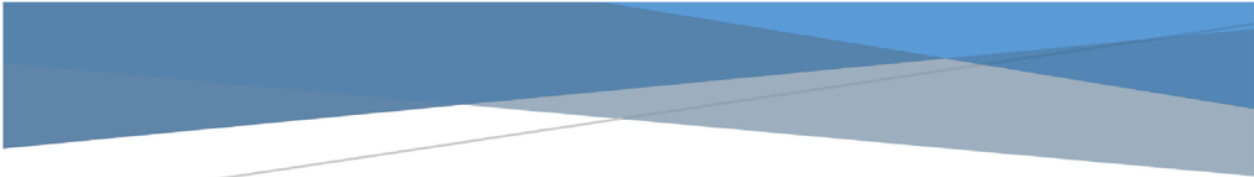


Adam Davies - Programming and Testing Assignment - Submission

by Adam Peter Davies

FILE	2784_ADAM_PETER_DAVIES_ADAM_DAVIES_-_PROGRAMMING_AND_TESTING_ASSIGNMENT_-_SUBMISSION_159704_78228099.DOCX (3.77M)		
TIME SUBMITTED	11-JAN-2019 08:32PM (UTC+0000)	WORD COUNT	22106
SUBMISSION ID	98852979	CHARACTER COUNT	130381



(ACCA7000) PROGRAMMING AND SOFTWARE TESTING

Java Programming and Testing Part 1 and 2

Abstract

This document is split into two reports. The first report provides an evaluation of the software testing techniques and process at the DVLA. It also includes the solution to a programming task which was to manipulate two sets of numbers in various ways. It was found that the DVLA uses process such as Continuous Integration which automates Unit testing and acceptance testing. The DVLA also uses a large amount of performance testing. All testing techniques used benefit the DVLA and produce no drawbacks due to the use of extensive team collaboration methodologies. The programming task is also completed with the solution provided within the report.

The second report provides a discussion on the importance of high cohesion, loose coupling, data hiding and information hiding within software engineering. Also included is the software solution to a set programming task of creating a console based connect four game. It was found that all four principles are desirable qualities of producing quality software. They can lead to more manageable and reusable code which can save bug fixing and make required changes easy. The complete solution to the task is also provided in java.

Author: Adam Davies

P147908

PART ONE

Table of Contents

Table of Contents	I
Table of Figures	III
Table of Tables	IV
1. Introduction	1
2. Background Information	1
2.1. Testing Overview	1
2.1.1. Black Box Testing	1
2.1.2. White Box Testing	2
2.2. Types of Testing	2
2.2.1. Unit Testing	2
2.2.2. Integration Testing	2
2.2.3. System Testing	3
2.2.4. Acceptance Testing	3
2.2.5. Regression Testing	3
2.2.6. Levels of Testing	3
2.2. Test Driven Development	4
2.3. Continuous Integration	4
3. Testing at the DVLA	5
3.1. Testing Methodology	5
3.2. Phases	5
3.3. Testing Process	5
3.4. Evaluation	7
4. Programming Task	8
4.1. Task Brief	8
4.2. Problem Solving Methodology	8
5. Conclusion	10
References	11

Bibliography	13
Appendix	14
Problem Solving Mind Map.....	14
Evidence Using Version Control	15
Final Code	19
J-Unit Tests	25

Table of Figures

Figure 1: Software Testing Levels Pyramid 3

Figure 2: TDD Flowchart..... 4

Table of Tables

Table 1: Problem Breakdown..... 9

1. Introduction

The purpose of this report is to present an evaluation of the software testing techniques and processes which are employed at the *DVLA*. It also demonstrates the solving of a set programming task which is to manipulate two sets of numbers entered by the user in various ways. The report is structured with firstly a brief background to software testing and various testing techniques. The main body includes the evaluation of techniques which are employed at the *DVLA* including *Test Driven Development (TDD)*, *Continuous Integration (CI)* and performance testing. Evidence of the problem-solving techniques used to solve the set programming task is also included. All results and concepts are concluded at the end of the report.

2. Background Information

This section of the report overviews the background literature which includes an overview of various testing methods, techniques and processes.

2.1. Testing Overview

Software testing is the process of evaluating produced software for specification functionality [1]. There are two main methods of software testing these being *black box* and *white box* testing.

2.1.1. Black Box Testing

Black box testing is a testing method which involves examining the specification of designed software and not the internal workings [2]. Testers do not know the technical structure and formal design of the software and test the system from a customer's view point.

There are various techniques which are associated with *black box* testing. One of these techniques is *equivalence partitioning* which involves dividing input values into invalid and valid partitions [3]. *Boundary value analysis* is another technique which involves testing the boundaries of conditions by setting input values within the proximity of set boundaries [4]. *Cause-Effect graphing* is a way of generating test cases by identifying inputs and the effects of this input on the output [3].

2.1.2. White Box Testing

White box (clear box) testing is a testing method which evaluates a software's functionality by examining the internal workings [2]. The tester generally is not concerned with the user interface.

There are two main *white box* testing techniques, being *static* and *structural*. *Static white box* testing involves only the source code before it is executed or completed. The main aim of this technique is to check the software in accordance to the design, functional specification and coding standards [5].

Structural white box testing inspects the internal design, structure and data flow of the system [5]. *Coverage* testing is performed in *structural* testing which involves testing loop structures and conditional branches to full effect [3].

2.2. Types of Testing

There are various testing types which are classified as *black box* testing or *white box* testing. The main tests are *unit*, *integration*, *system* and *acceptance* testing.

2.2.1. Unit Testing

Unit testing is a level of software testing which tests individual components and modules of a system. *Unit tests* are usually written by the developer as the software is being developed [6]. Testers are not concerned with testing the entirety of a system and concentrate on smaller modules which make up a system. *Unit tests* usually make use of *stubs*, *mocks* and *fakes* [7].

2.2.2. Integration Testing

Integration testing is where various individual system modules are combined and tested together. This testing exposes errors and bugs in the interaction between two or more units [8].

There are two main types of *integration testing*, *Bottom Up* and *Top Down*. *Top Down integration testing* is a method where top level units are tested first. *Stubs* are used to provide a simulation of units which may not have already been developed. *Bottom Up* is the opposite of this where bottom level units are tested first [2] [9].

2.2.3. System Testing

System testing is performed on completed software which has gone through *integration testing*. It looks at the whole system and how each sub-system interacts with each other [10].

2.2.4. Acceptance Testing

Acceptance testing is performed to check that the software functions based upon the user requirements. Usually this is completed using a *black box testing* method where user needs, and requirements are viewed for acceptability [2].

2.2.5. Regression Testing

Regression testing is performed when changes are made to existing software. *Regression tests* are used to test whether bugs or errors have been introduced when changes have been applied to existing software [11].

2.2.6. Levels of Testing

All types of software tests are distinguished at different levels and are shown via the testing pyramid (*figure 1*).

