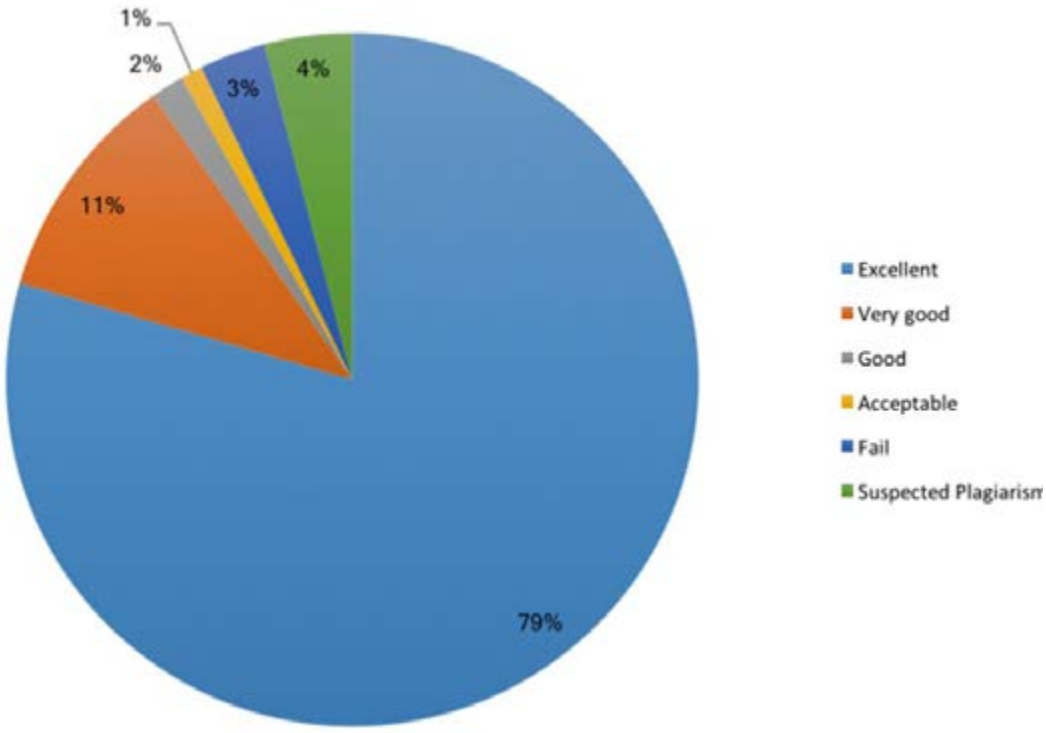
Coursework commentary 2017–2018

CO2220 Graphical object-oriented and internet programming in Java

Coursework assignment 2

General remarks

On the whole this coursework assignment was attempted well, with the majority of students gaining excellent marks. One exception to this was Question 1 of Part B, which was generally answered quite poorly. The chart below shows the mark distribution.



Model answers

The following files are provided with this commentary:

Part A

The following were as given out with the assignment:

- *Admissions.java*
- *FeeType.java*
- *Gender.java*
- *OfferStatus.java*
- *QualificationsType.java*
- *student_admissions_file.ser*
- *student_admissions_file.txt*

The following contain answers to the assignment:

- *ProspectiveStudent.java* (with added constructor)
- *AdmissionsFileManager.java* (with completed methods and an added constructor)

Part B

The following were as given out with the assignment:

- *Anagram.java*
- *AnagramClient.java*
- *AnagramClientGUI.java*
- *AnagramServer.java*
- *AnagramServerEngine.java*
- *History.java*
- *Permutation.java*
- *ServerUtils.java*
- *words.small*

The following contain answers to the assignment:

- *AnagramFactory.java*
- *PermutationFactory.java*
- *Dictionary.java*

Comments on specific questions

Part A

Students were given the following files:

`Enum` files

- *FeeType.java*
- *Gender.java*
- *OfferStatus.java*
- *QualificationsType.java*

Data files

- *student_admissions_file.ser*
- *student_admissions_file.txt*

Java classes

- *Admissions.java*
- *AdmissionsFileManager.java*
- *ProspectiveStudent.java*

The *FeeType*, *Gender*, *OfferStatus* and *QualificationsType* were `Enum` classes used in the constructor of the *ProspectiveStudent* class. The *AdmissionsFileManager* class contained some static methods, and some skeleton static methods, used in the *Admissions* class, to process a list of prospective students. The *student_admissions_file.ser* contained a serialized list of *ProspectiveStudent* objects, and the *student_admissions_file.txt* contained data with which to construct *ProspectiveStudent* objects.

