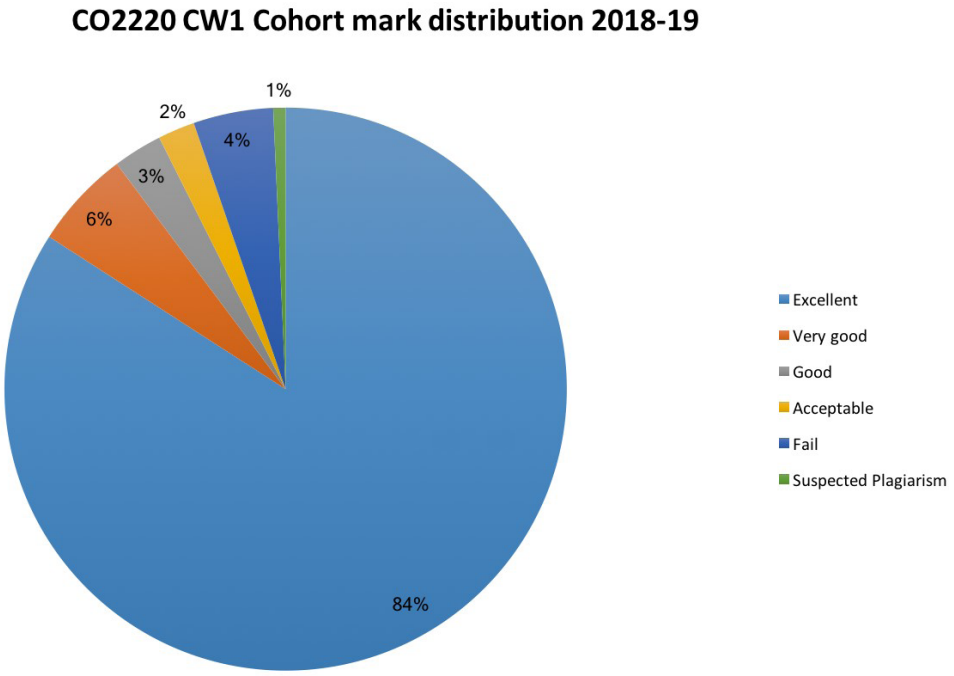
Coursework commentary 2018–2019

CO2220 Graphical object-oriented and internet programming in Java

Coursework assignment 1

On the whole this coursework assignment was attempted well, with the majority of students gaining excellent marks.

See 2018–2019 cohort mark distribution for CO2220 CW1 below:



Model answers

The following files, which include model answers, are provided with this report:

Part A

- *Hangman.java*
- *AbstractRandomWordGame.java (with answer to question 2)*
- *WordGamesArcade.java (with answer to question 5)*
- *WordSleuth.java (answer to questions 3 and 4)*
- *gamedictionary.txt*

Part B

- *AnimatedGUI.java (with answers to questions 1 and 2)*

Comments on specific questions

Part A: Wordsleuth

In Part A, students were given the *WordGamesArcade* class, together with the *Hangman* and *AbstractRandomWordGame* classes. The *AbstractRandomWordGame* class read in from a list of words in a text file and saved them in a `List`. The *Hangman* class extended the *AbstractRandomWordGame* class. The *WordGamesArcade* class showed the user a menu, allowing the user to play *Hangman* or quit.

Students were asked to write a word game, called *WordSleuth*, and implement its rules precisely. Following this they were asked to link the game to the *WordGamesArcade* class, such that when running the class, the option to play *WordSleuth* could be accessed by the user.

Question 1

Students were asked to write a comment in the *Hangman* class with suggestions for improvements. This was answered well overall. Most students suggested that the player should be shown the rules of the game, and that input should be limited to alphabetic characters only. Many students missed that an exception would be thrown if the player accidentally pressed enter on a blank line. Those that had noticed this made the very good point that this exception should be handled so that player error would not end the game.

Some students lost marks by writing very brief comments that inevitably missed obvious improvements, such as displaying the rules or only allowing characters from the alphabet. A minority made readability suggestions for the Hangman class, a few were trivial, but some showed a very good understanding of readability.