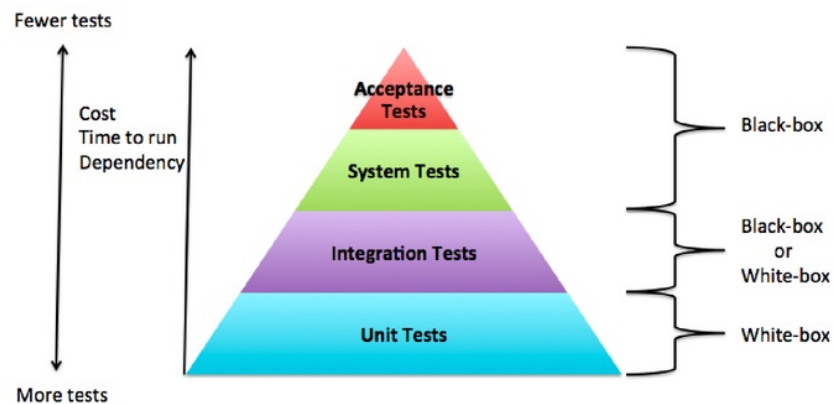


Figure 1: Software Testing Levels Pyramid [12]

As you move up through the testing pyramid any tests which fail are costlier to rectify and take more time to run and fix.

2.2. Test Driven Development

Test Driven Development (TDD) is a development methodology where development is driven by testing. *TDD* relies on short iteration cycles where tests are firstly developed before any code is developed. After a test is made to fail, code is developed with continuous refactoring of the test, design and software to satisfy changing requirements [13]. Once the code passes, another test is written, and the process is repeated. Below is a flowchart depicting this operation (*figure 2*).

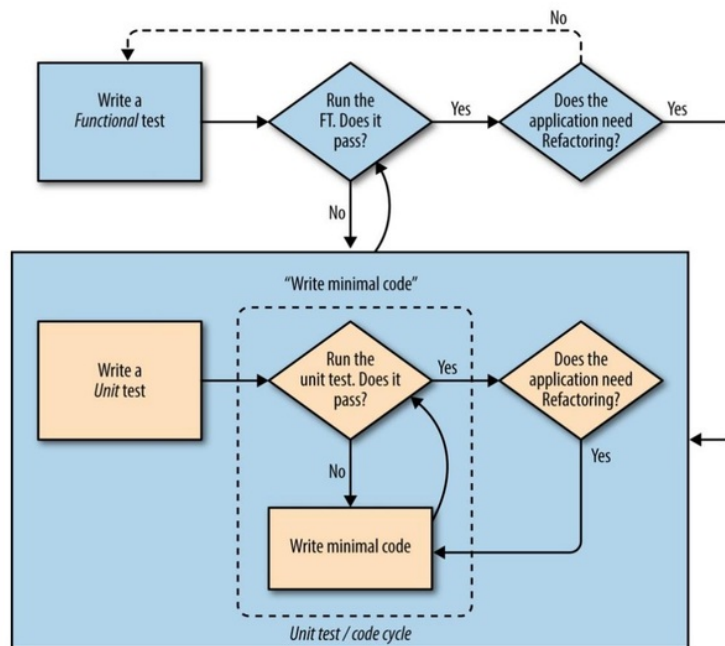


Figure 2: TDD Flowchart [14]

2.3. Continuous Integration

Continuous Integration (CI) is the process of merging developers work several times within a day to a shared repository [6]. It is also the process of automating the build and testing of developed code. Once a developer commits code to a set repository it triggers the latest version of code and runs a build, test and validate before merging to the master can commence [15]. There are many software tools available to manage *CI* including *Jenkins*, *Team City* and *Drone*.

3. Testing at the DVLA

This section overviews the testing processes and techniques at the DVLA. These processes are described and then evaluated at the end of the section.

3.1. Testing Methodology

The DVLA uses *Acceptance Test Driven Development* as its overlying methodology. *ATDD* is a variation on traditional *TDD*. It incorporates a *Three Amigos* practice which involves team members that have different perspectives to come together. *Acceptance* tests are written together before any development takes place [13] [16]. Usually at the DVLA a business analyst, developer and tester are involved in this process. It also follows the same structure and operation as traditional *TDD* with refactoring taking place to make changes to code.

3.2. Phases

The DVLA uses distinct phases for testing and development. There are three main development stages which are *Alpha*, *Beta* and *Live*. The *alpha* phase consists of building a prototype and performing various tests on this prototype. Collaboration between team members is key in this stage with *acceptance* tests being generated within this stage using the three amigos practice [17].

When the system is ready for the *beta* phase the software is deployed in either a private or public state. By releasing the software as *beta*, it allows members of the public (*public beta*) or team (*private beta*) to perform user and acceptance testing on the software [18]. Once all requirements have been satisfied from the product backlog the software is released and is set to live. This is where the majority of performance analysis is carried out by performing relative performance tests.

3.3. Testing Process

Acceptance tests are written before any code is developed. The tools used to develop acceptance tests at the DVLA varies between teams. Commonly used tools include *Cucumber* framework with *Gerkin* language and variations of *Ruby* and *RSpec*.

The DVLA employ a rule in which for every new feature developed, it is mandatory that the developer makes a new branch on the repository [19]. Once a commit is made the DVLA uses

Continuous Integration tools either *Drone* (for newer software) or *Jenkins* (for older software) to detect the commit and push request which runs automated *unit* tests and acceptance tests within the build pipeline.

Unit tests are written by the developer which use the *J-Unit* framework. These tests are written before code is developed following a *TDD* methodology. *Unit* tests are also run in the *CI* build pipeline allowing them to be automated.

When the developed software passes a pull is requested. Before a merge is made the code must be peer reviewed by a tester and a developer. Once this is performed the software is merged with the master which allows a *regression* test to be performed. At the *DVLA* a pre-developed *regression* pack is run to check if any errors are introduced from the merge. Regression tests at the *DVLA* are usually performed using *Selenium*.

The *DVLA* uses a variety of environments where software lives. They use various environments for different pieces of software (which are at different stages). Environments include *local*, *AWS*, *INT (Integration) Layer* (formal performance testing), *PROD* (Live environment) and *PRE-PROD* (Ops support) [19].

Once software is developed it is moved into *INT* where performance testing is carried out. There are a variety of performance tests which are carried out at the *DVLA* including *load* testing, *stress* testing and *soak* testing. *Load* testing is used to ensure it can cope with peak volumes of traffic. This is performed regularly at the *DVLA* due to continuously changing software e.g. changes in the way credit cards are verified may change the amount of resources the software uses [19].

Another performance test is *stress* testing which is pushing the software to deliberately fail, testing its threshold. *Soak* testing is also performed to check for any memory leaks. The *DVLA* do this by following the *Spring framework*. For all performance tests *Apache JMeter* is used.