Students were asked to write a constructor for the *ProspectiveStudent* class, to write an 'appropriate' constructor for the *AdmissionsFileManager* class and to complete the following methods in the *AdmissionsFileManager* class:

- *parseProspectiveStudent(ArrayList<String>)*

- *write(ArrayList<ProspectiveStudent>, String)*
- *deserialize(String)*
- *serialize(ArrayList<ProspectiveStudent>, String)*

Question 1

Students were asked to write a constructor for the *ProspectiveStudent* class. The class had one constructor that set default values for all of its fields. Students were asked to write a second constructor to allow the user to set the values held in the fields. This meant writing a constructor that took 10 arguments in order to initialise *ProspectiveStudent's* 10 instance variables. Most of the errors seen with answers to this question were from students who were careless in some way with initialising the fields:

- the *gender* variable will always be null because it has not been assigned in the constructor so the field gets its default value.
- the *name* field has not been initialised in the constructor, hence every object has the name null.
- the *email* field has not been initialised in the constructor, and so defaults to null.
- the statement this.offer = offer; assigns the *offer* field to itself, meaning that it will take the default value, null, in every case. This is because the variable in the parameter list is called 'Offer', and since Java is case sensitive 'Offer' and 'offer' are two different things as far as the compiler is concerned.
- fields have their assignment statements the wrong way round (*e.g.* gender = this.gender), hence all fields set to themselves and hence take their default values.
- because of a spelling mistake or typo in the parameter list (*courseAppliendTo* instead of *courseAppliedTo*) the *courseAppliedTo* field is being set to itself, hence it is defaulting to zero.

All of these errors could have been found and corrected with better testing.

Question 2

In the *AdmissionsFileManager* class, the *read()* method had a loop that read 10 lines at a time from the text file given with the assignment, storing each line as a separate entry in an `ArrayList`. The `ArrayList` was then sent to a second method, *parseProspectiveStudent(),* to be parsed to a *ProspectiveStudent* object, and if this was successful the new object was added to a second `ArrayList`, which the method returned once the loop had ended. Students were asked to complete the *parseProspectiveStudent()* method.

Many good attempts were seen at this question. Some students had problems with getting the order of parsing correct, and some with parsing the *dateOfBirth* field. In all instances where students encountered problems with the *dateOfBirth* field, it was because they were not using the *parseDate()* method included with the class. Students either wrote their own parsing method, which did not work as it should, or incorporated parsing the *dateOfBirth* field into the *parseProspectiveStudent()* method, which again did not work. Some students taking this approach did correctly parse the *dateOfBirth* field, but it is odd that these students ignored the working *parseDate()* method included with the class.

Other errors again seemed to be caused by simple carelessness, such as the *email* and *phone* fields being assigned each other's values, by being parsed in the wrong order, or causing a `NumberFormatException` by attempting to parse the *id* field to an `int`, when what is being parsed is actually intended for the *name* field, which is of type `String`.

Question 3

Students were asked to complete the *write()* method, such that it would save the data in its `ArrayList<ProspectiveStudent>` parameter into a text file with a name given by the user. The objects in the `ArrayList` should be written into the text file using *ProspectiveStudent*'s *toString()* method. Each element from the `ArrayList` should be separated in the text file with a blank line. Students were asked to handle any potential exceptions, and show the user an appropriate error message if an exception was thrown. The tasks of the *write()* method were described in a comment immediately above the method in the file given to students.

Blank line errors: A major source of confusion here was the request to separate distinct objects in the text file with a blank line. The comment written directly above the *write()* method detailed what tasks the method should perform. Some students misunderstood the sentence "Each element from the ArrayList is separated in the text file with a blank line". Each element in the `ArrayList` is a *ProspectiveStudent* object, so the blank line is used to separate the data of one *ProspectiveStudent* object from the data of the next *ProspectiveStudent* object. Some students thought that the data from each field of every object saved in the `ArrayList` should be followed by a blank line. Students making this mistake did not use *ProspectiveStudent's toString()* method to write data into the file, as asked, since this did not separate fields with a blank line. Perhaps if these students had reflected more on the contradiction between being asked to use *ProspectiveStudent's toString()* method to write each object into the text file, and inserting a blank line after each field, they would have realised their error. Students making this error gave themselves a lot of unnecessary extra work, and lost marks for not following the assignment instructions.

