |8|, 11, |12|
Human has 5 coins totaling 61.
Taxman has 5 coins totaling 18.
> Enter the number of the coin to take: 8
6, 11, |12|
Human has 6 coins totaling 69.
Taxman has 6 coins totaling 22.
> Enter the number of the coin to take: 12
----- There are no more coins left -----
Human has 7 coins totaling 81.
Taxman has 8 coins totaling 39.
----- The game has ended with Human as the winner. -----
```

N = 25

```
1, |2|, |3|, |4|, |5|, |6|, |7|, |8|, |9|, |10|, |11|, |12|, |13|, |14|, |15|, |16|, |17|,
|18|, |19|, |20|, |21|, |22|, |23|, |24|, |25|
Human has 0 coins totaling 0.
Taxman has 0 coins totaling 0.
> Enter the number of the coin to take: 23
2, 3, |4|, 5, |6|, 7, |8|, |9|, |10|, 11, |12|, 13, |14|, |15|, |16|, 17, |18|, 19, |20|,
|21|, |22|, |24|, |25|
Human has 1 coins totaling 23.
Taxman has 1 coins totaling 1.
> Enter the number of the coin to take: 25
2, 3, |4|, |6|, 7, |8|, |9|, |10|, 11, |12|, 13, |14|, |15|, |16|, 17, |18|, 19, |20|, |21|,
|22|, |24|
Human has 2 coins totaling 48.
Taxman has 2 coins totaling 6.
> Enter the number of the coin to take: 15
2, |4|, |6|, 7, |8|, 9, |10|, 11, |12|, 13, |14|, |16|, 17, |18|, 19, |20|, |21|, |22|, |24|
Human has 3 coins totaling 63.
Taxman has 3 coins totaling 9.
> Enter the number of the coin to take: 21
2, |4|, |6|, |8|, 9, |10|, 11, |12|, 13, |14|, |16|, 17, |18|, 19, |20|, |22|, |24|
Human has 4 coins totaling 84.
```

```
Taxman has 4 coins totaling 16.
> Enter the number of the coin to take: 14
4, 6, |8|, 9, 10, 11, |12|, 13, |16|, 17, |18|, 19, |20|, |22|, |24|
Human has 5 coins totaling 98.
Taxman has 5 coins totaling 18.
> Enter the number of the coin to take: 22
4, 6, |8|, 9, 10, |12|, 13, |16|, 17, |18|, 19, |20|, |24|
Human has 6 coins totaling 120.
Taxman has 6 coins totaling 29.
> Enter the number of the coin to take: 20
6, 8, 9, |12|, 13, |16|, 17, |18|, 19, |24|
Human has 7 coins totaling 140.
Taxman has 8 coins totaling 43.
> Enter the number of the coin to take: 16
6, 9, |12|, 13, 17, |18|, 19, |24|
Human has 8 coins totaling 156.
Taxman has 9 coins totaling 51.
> Enter the number of the coin to take: 24
9, 13, 17, |18|, 19
Human has 9 coins totaling 180.
Taxman has 11 coins totaling 69.
> Enter the number of the coin to take: 18
----- There are no more coins left -----
Human has 10 coins totaling 198.
Taxman has 15 coins totaling 127.
----- The game has ended with Human as the winner. -----
```

3.3 Stage II and III

To save space in the Stage II results, only the moves selected by the algorithm and resulting total will be displayed, along with the winning margin for the strategy. Using temporary modifications to the startGame(int mode) method in TaxmanOperator to remove most of the printing functionality and changing setupGame() to the following to automate the output, the feedback will be displayed.

```
public static void setupGame() throws TaxmanError {
        for (int i = 2; i <= 25; i++) {
                upperLimit = i;
                System.out.println("N=" + i);
                startGame(3); // 2 for naive, 3 for greedy
                System.out.println();
        }
}
```

Naive

Before getting to the greedy and optimal strategies, directly below is the console output for the primitive naive method of choosing coins. It can be seen that the Taxman wins the majority of games using this strategy, rendering it ineffective.