User Interface (UI) testing is also performed using *Selenium*. *UI* is also tested via *betas* which sometimes incorporates public user testing. The *DVLA* uses a *UX lab* in which they invite selected numbers of the public to test new software within *beta* phase. They use feedback from this to improve *UI* and overall software casting requirements into user stories [20].

Once software has completed and passed performance testing, it is made live and moved into the *PROD* environment. A smoke test is used here to show that the software is running as intended.

3.4. Evaluation

With the DVLA adopting an ATDD methodology it provides greater communication between tester and developer, this allows the tester to write tests as the developer writes code as collaboration between the two is increased. ATDD is adaptive and therefore, any changes in requirements are easily enforced within tests due to the refactoring of the process [21]. This fits in well with the DVLA's AGILE development methodology.

One problem the DVLA have is that due to adaptive nature of ATDD it can be difficult to implement on legacy systems. Early stages of refactoring can also require a lot of code to be changed which can lead to increased time adapting tests for the code [13].

Acceptance testing can result in increased collaboration between customer and developer/tester. This is beneficial as it provides a means where the developer/tester totally understands what the customer wants [2] [22]. If therefore acceptance tests pass, then the developer/tester can be confident that it is adequate for the customer. Performing acceptance tests can also increase the quality of the software product, as the tests are written around the user stories which usually directly come from the customer requirements.

Unit tests also provide similar benefits and can increase the quality of code as particular functions of code are tested independently. This means that confidence is increased when software is merged as bugs can be identified early on. By eliminating bugs early on the cost of fixes are decreased as per testing pyramid [6].

The problems with unit and acceptance testing is the time taken for test implementation and the resources required. Collaboration is also extensively needed between customer, testers, developers and business analysts to fully contribute to producing successful test criteria [23]. Without valid criteria testing becomes irrelevant as it may not pass customers requirements. Although this may be an issue, the DVLA promote AGILE and team collaboration methodologies so the risk of this effect is reduced.

By the DVLA automating tests through CI it allows integration bugs to be detected early on, therefore the cost of removing these bugs is potentially reduced [15] [24]. With continuous commits comes constant feedback, therefore code can be improved, and errors can be eliminated early.

The DVLA uses a large amount of performance testing. This is beneficial as it ensures features are operating to the correct standard, thus keeping customers happy [24]. This is important for key web services deployed by the DVLA e.g. pay your tax online. If this website goes down a day before a user requires to tax their vehicle, then their car is illegal to drive. This highlights

why the DVLA use extensive performance testing. Although performance testing brings many benefits it does however take up a lot of resources and requires a specific team to perform required tests [11].

4. Programming Task

This section of the report outlines the process of solving the given programming task. Evidence of the methodology used to produce a solution is provided. It also includes evidence of *unit testing* and using version control *git*.

4.1. Task Brief

The task provided was to allow the user to enter two sets of numbers named as A and B respectively, with the user unable to enter duplicate numbers. A menu is required to be displayed allowing the user to select from the following functionality:

- Display the numbers in both sets.
- Determine the maximum and minimum values of each set.
- Calculate the average of each set.
- Determine if there is an overlap between both sets.
- Determine if there is an intersection.
- Determine the set difference.
- Allow the user to quit the program.

4.2. Problem Solving Methodology

Firstly, the problem was broken down into smaller more manageable problems. The problems which emerged from the task requirements are outlined below (**table 1**).

Problem	Priority
How can the user enter two sets of numbers? 	2
How are the numbers going to be displayed?	3
How is the menu going to be displayed?	1
How is the menu going to be repeated?	5
How are the minimum and maximum values going to be calculated?	7
How is the average going to be calculated?	8
How is an overlap going to be found? 	9
How is an intersection going to be determined and displayed?	10
How is the set difference going to be determined and displayed?	11
How is the user going to quit the program?	6
How is the program going to check for duplicates?	4

Table 1: Problem Breakdown

These problems were prioritised in terms of difficulty which is denoted by the number at the side, with one being the most manageable. Once the problem was broken down mind mapping was performed to introduce ideas on how each of the problems could be solved. This mind map is shown in the appendix.

After mind mapping was complete the solution to each problem (following priority) was attempted with firstly a test being written. By using *git* and *bit bucket* for version control, branches could be made to try and add more difficult features, such as the intersection and set difference. This allowed a form of experimentation to be carried out without it affecting any of the master code. Once the code was tested to an acceptable standard it was merged with the master code. The full program and evidence of *git* is shown in the relevant sections of the appendix.

5. Conclusion

Testing is regarded highly at the *DVLA* with the main software methodology used being tailored for testing. *ATDD* is used with a three amigos approach used to design software and tests based around user stories. The *DVLA* uses a phase-based system which classifies the stage in which developed software is at. At these stages various testing processes are employed. The overall process consists of *acceptance* testing, *unit* testing and using *continuous integration* to ensure that any committed code is tested thoroughly. This happening before a pull request is made in which a peer review of committed code is performed. All this is completed before code is merged to a master branch.

By using *ATDD* and along with *continuous integration* it allows bugs to be eliminated early. With the *DVLA* using large amounts of performance testing it ensures customers get the best possible services. Overall it is believed that the *DVLA* adopts a thorough testing process which integrates well to their *AGILE* development methodologies and frameworks such as *SCRUM*.

The solution to the given programming task was completed with all functionality operating in accordance to the given brief. It was found that using problem solving techniques offered such as mapping out and breaking problems down beneficial to solving the task. Instead of writing software initially, writing problems in a mind map form was considered most beneficial from the techniques used. Overall the solution is believed to satisfy all requirements outlined in the brief with various problem-solving techniques found beneficial in solving the problem.

References

- [1] M. Steve, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed., Microsoft Press, 2004.
- [2] D. Srinivasan and R. Gopalswamy, *Software Testing: Principles and Practice*, Pearson Education India, 2006.
- [3] N. Srinivas and D. Jagruthi, "Black Box and White Box Testing Techniques - A Literature Review," *International Journal of Embedded Systems and Applications*, vol. 2, no. 2, pp. 29-50, June 2012.
- [4] T. Murnane, K. Reed and R. Hall, "On the Learnability of Two Representations of Equivalence and Boundary Value Analysis," in *Software Engineering Conference*, 2007.
- [5] F. Saglietti, N. Oster and F. Pinte, "White and grey-box verification and validation approaches for safety- and security-critical software systems," *Information Security Technical Report*, vol. 13, no. 1, pp. 10-16, 2008.
- [6] L. Johannes and F. Peter, *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufmann, 2003.
- [7] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *IEEE Computer Society*, 2007.
- [8] Delamaro, M. Eduardo, J. C. Maidonado and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE transactions on software engineering*, vol. 27, no. 3, pp. 228-247, 2001.
- [9] softwaretestingfundamentals.com, "Integration Testing," [Online]. Available: <http://softwaretestingfundamentals.com/integration-testing/>. [Accessed 11 December 2018].
- [10] J. A. Whittaker, "What is software testing? And why is it so hard?," *IEEE Software*, vol. 17, no. 1, pp. 70-79, 2000.
- [11] G. Dorothy, V. V. Erik and E. Isabel, *Foundations of Software Testing: ISTQB Certification*, illustrated, revised ed., Cengage Learning EMEA, 2008.
- [12] softwaretesting.natashaleite.com, "Testing Phases," 8 May 2014. [Online]. Available: <http://softwaretesting.natashaleite.com/wp/?p=54>. [Accessed 11 December 2018].
- [13] B. Kent, *Test-driven Development: By Example*, illustrated, reprint ed., Addison-Wesley Professional, 2003.
- [14] J. P. Harry, "Chapter 4. What Are We Doing with All These Tests?," in *Test-Driven Development with Python*, 1st ed., O'Reilly Media, 2014.
- [15] R. Sander, *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*, Packt Publishing Ltd, 2017.