Other Q3 errors:

- In contrast to students doing much unnecessary work to insert too many blank lines into the text file, some students lost credit by not inserting any blank lines.
- A few students did not close the text file, which meant that all the data written into the file was lost. Again, this is an error that could have been found and corrected with better testing.
- For some reason, a few students confused saving data to a text file with serialization, and either wrote methods to serialize, or attempted to use an ObjectOutputStream to write to a text file. This meant that the file contained some non-text data, hence the *read()* method (see Question 2 above) would not be able to read the file successfully and make objects with the data it contained.

Exception handling was normally correct. A few students wrote statements such as `bw.write(student.toString());` where *student* was a *ProspectiveStudent* object. This is not wrong, but students should know that there is no need to explicitly call an object's *toString()* method.

Question 4

Students were asked to complete the *deserialize()* method. A comment above the method described its tasks as follows:

```
/*returns an ArrayList<ProspectiveStudent>, populated
by deserializing from the file student_admissions_file.
ser. Handles the ClassNotFoundException in its own catch
block, handles any other potential exceptions. If an
exception is thrown the user will see an appropriate
error message, and the method will return null.*/
```

Major errors here were with the loop to read from the *student_admissions_file.ser* file, plus not implementing a separate catch block for the `ClassNotFoundException`.

Loop errors: If the *serialize()* method serialized one object, an `ArrayList<ProspectiveStudent>`, then the *deserialize()* method only needed to deserialize that one object, and hence did not need a loop. If the *serialize()* method read each *ProspectiveStudent* object from the `ArrayList<ProspectiveStudent>` parameter it was given, and serialized each *ProspectiveStudent* object individually into the file, then the *deserialize()* method would need a loop in order to read each *ProspectiveStudent* object, cast it and add it to an `ArrayList`.

Confusion between serialize() and deserialize(): There were three errors made by students whose *serialize()* and *deserialize()* methods would not be able to work together to successfully serialize and deserialize:

- Writing a *serialize()* method that had a loop that serialized each individual *ProspectiveStudent* object from the ArrayList, and writing a *deserialize()* method that had no loop and deserialized one *ProspectiveStudent* object, storing it in an ArrayList.
- Writing a *serialize()* method that serialized the method's ArrayList parameter, but then writing a *deserialize()* method that attempted to deserialize one *ProspectiveStudent* object at a time in a loop. Having written their own *serialize()* method, these students must have known that the serialized file contained only one object, an ArrayList<ProspectiveStudent>.
- Writing a *serialize()* method that serialized each *ProspectiveStudent* object, and a *deserialize()* method that attempted to deserialize one object, an ArrayList<ProspectiveStudent>.

Difficulties in writing a successful loop in the deserialize() method: Some students wrote methods that serialized each *ProspectiveStudent* object in a loop, and then correctly wrote a *deserialize()* method with a loop to deserialize each *ProspectiveStudent* object, cast it and store it in an `ArrayList`. The problem these students had was in writing a loop that ended successfully when it detected the end of the file. Some tried to detect the end of the serialized file by using statements suitable for text files, such as `while(in.read() != -1)`.

Detecting the end of a serialized file is difficult, but can be done by using the `EOFException` and a `while(true)` loop, as follows:

```
public static ArrayList<ProspectiveStudent> deserialize(String s){
   ArrayList<ProspectiveStudent> ps = new
                        ArrayList<ProspectiveStudent>();
   try{
      ObjectInputStream in = new ObjectInputStream(new
                        FileInputStream(s));
      try{
         while(true) ps.add((ProspectiveStudent)in.readObject());
      }
      catch (EOFException e){
         System.out.println("Deserialization successful ");
         in.close();
      }
   }//end of outer try
```

```
   catch (ClassNotFoundException e){
      System.err.println("Class not found " + e);
      return null;
   }
   catch (Exception e) {
      System.err.println("Error reading from file " +
                         serialFilename + ". " + e);
      return null;
   }
   return ps;
}
```

Exception handling errors: Exception handling was done correctly by many students, with only a few errors seen. However, many students did not give the `ClassNotFoundException` its own catch block, as requested. The `ClassNotFoundException` is thrown if the JVM cannot find the class the method is attempting to cast the deserialized object to. It makes sense to handle it separately and send the user a helpful message about what has gone wrong.