Question 2

The developer had left two *TODO* comments in the *AbstractRandomWordGame* class, relating to the *loadWords()* method. This method attempted to make a `List` of words from a file, but if the file could not be accessed, it would instead make a short `List` of default words. The access level of the method was private, meaning that it could not be overridden in subclasses. The *TODO* comments were:

```
//TODO: protected or private?
//TODO: rename to tell of side effects
```

Students were asked to take the actions described by the comments.

A good response to the first comment might have been to change the access modifier to protected, allowing a subclass to override the method. A subclass might want to change the error message given if the file would not load, and/or how the default `List` is loaded. An even better answer would have made the method private, and put its actions in returning an error message and a default `List` into separate *protected* methods as follows:

```
private void loadWordsOrGetDefaultWords() {
   if (wordsFilePath == null) wordsFilePath = defaultWordsFilePath;
   if (wordsFileCharset == null) wordsFileCharset =
            defaultWordsFileCharset;
   try {
      words = Files.readAllLines(wordsFilePath, wordsFileCharset);
   } catch (IOException e) {
         reportWordsLoadingError(e);
         words = getDefaultWordsList();
      }
}
```

```
protected void reportWordsLoadingError(IOException e) {
   System.err.println("Error reading file '" +wordsFilePath+ "'");
}

protected List<String> getDefaultWordsList() {
   return Arrays.asList("compilation", "popstar", "symphony");
}
```

This would allow subclasses to override one behaviour, or both, as the developer wished.

A good answer would have dealt with the second comment by renaming the *loadWords()* method to describe the default option taken if the file could not be found, for example:

loadWordsOrGetDefaultWords().

Most students did not rename *loadWords(),* and the few who did, used names that showed they had not understood that it was important to include the default action taken by the method in the name. For example names included: *loadWordsFromFile()* and *loadDictionary()*. Students who did not rename *loadWords()* at all, or appropriately, lost some marks for this question. Nevertheless, some new names, such as *loadWordsOrUseDefaultList(), loadDictionaryOrUsePresetWords()* and *loadWordsOrUseDefaultWordList()* showed excellent understanding.

A few students deleted the side effect of loading a default `List` instead of renaming the method. This is arguably a reasonable approach, but it would remove from subclasses the option of dealing with the default `List` in a different way.

Another mistake made by a few students was in putting the side effect into a separate method, invoked by the *loadWords()* method, and then claiming that the *loadWords()* method did not need renaming since the side effect had been put into a separate method. This is both right and wrong. It was good practice to put loading the default `List` into a separate method, but *loadWords()* still invoked the method, and hence still had the side effect and needed to be renamed to reflect this.

Question 3

Students were given the rules of a game called *WordSleuth*, and asked to implement the game as a Java class that, like *Hangman*, inherited from the *AbstractRandomWordGame* class. Most students made a reasonable attempt at the new class. The rules for *WordSleuth* were complicated, so it was not surprising that some students made minor errors, such as with calculating the player's score or the number of guesses, while still making a good attempt at the class.

Minor errors: The most common error relating to calculating the number of guesses was that the total would be off by one. Sometimes the player would get one more guess than they should, and sometimes one less. Sometimes the error happened for every word; sometimes the error only appeared for words with an odd number of characters.

Common minor errors relating to scoring were:

- Before the player's final guess, one mark would be deducted from the potential final score, followed by two marks being deducted, giving a final score that was one mark lower than it should have been. The rules stated that one mark should be deducted from the potential score after each wrong guess, except for before the final guess when two marks should be deducted. This was probably the most common of all errors.

- Before the final guess one mark, instead of two, would be deducted from the potential final score.
- The player would be given a positive score after failing to guess the word.

There were other idiosyncratic minor errors with score calculation, as well as a tiny minority of students who set the score to a constant value that was not updated as the player progressed through their guesses.