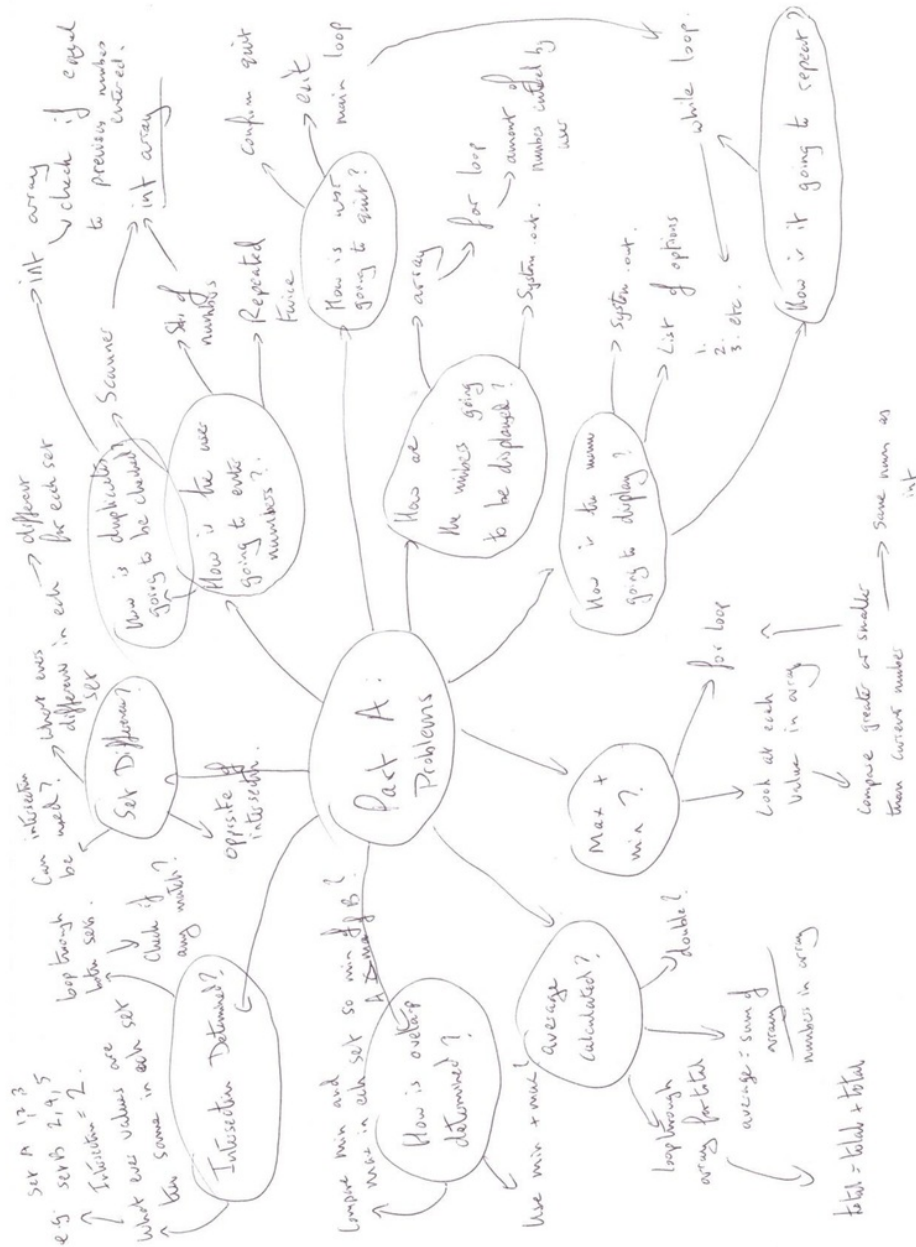
- [16] L. Roberto, "A successful application of a Test-Driven Development strategy in the industrial environment," *Empirical Software Engineering*, vol. 19, pp. 753-773, 2013.
- [17] gov.uk, "How the alpha phase works," 2017. [Online]. Available: <https://www.gov.uk/service-manual/agile-delivery/how-the-alpha-phase-works>. [Accessed 18 December 2018].
- [18] gov.uk, "How the beta phase works," 2017. [Online]. Available: <https://www.gov.uk/service-manual/agile-delivery/how-the-beta-phase-works>. [Accessed 18 December 2018].
- [19] C. Ashley, Interviewee, *Questions on Testing at the DVLA*. [Interview]. 17 December 2018.
- [20] T. Jamie, "User testing at DVLA," 3 May 2013. [Online]. Available: <https://gds.blog.gov.uk/2013/05/03/user-testing-at-dvla/>. [Accessed 18 December 2018].
- [21] H. Thomas, G. David and S. Tom, *Test-Driven Development: A J2EE Example*, Apress, 2007.
- [22] M. G. Limaye, *Software Testing*, Tata McGraw-Hill Education.
- [23] W. Laurie, K. Gunnar and N. Nachiappan, "On the Effectiveness of Unit Test Automation at Microsoft," 2009.
- [24] D. Elfriede, R. Jeff and P. John, *Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley Professional, 1999.

Bibliography

1. G. L. Milind, Software Testing, Tata McGraw-Hill Education, 2009.
2. G. Shekhar and S. Rahul, Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5, Apress, 2017.
3. D. R. Kuhn, R. W. Dolores and M. G. Albert, "Software fault interactions and implications for software testing," *IEEE transactions on software engineering*, vol. 30, no. 6, pp. 418-421, 2004.
4. T. Courtney, "What Is the Intersection of Two Sets?," 26 March 2018. [Online]. Available: <https://www.thoughtco.com/intersection-in-set-theory-3126587>. [Accessed 10 December 2018].
5. T. Courtney, "What Is the Difference of Two Sets in Set Theory?," 13 June 2018. [Online]. Available: <https://www.thoughtco.com/difference-of-two-sets-3126580>. [Accessed 12 December 2018].
6. A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *IEEE Computer Society*, 2007.