Question 5

The following comment, written directly above the skeleton of the *serialize()* method in the file given to students, described the tasks of the method:

```
/*takes an ArrayList<ProspectiveStudent> and serializes
it into the file student_admissions_file.ser. Handles any
potential exceptions. If an exception is thrown the user
will see an appropriate error message. */
```

Most students wrote a correct method, with a small minority having problems when attempting to write a loop to serialize one *ProspectiveStudent* object at a time. Writing a correct loop to serialize does take some thought, although some basic errors were seen, such as closing the `ObjectOutputStream` in the loop, when it clearly should be closed only once the loop had completed. Most students avoided loop problems by serializing the `ArrayList` as one object. Another very small minority wrote methods that would not work, as the students had not understood the rather basic fact that the `ArrayList` parameter given to the method (or its contents) should be serialized. These students made an `ArrayList` in their methods, and then serialized the empty `ArrayList`.

Question 6

Students were asked to add an appropriate constructor to the *AdmissionsFileManager* class. Since the class has only static methods, it would not be appropriate to make an object of it. Classes that contain static methods for doing some useful tasks with other objects, are known as static utility classes. Sometimes they are given a private, empty constructor. Since the constructor is private it cannot be accessed from outside the class, therefore it is not possible to make an object of the class.

Java has four access modifiers for classes, constructors, methods and variables: *private, protected, public,* and the *default (package)*. The *default* access modifier is given to items that do not have an access modifier, and means that the item can only be accessed within its containing package. *Public* means, as you would expect, it is open to all, *private* means an item can only be accessed from within its class, and *protected* means that the item can be accessed within its containing package, and in any sub classes. Hence *private* is the

right access modifier to prevent instantiation of the *AdmissionsFileManager* class. This question could be answered simply, and correctly, with `private AdmissionsFileManager(){}`

The coursework assignment instructions included reading on static utility classes (see *Head First Java* pages 275-278). These four pages clearly explained the concept of a *private* constructor for static utility classes. Despite this there were quite a number of students who wrote a *public* constructor, while others wrote a *protected* one. Some students correctly gave their constructor *private* access, but then added statements to their constructor, such as `super();` or a statement to initialise the *filename* variable. In fact, a few students wrote very elaborate constructors, representing considerable wasted effort that could have been avoided if these students had read the four pages of *Head First Java* on static utility classes.

Conclusion

Many students made a very good attempt at Part A. Some students could have improved their work with better testing.

Part B

Students were given the following files:

- *Anagram.java*
- *AnagramClient.java*
- *AnagramClientGUI.java*
- *AnagramFactory.java*
- *AnagramServer.java*
- *AnagramServerEngine.java*
- *Dictionary.java*
- *History.java*
- *Permutation.java*
- *PermutationFactory.java*
- *ServerUtils.java*
- *words.small*

Students were asked to run the *AnagramClientGUI* class, which would display a simple GUI with a `JButton`, a scrollable `JTextArea` and a `JTextField`. Clicking the `JButton`, which displayed the text 'Connect', would bring up a message similar to the following, on the `JTextArea`:

```
Welcome IP address '127.0.0.1' to the Anagram Server.
Available commands:
ANAGRAM <word>    show all anagrams of the word <word>
PERMUTE <word>    show all permutations of the word <word>
SHOW HELP         show this help
```

Permutations of a word are all possible rearrangements of the letters in the word. Anagrams, in this case, were all permutations of a word that were words in their own right. This excludes anagrams of a word that are two words, such as *pierrot* and *papa* from *appropriate*.

Permutations of a word could produce the same arrangement of letters more than once, if the word contained repeated letters, for example all possible permutations of TAT, are:

TA**T**

T**T**A

AT**T**

A**T**T

TAT

TTA

In the above the second 'T' is in bold. As far as theory is concerned, a 3 letter word has 6 permutations (incuding itself). Hence as far as theory is concerned AT**T** and A**T**T are distinct permutations of TAT. However the system given to students stored all permutations in a `TreeSet`, since this data structure does not allow repeated items. This gives a saving on storage space, important as the number of permutations of a word increases enormously with the number of letters in the word. This is because the formula to count the number of *n* length permutations of an *n* length word reduces to the factorial of the number of letters in the word, where the factorial of a number means multiplying a number by every positive integer smaller than itself (not including zero), and is written *n!*. Factorial numbers start small, but they soon increase hugely, even for numbers less than 20, as the following shows:

0!		=	1
1!		=	1
2!	= 1 x 2	=	2
3!	= 1 x 2 x 3	=	6
4!	= 1 x 2 x 3 x 4	=	24
5!	= 1 x 2 x 3 x 4 x 5	=	120
9!		=	362,880
12!		=	479,001,600
15!		=	1,307,674,368,000

Hence as far as the system given to students is concerned, the permutations of 'TAT' are `att tta`, as the system will not return duplicate permutations, and also does not include the word itself, although the theory does. In addition the system put every word given to it into lower case before permuting it.