Other common minor errors were:

- The rules state that players get given a random character from the word before each guess, except for the last guess, when they are shown a final guess `String`, consisting of the characters they have seen so far in the order they appear in the word. Despite this, often players were shown another random character from the word before their final guess. This character would then be included in the final guess `String`.
- When the final guess `String` was displayed, the player was not told that the random characters were now in the order that they appeared in the word.
- The rules stated that at the start of the game the player would be shown the first character in the word, together with the number of characters in the word. This meant that the first character should be included in the final guess `String`, but sometimes it was not, possibly due to a misreading of the rules.
- Students were not asked to reject duplicate guesses, although some did, probably copying the functionality from the *Hangman* game given with the assignment. While this was not wrong, it spoiled the game play when duplicate guesses were rejected, but with no message to the player to let them know what was happening in the game.
- After each guess the next game board shown to the user would have too much information, making it hard for the player to pick out what was important. The most common form of this error, probably copied from the *Hangman* game, was to show the player all of their incorrect guesses, which was necessary for Hangman but potentially confusing in the *WordSleuth* context.

In addition to the above common errors with the information displayed to the player, some students did not show the player all of the information asked for in the rules (for example not telling the player the number of guesses they had left, or that they had reached their final guess).

Students were asked to display appropriate winning and losing messages to the player. Only a tiny number implemented this badly or not at all, the vast majority of students received full credit for this part of the game.

Moderately serious errors: Quite a number of students lost credit because their *WordSleuth* game had not implemented that when players reached their final possible guess, they should be informed that they had reached their final guess and shown all the characters that they had seen for the current word, rearranged into the order in which they appeared in the word. This lost those students quite a lot of credit.

Other, much less common errors relating to the display of character hints to the user, included failing to show the player a random character after each guess, or showing the player the same random character after each guess.

Major errors: Very few students made the error of testing two `Strings` for equality with `"=="` instead of `String.equals()`, but those who did make this mistake could not make much progress with implementing *WordSleuth* rules since the game could not recognise a win.

Recursion: A few students used recursion to implement the game playing loop. Using recursion without carefully analysing its memory requirements

is not a good idea, as recursion can be memory heavy and lead to the class ending with a `StackOverflowError`. None of the very few *WordSleuth* classes implemented with recursion ended with this error, but most had other effects that compromised the game playing experience. In one case, when the game ended, the player would see the message that they had won or lost for as many times as the guesses they had made, as each guess called up a new copy of the method with the game playing loop, and each method ended with a message to the user. Another effect of using recursion was that the base case to end the recursion was poorly chosen, for example the method only ended when the player had used all their guesses. Hence when the player guessed correctly with guesses still in hand, they were congratulated on their win and invited to guess again, and again. Both of these effects of using recursion could easily have been found with testing.

Duplicate guesses: Excluding duplicate guesses was an unnecessary addition to the game play that could compromise the game playing experience if players were not informed that their guess was a duplicate, and was not being counted as a guess, and that this was why they were not being shown another random character from the word. A more serious effect was when the player was shown an extra random char when making a duplicate guess, but the random character was not counted towards the total number of random characters shown, because the duplicate guess was not included in the guess total. This means the player would see too many characters from the word, and that in the worst case too many duplicate guesses would cause an infinite loop in the method to find the next random character from the word. This would be because the loop to find a character not already shown to the player would reject every randomly generated character index number as the index of a character already shown to the user.

Repeated characters in the final guess `String`: Reordering the characters shown to the user into a `String` with the characters in the order they appeared in the word was challenging, and those students who managed to implement this successfully are congratulated. A very small minority did not attempt to implement this, simply, at the final guess stage, showing the player the characters again but just in the order shown to the player with no attempt at reordering. A larger minority implemented reordering the `String` but with logical errors where the `String` contained the same character more than once.

In some cases, where the word contained repeated characters, when more than one of these characters had been shown to the player during the current game (e.g. *i* and *i* from 'configuration'), the final guess `String` would display only one of the repeated characters. In the most common repeated character error, the final guess `String` would contain ***all*** repetitions of the character, whether or not every repetition of the character had been shown to the user during the game. That is, provided that at least one of the repeated characters from the word had been shown to the user during the game, the final guess `String` would contain all repetitions of the character, usually in the correct word order.

Game-playing loop errors: A minority of students had difficulty with the logic of their game-playing loop. Some wrote loops that could not recognise when the player had won or lost, so the game would end when the player had used all their guesses, without the player being informed of their win/lose status. The most common logical error was that the number of guesses was unlimited – the game would continue until the user guessed correctly.