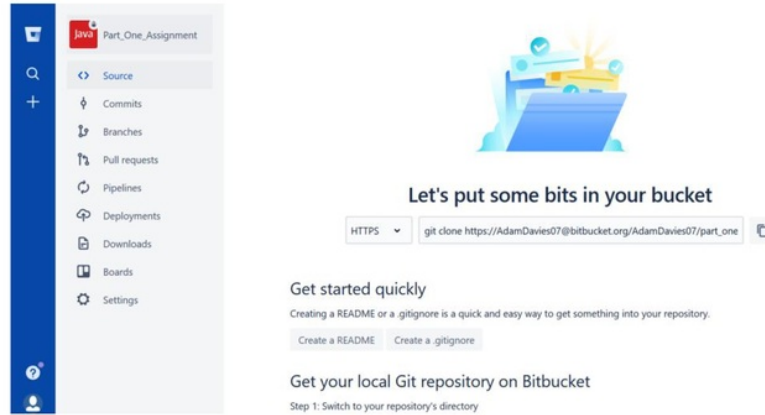
Appendix

Problem Solving Mind Map



Evidence Using Version Control

Bit bucket was used to create a blank repository.



A local folder was created and cloned the remote repository into local destination.

```
User-Admin@DESKTOP-VCV1P9C MINGW64 ~  
$ cd ..  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Users  
$ cd ..  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c  
$ cd Programming\ and\ Software\ Testing/  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing  
$ dir  
Part\ One  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing  
$ cd Part\ One/  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part  
One  
$ git clone  
https://AdamDavies07@bitbucket.org/AdamDavies07/part_one_assignment.git  
Cloning into 'part_one_assignment'...  
warning: You appear to have cloned an empty repository.  
  
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part  
One  
$
```

A project was created and an initial commit was made to check if files were added.

```
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git add *

User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git commit -m 'Initial Commit (Added Source Code File)'
[master (root-commit) 0c2e94b] Initial Commit (Added Source Code File)
 2 files changed, 21 insertions(+)
 create mode 100644 part_one_assignment.iml
 create mode 100644 src/Part_One_Assignment/Main.java
```

A push was used for the initial commit to bit bucket to test if the source is at the remote repository.

```
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git push
https://AdamDavies07@bitbucket.org/AdamDavies07/part_one_assignment.git
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 730 bytes | 365.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://bitbucket.org/AdamDavies07/part_one_assignment.git
 * [new branch]      master -> master
```

Adam Davies / Part_One_Assignment

Commits

All branches ▾

Find commits

Author	Commit	Message	Date	Builds
 Adam Davies	0c2e94b	Initial Commit (Added Source Code File)	11 minutes ago	

Code was written and committed. Creating a new branch for the more complex features with the first being overlap.

```
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git branch overlapFunction

User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git checkout overlapFunction
Switched to branch 'overlapFunction'
```

```
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (overlapFunction)
$ git commit -m 'Added test for overlap function'
[overlapFunction f897ea5] Added test for overlap function
 4 files changed, 40 insertions(+), 6 deletions(-)
 rewrite out/test/part_one_assignment/Part_One_Assignment/Tests.class (67%)
```

```

User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (overlapFunction)
$ git checkout master
Switched to branch 'master'
Your branch is based on 'origin/master', but the upstream is gone.
(use "git branch --unset-upstream" to fixup)

User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git merge overlapFunction
Updating 2aec513..b997a1c
Fast-forward
 Tests/Part_One_Assignment/Tests.java      | 24 ++++++
 .../Part_One_Assignment/Main.class        | Bin 6063 -> 6515 bytes
 src/Part_One_Assignment/Main.java         | 65 ++++++
-
3 files changed, 82 insertions(+), 7 deletions(-)

```

All commits are shown.

```

User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
One/part_one_assignment (master)
$ git log
commit 34ab0bacb6ee5c1b5cf0bc63cbf3d15ceeb48bed (HEAD -> master)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 21:31:35 2018 +0000

    Refined set difference tests and functions for more effecincy

commit f453fdfa947c93a4d88f98a62701c510212454d6
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 20:27:21 2018 +0000

    Added comments to all functions

commit ee1be33e80d93aade82c9fcdcf4991540a3247af1
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 15:59:47 2018 +0000

    Added set difference function with counter test

commit e5b1550537263b5361d7d8b4a64bc862742a35a1 (intersectionFunction)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 15:02:26 2018 +0000

    Added intersection function and tested

commit f46667bd92752a92234a83a67c5ee60d58f81f53
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 14:18:57 2018 +0000

    Added intersection function and tested

commit 5320f063ad973b8968f96421f4f16593b93d74f9
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 14:10:59 2018 +0000

    Added test for intersection function with blank method

commit b997a1cedfdcf012e75727e3ce429ebb20b74195 (overlapFunction)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 13:30:37 2018 +0000

    Overlap function and tests added

commit f897ea5348bf7f58642066beb2ac6c21bbdaebf4
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 12:55:24 2018 +0000

    Added test for overlap function

commit 2aec513b806e34fa2efcc8f4aedc385926640126
Author: AdamDavies07 <adamdavies07@btinternet.com>

```

Date: Wed Dec 19 12:28:30 2018 +0000

Added test and function for calculating the average

commit 83a1d8476e5490dfe4489ae194244035a94b4f66
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 11:58:42 2018 +0000

Added test and function for getting min and max for each set

commit 4cfc01a9a6905eab3071b7a7b69ab398803ca757
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 11:32:52 2018 +0000

Added display numbers in each set function

commit 8a9065dd479445e3d6cc0e28c184cfd6b3dd82c3
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 10:47:31 2018 +0000

Added menu loop and quit function with unit tests

commit 464d88899ca0770f728d978e73c0d2b3a76d9dd1
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 10:14:29 2018 +0000

Added unit tests and functions for entering number sets and duplicate number check

commit 5d93dccb438172b599a4315625b8a2dde9b2c1ba
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 08:58:03 2018 +0000

Added enter amount of numbers in each set with JUnit tests

commit f777d49dcc10e585b59a118a2b862a91f1bdf6bd
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 16:21:32 2018 +0000

Added display menu function

commit 0c2e94b63519effc197e643c70c37d56712bdb67
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 15:34:18 2018 +0000

Initial Commit (Added Source Code File)

(END)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 10:14:29 2018 +0000

Added unit tests and functions for entering number sets and duplicate number check

commit 5d93dccb438172b599a4315625b8a2dde9b2c1ba
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 08:58:03 2018 +0000

Added enter amount of numbers in each set with JUnit tests

commit f777d49dcc10e585b59a118a2b862a91f1bdf6bd
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 16:21:32 2018 +0000

Added display menu function

commit 0c2e94b63519effc197e643c70c37d56712bdb67
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 15:34:18 2018 +0000

Initial Commit (Added Source Code File)