Students were asked three questions about the system.

Question 1

The first question asked students to explain why the *Dictionary*, *PermutationFactory* and *AnagramFactory* classes all implemented the Singleton design pattern. Students were expected to do some research to answer this question, but were given the following link to start with: https://en.wikipedia.org/wiki/Singleton_pattern

Students were asked to write a comment in each of the classes implementing the Singleton design pattern, to explain why it was being used, and how. Very few good answers were seen. Disappointingly most students wrote one comment and copied it into each of the three files, changing only the class name. Many repeated things they had read about the singleton design pattern without explaining their meaning, such as describing the Singleton design pattern as encompassing *lazy implementation*. Many students wrote only that the purpose of the implementation was to provide a *single point of access*, without explaining why a single point of access was needed. Others wrote that the design pattern had been implemented as only one instance was needed of the class. Such comments did not explain: (1) how the Singleton design pattern was implemented in that particular class; and (2) why a *single point of access* or 'only one instance of the class' was needed.

The Dictionary class comment: The Singleton design pattern limits the number of instances of a class, normally to just one. The *Dictionary* class reads from a text file containing more than 45,000 words, and stores the words in an `ArrayList`. We make the constructor private so it cannot be accessed from outside the class. In order to access the constructor classes must use the public *getInstance()* method, this first checks for an existing instance of the object, and if there is one returns that. Otherwise it uses the constructor to make a new object. Hence the Singleton design pattern is implemented in the *Dictionary* class by having a private constructor, a static variable of type *Dictionary* called *instance* that the constructor initialises, and a public *getInstance()* method. The private constructor means that only the *Dictionary* class itself can make an instance of the class.

The purpose of implementing the Singleton design pattern in the *Dictionary* class is to make the best use of system resources. If we kept on making *Dictionary* objects, that would be heavy on storage since the file is fairly big at 400 KiB, as well as unnecessary since we only need access to one copy of the file with the words in it. Because it makes sense to have only one dictionary, and since I/O operations are expensive, we limit ourselves to one reading of the dictionary file. This saving on storage also gives us the flexibility of using even larger dictionary files without compromising the memory. Careful husbanding of system resources could allow the server to be extended to multiple clients without quickly running out of memory.

The PermutationFactory class comment: Similarly to the Dictionary class, the *PermutationFactory* class has a private constructor, and a public *getInstance()* method. The private constructor means that only the class itself can make an instance of the class. The *getInstance()* method checks to see if there is already an instance of the class and either returns the existing instantiation, or makes and returns a new *PermutationFactory* object if one does not already exist. An instance of the class is a `HashMap` variable used as a cache for *Permutation* objects. However, while we do not add words to the dictionary, we may want to add new *Permutation* objects to the *PermutationFactory* instance, and since the work of permuting a word can potentially take much time and memory, we do not want to permute the same word twice.

The *createPermutation()* method of the *PermutationFactory* class will first check to see if the cache already contains a *Permutation* object for the word that it has been asked to permute, if so the *Permutation* object is returned, if not the *Permutation* object is made and then returned. Every new *Permutation* object is added to the cache. Hence the Singleton design pattern is implemented twice, once to prevent more than one instantiation of the *PermutationFactory* object, and again to prevent a second instantiation of *Permutation* objects so that once a word is permuted, it is not done again. The reason for implementing the design pattern is to make careful use of system resources. Preventing duplication of effort saves memory, minimises storage space, and potentially allows results to be shared among multiple clients.

The AnagramFactory class comment: For the *AnagramFactory*, similarly to the other two classes, we only make a new instance if we do not have a stored instance. This is implemented using a private constructor and a *getInstance()* method. We do this because it can take a long time to search for all possible anagrams of a word, since firstly we permute the word, and then we filter the permutations using the dictionary, to leave only those permutations that are words.

An *AnagramFactory* object consists of a `HashMap`, a *Dictionary* object, and a *PermutationFactory* object. Classes wanting to find anagrams of a word will invoke the class's *createAnagram()* method, which returns an *Anagram* object.