Other errors were less serious but still compromised the game play:

- The game could not recognise when the user has guessed correctly on their first go.

- The game could not detect a win at the final guess stage.
- The game could only detect a win at the very first guess and/or at the final guess stage, but not at any intermediate stages.

Initialisation errors: The *Hangman* game was run through the *WordGamesArcade* by making a *Hangman* object, that was used for all games of *Hangman* played in the same session. The *WordGamesArcade* class had an instance variable of type *Hangman*, called *hangman*. Note that this instance variable was not initialised in the *WordGamesArcade* constructor, and so would take the default value for a reference variable of `null`. The *playHangman()* method would only initialise the *hangman* variable if it found that the variable was `null`. If the variable was not `null`, then the existing *hangman* variable was used for all subsequent games played in the same session.

```
private void playHangman() {
   if (hangman == null) {
      hangman = new Hangman(inputStream, output);
   }

   output.println("Starting Hangman...");
   hangman.play();
}
```

The *Hangman* class had a method called *initialiseGameState()* that set all the class's instance variables back to their starting values. In this way values from previous games would not compromise the game play of the current game.

Many students copied this implementation with their *WordSleuth* class, running all games in a particular session with the same object. These students usually wrote an *initialiseGameState()* method in their *WordSleuth* class, but often forgot to reinitialise one or more of their variables. This could have various effects, depending on which one of their instance variables they forgot to initialise, but the most usual error was that too many letters were included in the final guess `String`. This would be because an `ArrayList` held the index numbers of characters shown to the player. As the `ArrayList` was not reinitialised, subsequent games simply added to it. This would mean the player would see a misleading final guess `String`, but it could also mean that, as more indexes were added to the `ArrayList` by subsequent games, the class would go into an infinite loop as it could not find the next random character to display to the player, since it could not find an index that was not already in the `List`. Alternatively, an out of bounds error might be generated if the `List` contained an index larger than the size of the current word.

Another possible error was that some students had logic to prevent the same index number being added to the `List` more than once. This could mean that in subsequent games the class could not display certain characters because their index was already in the array.

A less common issue with the final guess `String` was that a `String` was used to hold the random characters shown to the player, with new characters added in word order. Hence this `String` was used at the final guess stage to show all the letters picked in order to the player. Since it was not initialised, in subsequent games more and more characters would be added, and the player would see a final guess `String` that did not make sense to them, and could even contain more characters than the word they were being asked to guess.

The examiners found many and varied issues with the failure to initialise instance variables for subsequent games, only two of them were common:

- The `boolean` variable that became true when the player won, was not reinitialised such that if the player won their first game, all subsequent games would think that the player had won before they had even entered a guess.
- A subsequent game could end with an `IndexOutOfBoundsException`, because the `int` variable used to count the number of characters from the word shown to the player, was not initialised in the *initialiseGameState()* method. In subsequent games the variable started with a positive value, and, as it was incremented with each guess, might become so large that it was bigger than the size of the `ArrayList` holding randomised characters from the current word. Hence, when it was used to get the next random character there could be an exception.

There were other errors caused by not initialising all instance variables in a dedicated method, but instead setting them to a new value at some point in the game playing loop. This could lead to such things as showing the player wrong information at the start of the game, as variables used to display that information (such as the number of guesses) were showing information from the previous game as they had not yet been updated. In one case a variable that recorded a win was reinitialised if the player won, but not if the player lost.

Problems caused by not initialising instance variables is another example of poor testing. Clearly students were only running their *WordSleuth* class once, and assuming that if it worked as it should then it would continue to do so in subsequent games. The examiners ran the *WordSleuth* class through the *WordGamesArcade* several times, finding many and varied initialisation errors.

Question 4

Students were asked to prevent the player from seeing duplicate random characters. A duplicate character is defined as being a character that is shown to the user at least one more time than it appears in the word. The question noted that words may have more than one copy of the same letter. This would mean that provided that the index of the character shown to the player was distinct, then it was within the rules to show repeated characters to players, that is to show to the player some or all of the characters that were repeated in the word.