(END)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 10:14:29 2018 +0000

Added unit tests and functions for entering number sets and duplicate number check

commit 5d93dccb438172b599a4315625b8a2dde9b2c1ba
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Wed Dec 19 08:58:03 2018 +0000

Added enter amount of numbers in each set with JUnit tests

commit f777d49dcc10e585b59a118a2b862a91f1bdf6bd
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 16:21:32 2018 +0000

Added display menu function

commit 0c2e94b63519effc197e643c70c37d56712bdb67
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Tue Dec 18 15:34:18 2018 +0000

Initial Commit (Added Source Code File)

Final Code

```
package Part_One_Assignment;

import java.util.Scanner;

public class Main
{
    static Scanner stdin = new Scanner(System.in); //Define stdin for keyboard scanner input

    public static void main(String[] args)
    {
        int option, amountA, amountB, i, j; //declare integer variables
        int[] numbersA = new int[0];
        int[] numbersB = new int[0];
        boolean valid;
        char listLabel;

        amountA = enterAmountOfNumbers(listLabel = 'A', valid = false); //Call enterAmountOfNumbers function and
        return the value into amountA
        amountB = enterAmountOfNumbers(listLabel = 'B', valid = false); //Call enterAmountOfNumbers function and
        return the value into amountB
        numbersA = enterNumbers(i = 0, amountA, listLabel = 'A', numbersA, valid = false); //call enterNumbers
        function and return value into array for set A
        numbersB = enterNumbers(i = 0, amountB, listLabel = 'B', numbersB, valid = false); //call enterNumbers
        function and return value into array for set B
        operations(option = 0, i, amountA, amountB, numbersA, numbersB, listLabel, valid, j = 0);
        System.out.printf("\r\nSystem Shut Down!\r\n");
        //System.out.printf("\r\nOption selected is %d" , option); //display option for testing
    }

    /**
     * Function to display the users menu and asking to enter an option
     * @param option int value which is passed in and initially set to 0
     * @return option int which is the option from the menu entered by the user
     */
    static int displayMenu(int option, boolean valid)
    {
        System.out.printf("\r\n---Menu---\r\n");
        System.out.printf("\r\n1. Display Both Sets of Numbers");
        System.out.printf("\r\n2. Display Max and Min Values");
        System.out.printf("\r\n3. Calculate the Average");
        System.out.printf("\r\n4. Do Set A and Set B Overlap?");
        System.out.printf("\r\n5. Intersection of Sets A and B");
        System.out.printf("\r\n6. Set Difference of A and B");
        System.out.printf("\r\n7. Quit");
        System.out.printf("\r\nPlease select an option: ");

        while(valid == false) //While loop to keep asking user for option until a valid int is entered
        {
            try //Try catch used to check if the option is a valid integer data type
            {
                option = Integer.parseInt(stdin.next()); //Ask user to enter an option from the menu
                valid = true; //Set valid to true to exit loop if no error is caught
            }
            catch (NumberFormatException nfe)
            {
                System.out.printf("\r\nInvalid value type entered!\r\nPlease try again:");
                valid = false; //Set valid to false if an error is found re do the loop
            }
        }

        return option; //return integer option
        //return option = 3;
        //Tested using Unit Test testMenuReturn
    }

    /**
     * Function asking the user to enter the length of a set of numbers
     * @param listLabel holds the list character to display purposes
     * @param valid boolean which is used as a check variable in the try catch loop
     * @return int amount which is the amount of numbers the user requires in the set
     */
    static int enterAmountOfNumbers(char listLabel, boolean valid)
    {
        int amount = 0; //declare and initialise set amount to 0
    }
```

```

while(valid == false) //While loop to keep asking user to enter number until a valid amount is entered
{
    try //Try catch used to check if the amount is a valid integer data type
    {
        System.out.printf("\nPlease enter the amount of numbers in list %c: ", listLabel);
        amount = Integer.parseInt(stdin.next());
        valid = true; //Set valid to true to exit loop if no error is caught
    }
    catch (NumberFormatException nfe)
    {
        System.out.printf("\nInvalid value type entered!\nPlease try again:");
        valid = false; //Set valid to false if an error is found re do the loop
    }
}

return amount; //return the amount of numbers in the set
}

/**
 * Function to allows the user to enter numbers into the set array
 * @param i int counter used as a for loop counter
 * @param amount int the amount of numbers the user required from enterAmount function
 * @param listLabel char which is used as a display variable
 * @param numbers int array which will hold the set of numbers entered by the user
 * @param valid boolean which is a checker for valid input entry
 * @return int[] numbers which holds the numbers entered by the user
 */
static int[] enterNumbers(int i, int amount, char listLabel, int[] numbers, boolean valid)
{
    int number = 0; //int number which is number entered by user set initially to 0
    boolean duplicate = true; //duplicate boolean used as a duplication checker

    numbers = new int[amount]; //make numbers array the length of amount of numbers required by user

    System.out.printf("\n---Enter Numbers for List %c---\n", listLabel);

    for(i = 0; i < amount; i++) //for loop to loop around the amount of numbers the user requires in the set
    {
        while((valid == false) && (duplicate == true)) //loop while the users number is not valid and is a
duplicate
        {
            try //try catch used to check for valid int input from user
            {
                System.out.printf("\nPlease enter number %d: ", i + 1);
                number = Integer.parseInt(stdin.next());
                valid = true; //set valid as true
            }
            catch (NumberFormatException nfe)
            {
                System.out.println("\nInvalid value type entered!\nPlease try again:");
                valid = false; //set valid as false if no valid int is entered
            }

            if(valid == true) //if the value is valid
            {
                duplicate = checkForDuplicate(i, amount, numbers, number); //call duplicate function to check for
duplicates return value in duplicate

                if(duplicate == true) //if duplicate is true then ...
                {
                    System.out.printf("\nNumber has already been entered!");
                    System.out.printf("\nPlease try again:\n");
                    valid = false; //make valid false
                }
                else //if no duplicates
                {
                    numbers[i] = number; //enter numbers into array
                }
            }
        }

        //numbers[i] = i + 1; //Line for testing
        duplicate = true; //reset duplicate and valid booleans for each iteration
        valid = false;
    }

    return numbers; //return number set array
}

/**
 * Function used to check for duplicate numbers entered
 * @param i int used as for loop counter
 * @param amount int which holds the amount of numbers the user requires in the set
 * @param numbers int array which is the numbers within the set passed through
 * @param number int number is the number which is entered by the user
 * @return return boolean duplicate which is true if a duplicate is found
 */
static boolean checkForDuplicate(int i, int amount, int[] numbers, int number)
{
    for(i = 0; i < amount; i++) //for loop used to loop around each number in the array..
    {
        if(numbers[i] == number) //check if the number is already in numbers array
        {
            return true; //make duplicate true if duplicate found
        }
    }

    return false; //return duplicate as false if no duplicate found
}

/**
 * Function which will hold the switch case for all operations and call all operator functions
 * @param option int which is the option from the menu selected by the user
 * @param i int used as a for loop counter
 * @param amountA int which is the amount of numbers in list A
 * @param amountB int which is the amount of numbers in list B