An *Anagram* object consists of a `String` word, and a `List` of anagrams of the word. The `HashMap` variable, called *cache*, contains *Anagram* objects, and is first checked to see if the word already has an associated *Anagram* object. If so, this object is returned. If not a new *Anagram* object is made and added to the *cache* before being returned.

In a sense the Singleton pattern is implemented three times in this method. Firstly because the *getInstance()* method and the private constructor together enforce that there can be only one instance of the object. Secondly because the *createAnagram()* method will enforce that there can only be one *Anagram* object associated with a particular word. Thirdly because the *createAnagram()* method uses the *PermutationFactory's createPermutation()* method to return a permutation of the word, and this method will only permute a word that has not already been permuted.

Again, the reason for the Singleton design pattern implementation is to conserve system resources in terms of storage and memory. Having only one *AnagramFactory* object saves storage, and only making an *Anagram* object where one does not already exists means that less permuting is done, which is very expensive in terms of memory, and also means that the *AnagramFactory's filter()* method, which is very slow, is only run when necessary.

Question 2

On the system as given to students, anagrams and permutations were presented to the user on one line. Students were asked to change this so that each `String` had its own line, for example instead of seeing `att tta` output as the permutations of 'tat', the output would be:

```
att
tta
```

Some students wrote quite complicated implementations of this which often did not work properly, when all that was needed was to look at the *formatWords()* method in the *AnagramServerEngine* class, as this method formatted the anagrams and permutations for display on the GUI. In the method the statement `output.append(" ");` adds spacing between the `Strings` to be displayed. The model answer has replaced this statement with `output.append(System.lineSeparator());` in order to add a platform independent new line after every `String`.

Question 3

Students were told that finding anagrams of a 10-letter word took more than 4 minutes on the machine used by the examiner, and were asked to trace the source of this delay, and to change the data structure used in one of the classes *Dictionary*, *PermutationFactory* or *AnagramFactory* such that finding anagrams was faster. Students were also asked to explain the change that they had made with a comment in the appropriate class.

Students were expected to find that the bottleneck was in the *filter()* method of the *AnagramFactory* class, and to explain that this was because it was using the `ArrayList.contains()` method to search the dictionary to find if a permutation of a word was contained in it. Since a word could have millions, or billions of `String` permutations, and since the `contains()` method was invoked on every `String` permutation, the speed of the method was very important. The commentary in the `ArrayList` API notes that:

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. [...] All of the other operations run in linear time (roughly speaking).

Hence the `ArrayList.contains()` method runs in linear time, that is, the running time increases linearly with the size of the input. Other `Collection` objects have faster methods, in particular the commentary in the `TreeSet` API notes that:

This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).

Log(n) time is considerably faster than linear time, since it means that the time taken increases in proportion to the log of the size of the input, and the log function grows very slowly. Hence storing the dictionary in a `TreeSet` would mean that the *filter()* method would be using the `Treeset.contains()` method, which, since the method is invoked once for every permutation of a word, is a potentially huge time saving.

A few students misidentified the source of the delay, and changed the data structure holding the anagrams from an `ArrayList` to a `TreeSet`, or changed the `HashMap` instance variable in the *PermutationFactory* class to a `TreeSet`, claiming a time improvement that testing did not find. Some students made the right correction, but wrote in their comment that the reason for the improvement in search time was that a `TreeSet` removes repetition from the permutations. This showed a lack of understanding of how the system worked, since repetitions were removed in the *permute()* method of the *PermutationFactory* class by adding permutations to a `TreeSet` which does not permit repeated elements. Some students made complicated changes to more than one class, despite guidance in the question that changing a data structure in *one* class was all that was needed. Sometimes these complicated attempts worked, and sometimes they did not. A few students even implemented changes that made the system fail with `java.lang.OutOfMemoryError: Java heap space` when searching for anagrams of 12-letter words.

Some students correctly identified the *filter()* method in the *AnagramFactory* class as the source of the delay, but instead of looking for a data structure with a faster *contains()* method, they wrote their own *filter()* methods. Sometimes these methods worked, and sometimes they made the search slower. More than one student changed the *filter()* method so that it returned all permutations, which was not an improvement.

All of these mistakes could have been found and addressed with testing.

Conclusion

While some students made a good or very good attempt at Part B, most students did not put enough thought into answering Question 1.