```
N=2
2 | Sum: 2, Taxman: 1
N=3
3 | Sum: 3, Taxman: 3
N=4
4 | Sum: 4, Taxman: 6
N=5
5 4 | Sum: 9, Taxman: 6
N=6
6 | Sum: 6, Taxman: 15
N=7
```

```
7 6 | Sum: 13, Taxman: 15
N=8
8 6 | Sum: 14, Taxman: 22
N=9
9 8 | Sum: 17, Taxman: 28
N=10
10 9 8 | Sum: 27, Taxman: 28
N=11
11 10 9 8 | Sum: 38, Taxman: 28
N=12
12 10 | Sum: 22, Taxman: 56
N=13
13 12 10 | Sum: 35, Taxman: 56
N=14
14 12 10 | Sum: 36, Taxman: 69
N=15
15 14 12 | Sum: 41, Taxman: 79
N=16
16 15 14 12 | Sum: 57, Taxman: 79
N=17
17 16 15 14 12 | Sum: 74, Taxman: 79
N=18
18 16 15 14 | Sum: 63, Taxman: 108
N=19
19 18 16 15 14 | Sum: 82, Taxman: 108
N=20
20 18 16 14 | Sum: 68, Taxman: 142
N=21
21 20 18 16 | Sum: 75, Taxman: 156
N=22
22 21 20 18 16 | Sum: 97, Taxman: 156
N=23
23 22 21 20 18 16 | Sum: 120, Taxman: 156
N=24
24 22 21 20 18 | Sum: 105, Taxman: 195
N=25
25 24 22 21 20 18 | Sum: 130, Taxman: 195
```

Greedy and Optimal

The coin choices for the greedy and optimal strategies along with the winning margins will be presented in the following table to compare the effectiveness of the two strategies. The results for the greedy strategy were generated the same way that the naive method output was obtained. The data from the optimal method was obtained by making modifications to the getOptimal() method for different values as the algorithm only worked for certain values of N dependant on the pattern.

N	Strategy (Greedy)	Total	Winning margin	Strategy (Optimal)	Total	Winning margin
2	2	2	1	2	2	1
3	3	3	0	3	3	0
4	3, 4	7	4	3, 4	7	4
5	5, 4	9	3	5, 4	9	3
6	5, 4, 6	15	9	5, 4, 6	15	9
7	7, 4, 6	17	6	7, 4, 6	17	6
8	7, 4, 6	17	-2	7, 8, 6	21	6
9	7, 9, 6, 8	30	15	7, 9, 6, 8	30	15
10	7, 9, 6, 8, 10	40	25	7, 9, 6, 8, 10	40	25
11	11, 9, 6 ,8 10	44	22	11, 9, 6 ,8 10	44	22
12	11, 9, 6, 12, 10	48	18	11, 9, 10, 8, 12	50	22
13	13, 9, 6, 12, 10	50	9	13, 9, 10, 8, 12	52	13
14	13, 9, 14, 10, 8, 12	66	27	13, 9, 14, 10, 8, 12	66	27
15	13, 15, 10, 14, 8, 12	72	24	13, 9, 15, 10, 14, 8, 12	81	42
16	13, 15, 10, 14, 8, 12	72	8	13, 9, 15, 10, 14, 16, 12	89	42
17	17, 15, 10, 14, 8, 12	76	-1	17, 9, 15, 10, 14, 16, 12	93	33
18	17, 15, 10, 14, 8, 12, 18	94	17	17, 9, 15, 10, 18, 14, 12, 16	111	51
19	19, 15, 10, 14, 8, 18	96	2	19, 9, 15, 10, 18, 14, 12, 16	113	36
20	19, 15, 10, 20, 16, 14, 12, 18	124	38	19, 15, 10, 20 16, 14 12, 18	134	58
21	19, 21, 14, 15, 20, 16, 12, 18	135	39	19, 9, 21, 15, 14, 18, 12, 20, 16	144	57
22	19, 21, 14, 22, 15, 20, 16, 12, 18	157	61	19, 9, 21, 15, 14, 22, 18, 12, 20, 16	166	79
23	23, 21, 14, 22, 15, 20, 16, 12, 18	161	46	23, 9, 21, 15, 14, 22, 18, 12, 20, 16	170	64
24	23, 21, 14, 22, 15, 20, 16, 12, 18	161	22	23, 9, 21, 15, 14, 22, 20, 18, 16, 24	182	64
25	23, 25, 15, 21, 14, 22, 20, 16, 12, 18	186	47	23, 25, 15, 21, 14, 22, 20, 16, 24, 18	198	71

4. Discussion

It was specified that Stage I is interactive in gaining input from the user each turn. Due to this reasoning it is safe to say a decent level of interface design would be required. Subsection 3.1 shows the level of design implemented. While a traditional GUI is not used for the game, the output printed to the console is comprehensive and has added characters to stylise the experience and ensure clarity and good game-flow.

During runtime when the game starts text appears to welcome the player to the game and then prompts them to choose a game mode and upper limit for the game. Each turn the coins left in the game is printed; the coins that are valid choices are surrounded by pipes to provide conformation that the integers have divisors left in the game. Each turn the total amount of coins taken by the player and taxman are displayed along with their combined sum. This pattern continues throughout the round until the game has ended, with additional output printed to display the winner and their score. From here the user is prompted to choose if they would like to play again and given an exit message if they don't. Stages II and III used the same design for producing output, the difference being that 'The computer has chosen…' was printed instead of 'Enter the number of coins to take: '.

The approach taken for Stage I came from the design; prompt the user to enter input, and handle that input accordingly. Loops were used to repeat asking for input until validated in each part of the program, as human are prone to make mistakes and are not controlled like algorithms. While loops were also used to repeat the game flow until the end of the game.

Stage II extends from the Stage I design, however the approach diverges. The implementation for this strategy is still included in the game's main control flow, dictated by the user input at the start of the round. The naive method simply accepts a LinkedHashSet containing the coins in the stage of the game, calls a method to determine the valid choices, and then iterates to the last element to return. This works as when the set is first created, the elements are in order of increasing value.

The greedy strategy is more advanced and employs a more complex strategy… Similarly to a nested for loop, the greedyMove() method contains an algorithm that iterates through each 'taxable' coin remaining, and inside the while loop, iterates through each coin left in play. A pseudocode algorithm is shown below to demonstrate what happens inside the second while loop; if the current taxable coin is greater than the coin that is in play currently pointed to by the iterator, and the remainder of the division of the two is 0, add the current coin's value to the Taxman's total:

```
if (CURRENT_TAXABLE > CURRENT_ALL && CURRENT_TAXABLE % CURRENT_ALL = 0) {
        TAXMAN_TOTAL += CURRENT_ALL
}
```

Once all the coins in play have been iterated through, if the value of CURRENT_TAXABLE minus TAXMAN_TOTAL leads to the largest number recorded so far, it is then stored as the 'bestMoveSoFar'. This happens for every taxable coin and the best move at the end is returned.

The optimal strategy in Stage III uses the optimalMove() method as well as two helper methods: preOptimal() and nextPath() to complete the recursive backtracking needed to find every combination of the game tree for a given upper limit. preOptimal() accepts the upper limit as a parameter to initialise an array used to keep track of the current node path and to assign default values to required variables. The method for finding the optimal move follows the standard backtracking algorithm, with a few modifications:

1. If all nodes have been explored, return the optimal move.

2. If the current node has a child, continue the recursion.
3. Increment the current selection using nextPath() at the end of each node.
4. Return the optimal move.

The current algorithm is an improvement of a previous attempt using both recursive backtracking and for loops to complete the algorithm. The algorithm was modified as the method only worked for certain upper limits, N. The current implementation is not perfect, and works differently to all attempts of creating an optimal strategy algorithm.

The time complexity of the greedy algorithm compared to the optimal strategy results in a much quicker runtime for the greedy strategy. A call of the greedyMove() method took 0.000 seconds to complete a return for upper limits, N, where N <= 25. Conversely a call of the optimalMove() method took between 3-5 seconds to return values for N, where N <= 25. This is due to brute-forcing being an expensive process, drastically increasing in time for each increment of N.

The winning margins for the optimal strategy were obviously either equal to or better than the respective plays for the greedy strategy. For N=2 to 25, the greedy method only lost twice and for the majority was not too far behind the optimal methods results. For most values of N less than 15, the greedy method implementation was on par to the optimal strategy and with a much better time complexity.

Due to the nature of factors, from this research it is inconclusive to whether a simple algorithm can be implemented to reach the optimal winning margin for every integer N, with N > 1. From the data it is more effective to use the greedy strategy as the time difference is quite massive and the margins are not too different. It is possible that a better implementation of the greedy method could also negate the one advantage that the optimal strategy has over the greedy counterpart – it always results in the optimal play.