```

```

    * @param numbersA int array holds all values entered in set A
    * @param numbersB int array holds all values entered in set B
    * @param listLabel char list label used for display
    * @param valid boolean which is to check if option is selected is valid
    * @param j is used as an alternative for loop counter
    */
    static void operations(int option, int i, int amountA, int amountB, int[] numbersA, int[] numbersB, char
listLabel, boolean valid, int j)
    {
        boolean quit = false; //set quit to false initially
        while(quit == false) //loop while quit is false
        {
            option = displayMenu(option, valid = false); //call display menu function and return the option selected
into option variable

            switch (option) //switch case used as conditional branch to run what the user enters in option
            {
                case 1: //display number sets
                    System.out.printf("\r\n---Display Sets A and B---\r\n");
                    displayNumberSet(i, amountA, numbersA, listLabel = 'A'); //call display number sets for both A
and B
                    displayNumberSet(i, amountB, numbersB, listLabel = 'B'); //by passing relevant arrays
                    break;

                case 2: //get min and max and display
                    System.out.printf("\r\n---Display Min and Max---\r\n");
                    displayMinAndMax(i, amountA, numbersA, listLabel = 'A'); //call display min and max functions
                    displayMinAndMax(i, amountB, numbersB, listLabel = 'B');
                    break;

                case 3: //calculate and display the average
                    System.out.printf("\r\n---Calculate Averages---\r\n");
                    displayAverage(i, amountA, numbersA, listLabel = 'A'); //call display average function
                    displayAverage(i, amountB, numbersB, listLabel = 'B');
                    break;

                case 4: //check for an overlap
                    System.out.printf("\r\n---Overlap Check---\r\n");
                    displayOverlap(i, amountA, amountB, numbersA, numbersB); //call display overlap function
                    break;

                case 5: //determine the intersection
                    System.out.printf("\r\n---Intersection---\r\n");
                    displayIntersection(i, j, amountA, amountB, numbersA, numbersB); //call display intersection
function
                    break;

                case 6: //determine the set difference
                    System.out.printf("\r\n---Set Difference---\r\n");
                    displaySetDifference(i, j, amountA, amountB, numbersA, numbersB); //call get set difference
function
                    break;

                case 7: //quit the program
                    System.out.printf("\r\n---Quit---\r\n");
                    quit = verifyQuit(quit, valid = false); //call quit verify function and return bool into quit
                    break;

                default: //incorrect value entered
                    System.out.printf("Incorrect value entered");
                    break;
            }
        }
    }

    /**
    * Function which is to verify and ask the user to quit
    * @param quit boolean quit which is true if the user wants to quit
    * @param valid boolean used to check if the quit answer is a valid option
    * @return quit boolean which is true if the user wants to quit and it is a valid entry
    */
    static boolean verifyQuit(boolean quit, boolean valid)
    {
        String quitAnswer; //string quitAnswer used to holds string of the answer entered

        while(valid == false) //while a invalid answer is entered loop
        {
            System.out.printf("\r\nAre you sure you want to quit(yes/no): ");
            quitAnswer = stdin.next();
            //Test Lines used
            //quitAnswer = "no";
            //quitAnswer = "yes";
            //quitAnswer = "Adam";

            if (quitAnswer.compareToIgnoreCase("yes") == 0) //if yes is entered make valid and quit true
            {
                valid = true;
                quit = true;
            }
            else if (quitAnswer.compareToIgnoreCase("no") == 0) //if no is entered make valid true and quit false
            {
                valid = true;
                quit = false;
            }
            else //if invalid option is entered make valid false to loop
            {
                System.out.printf("\r\nInvalid Option!");
                System.out.printf("\r\nPlease try again:\r\n");
                valid = false;
            }
        }

        return quit; //return boolean quit
    }
}

```

```

/**
 * Function to display the numbers in each set
 * @param i int used as for loop counter
 * @param amount int which holds the amount of numbers in a set
 * @param numbers int array which holds the numbers in the set
 * @param listLabel char which holds the list letter for display purposes
 */
static void displayNumberSet(int i, int amount, int[] numbers, char listLabel)
{
    System.out.printf("\r\n---Set %c---\r\n", listLabel);

    for(i = 0; i < amount; i++)    //for loop used to loop around and print each number in the array
    {
        if(i < amount - 1)    //if number is less than one less than the amount of numbers in set then...
        {
            System.out.printf(" %d ", numbers[i]);    //...print with comma
        }
        else
        {
            System.out.printf(" %d", numbers[i]);    //else print without meaning last number
        }
    }

    System.out.printf("\r\n");
}

/**
 * Function which gets the minimum number in the set passed through
 * @param i int used as a for loop counter
 * @param amount int amount used to hold the number of numbers required by the user
 * @param numbers int array which is the number set passed through
 * @param min int min which holds the minimum value
 * @return int min which holds min value within the set
 */
static int getMin(int i, int amount, int[] numbers, int min)
{
    for (i = 0; i < amount; i++)    //for loop used to check each position in the array against min set value
    {
        if(numbers[i] < min)    //if numbers position i less than the min ....
        {
            min = numbers[i];    //...then make min equal to that number in array
        }
    }

    return min;    //return the min number
}

/**
 * Function which gets the maximum number from the passed through set
 * @param i int used as a for loop counter
 * @param amount int amount used to hold the number of values in the set
 * @param numbers int array which holds the numbers in the set
 * @param max int which will hold the max value found in the set
 * @return int max which holds the max value found
 */
static int getMax(int i, int amount, int[] numbers, int max)
{
    for (i = 0; i < amount; i++)    //for loop used to check each position in the array against max set value
    {
        if(numbers[i] > max)    //if numbers position i greater than the max ....
        {
            max = numbers[i];    //...then make max equal to that number in array
        }
    }

    return max;    //return max value
}

/**
 * Function used to display the max and min values of the set
 * @param i int used as a for loop counter
 * @param amount int used to hold the number of values in a set
 * @param numbers int array which holds all numbers in the set
 * @param listLabel char which is used for displaying set letter
 */
static void displayMinAndMax(int i, int amount, int[] numbers, char listLabel)
{
    int min = numbers[0];    //make both max and min variables equal to first number in set arrays
    int max = numbers[0];

    min = getMin(i, amount, numbers, min);    //call min and max functions making return value min and max
    max = getMax(i, amount, numbers, max);

    System.out.printf("\r\n---Set %c---\r\n", listLabel);
    System.out.printf("\r\nMin: %d\r\nMax: %d\r\n", min, max);    //print min and max values
}

/**
 * Function to calculate the averages of each set passed through
 * @param i int used as for loop counter
 * @param amount int is the amount of numbers in the set
 * @param numbers int array which stores the numbers in each set
 * @param listLabel char used as a list pointer for display purposes
 * @param average double which holds the value of the average calculated
 * @return double average which is the average calculated
 */
static double calculateAverage(int i, int amount, int[] numbers, char listLabel, double average)
{
    int sum = 0;    //make int sum initialise to 0

    System.out.printf("\r\n\r\n---Set %c---", listLabel);

    for(i = 0; i < amount; i++)    //for loop used to add each value of the array to sum variable
    {
        sum = sum + numbers[i];
    }