Most students made a good attempt at this question. The most common errors were:

- An attempt at the question, but nevertheless duplicate random characters were still possible.
- The first letter of the word (given at the start of the game) could also be given as a subsequent random character
- No letter would be shown to the user more than once, however many times it appeared in the word.

The last of these errors listed above means that some students implemented the call to prevent duplicates being shown to the user by preventing the user from seeing the same letter twice in all cases, including where there was more than one copy of the letter in the word.

Finally, there was one error caused by not initialising all instance variables ready for the next game in the session. If the `ArrayList` with indexes of random characters shown to the user was not initialised, then it would save its values from the previous games when the next game started. This could mean that in the next game the method to choose a random character from

the word would stop picking further characters, as the size of the `ArrayList` would now be equal to the number of characters the player should be shown. After this point the player would be shown again the last generated random character at each subsequent guess, so the player might see duplicate characters.

Question 5

Students were asked to add playing the *WordSleuth* game as item 2 on the *WordGamesArcade* menu. This question was answered correctly by most students. Very few errors were seen, although one seen more than once was that the option to quit had not been preserved. On the *WordGamesArcade* class as given to students, choice 1 on the menu was Hangman, and choice 2 was to quit. Most students changed the option to quit to option 3 (occasionally to option 0, instead) and then added playing *WordSleuth* as option 2. A tiny minority added the option to play *WordSleuth* as item 2 on the menu, overwriting the option to quit. Hence the player could choose to play *Hangman* or play *WordSleuth*, but could not choose to quit.

Question 6

Students were asked to make sure that their answers to questions 2, 3, 4 and 5 were readable, following advice given with the assignment. Most students showed a reasonable grasp of readability issues, with only a few losing marks due to:

- Using variables with names that tell the reader nothing about their function, for example, *x*, *y* and *z*.
- Writing long methods with repetitive code, that could be shortened by moving the repeated statements into a method, for the first method to invoke.
- The game playing loop obscured its intent, making it hard to read and follow the logic of the game.

Part B: AnimatedGUI

In Part B, students were given a Java class, *AnimatedGUI*. The class had a GUI that displayed a square and a circle. Students were asked to animate the shapes. The square should move back and forward across the top of the draw panel, while the circle should grow until it reached the edge of the frame and then shrink back to its starting size. The animations should continue indefinitely. Students were also asked to add buttons to the draw panel so that the user could stop and start the animations as many times as they wished.

Question 1

a. Part B was attempted well by most students. There were a few errors, the most common being those students who ignored the instruction in part (a) to write inner classes to implement `ActionListener`. Students were asked to use a separate inner class to listen to each of the two `JButtons`. A minority of students ignored this instruction and instead copied the direct implementation of the `ActionListener` interface given with the assignment. An even smaller minority used anonymous classes to implement `ActionListener`, severely reducing their submission's readability. A less common error was to write one inner class to listen to both buttons.

b. Most students who attempted part (b), making the animation of each of the two shapes independent of each other, did so successfully. Only one mistake was seen, using logic to control starting and stopping the animations that included a variable shared by both shapes, hence stopping or starting one could affect the other.

c. Similar to part (b), very few errors were seen in answers to part (c). A tiny minority of students compromised their otherwise successful attempt by not including `drawPanel.repaint();` in the inner classes listening to the buttons. This meant that if the user stopped both animations they would not restart.

Question 2

a. Part (a) was answered successfully by all those attempting it.

b. Part (b) had some errors, although very few. The only common error was limiting the number of repetitions of the circle animation as it grew and shrunk.

Question 3

Students were asked to apply readability rules. Those students who had written just one inner class to listen to both buttons, or who used anonymous classes to implement `ActionListener`, lost marks for readability. Students writing one inner class tended to give their class generic names such as *AnimationListener,* which only gave very general information about the purpose of the class. A small number of students also lost marks for giving their inner classes names such as *ButtonListener1* and *ButtonListener2*, or *ActionListener1* and *ActionListener2*, since these names hamper readability by giving no information about the purpose of implementing `ActionListener`.