```

```

    }

    average = (double)sum / amount; //average is = to the sum divided by amount of numbers in the set
    return average;
}

/**
 * Function to display the calculated average
 * @param i int used as a for loop counter in calculate averages
 * @param amount int holds the amount of numbers
 * @param numbers int array which holds the numbers in the set
 * @param listLabel char which holds the list character of the set
 */
static void displayAverage(int i, int amount, int[] numbers, char listLabel)
{
    double average; // declare average variable as a double

    average = calculateAverage(i, amount, numbers, listLabel, average = 0); //call calculateAverage and return the
value into average
    System.out.printf("\n\nThe average for set %c is: %.3f", listLabel, average); //display average
}

/**
 * Function which checks for a overlap between both sets
 * @param i int which is used as a for loop counter
 * @param amountA int amountA is to holds the amount of numbers within set A
 * @param amountB int amountB is to holds the amount of numbers within set B
 * @param numbersA int array which holds values in set A
 * @param numbersB int array which holds values in set B
 */
static boolean checkOverlap(int i, int amountA, int amountB, int[] numbersA, int[] numbersB)
{
    int minA = numbersA[i]; //make each min and max value as the first value in each test
    int maxA = numbersA[i];
    int minB = numbersB[i];
    int maxB = numbersB[i];

    minA = getMin(i, amountA, numbersA, minA); //call get min and max and store in relevant min and max variables
    maxA = getMax(i, amountA, numbersA, maxA);
    minB = getMin(i, amountB, numbersB, minB);
    maxB = getMax(i, amountB, numbersB, maxB);

    if((minA < maxB) && (minB < maxA)) //if min of set A is less than max of set B AND min of set B is less
than max of set A
    {
        //then an Overlap exists
        return true; //used for testing purposes
    }
    else
    {
        //else no overlap
        return false; //used for testing purposes
    }
}

/**
 * Function added to display overlap
 * @param i int used as for loop counter
 * @param amountA int used to hold the amount of numbers in set A
 * @param amountB int used to hold the amount of numbers in set B
 * @param numbersA int array holds the numbers in set A
 * @param numbersB int array holds the numbers in set B
 */
static void displayOverlap(int i, int amountA, int amountB, int[] numbersA, int[] numbersB)
{
    boolean overlap; //boolean for overlap

    overlap = checkOverlap(i, amountA, amountB, numbersA, numbersB); //call overlap function to get overlap

    if(overlap == true) //if overlap is true then display...
    {
        System.out.printf("\n\nOverlap Found!\n\n"); //then an Overlap exists
    }
    else
    {
        System.out.printf("\n\nNo Overlap Found!\n\n"); //else no overlap
    }
}

static int intersectionCounter(int i, int j, int amountA, int amountB, int[] numbersA, int[] numbersB, int
counter)
{
    for(i = 0; i < amountA; i++) //for loop used to check every value in loop and count how many values are
common
    {
        for(j = 0; j < amountB; j++)
        {
            if(numbersA[i] == numbersB[j])
            {
                counter = counter + 1; //add one to counter
            }
        }
    }

    return counter;
}

/**
 * Function for determining intersection
 * @param i int used as a for loop counter
 * @param j int used as a alternative for loop counter
 * @param amountA int used to hold the number of values in set A
 * @param amountB int used to hold the number of values in set B
 * @param numbersA int array used to hold set A values
 * @param numbersB int array used to hold set B values

```

```

    * @return intersection int array which holds all intersection values found
    */
    static int[] getIntersection(int i, int j, int amountA, int amountB, int[] numbersA, int[] numbersB)
    {
        int intersectionCounter = 0; //intersection counter used to count through intersect array
        int counter = 0; //counter used to keep count of how many intersection values are found so array can be
        //created to this size

        counter = intersectionCounter(i, j, amountA, amountB, numbersA, numbersB, counter);

        int[] intersection = new int[counter]; //make intersection array which is the length of this counter

        for(i = 0; i < amountA; i++) //nested for loop used to count through each set and compare position i of set
        //A to various pos in set B
        {
            for(j = 0; j < amountB; j++)
            {
                if(numbersA[i] == numbersB[j])
                {
                    intersection[intersectionCounter] = numbersA[i]; //make the intersection array position
                    //intersection counter to equal numbers set A
                    intersectionCounter = intersectionCounter + 1;
                }
            }
        }

        return intersection; //return intersection array
    }

    /**
     * Function to display the intersection of both sets
     * @param i int used as for loop counter
     * @param j int used as alternative for loop counter
     * @param amountA int used to hold the number of values in set A
     * @param amountB int used to hold the number of values in set B
     * @param numbersA int array used to hold set A values
     * @param numbersB int array used to hold set B values
     */
    static void displayIntersection(int i, int j, int amountA, int amountB, int[] numbersA, int[] numbersB)
    {
        int[] intersection;
        boolean overlap;
        boolean intersectionExists = false; // use boolean to check if a intersection exists

        intersection = getIntersection(i, j, amountA, amountB, numbersA, numbersB); //call intersection function and
        //return to intersection array

        for(i = 0; i < intersection.length; i++)
        {
            if(i < intersection.length - 1) //if number is less than one less than the amount of numbers in set
            //then...
            {
                System.out.printf(" %d ", intersection[i]); //...print with comma
            }
            else
            {
                System.out.printf(" %d", intersection[i]); //else print without meaning last number
            }

            intersectionExists = true; //make intersectionExists true if it has gone through for loop
        }

        if(intersectionExists == false) //if there is no intersection display not found
        {
            System.out.printf("No intersection found!\r\n");
        }
    }

    /**
     * Function which gets the set difference of each set
     * @param i int used as a for loop counter
     * @param j int used as a for loop counter
     * @param amount int used to hold the amount of numbers in a set
     * @param intersection int array holds the intersection values
     * @param numbers int array holds the values of the number set
     * @return return set difference array which holds set difference values in that set
     */
    static int[] getSetDifference(int i, int j, int amount, int[] intersection, int[] numbers)
    {
        int[] setDifference;
        int setDifferenceCounter, notEqualCounter;
        int setDiffFoundCounter = 0;

        setDifferenceCounter = amount - intersection.length;
        setDifference = new int[setDifferenceCounter];

        for(i = 0; i < amount; i++) //for loop to check each position of the set A array to see if it is in the
        //intersection result
        {
            notEqualCounter = 0; //reset counter to 0

            for(j = 0; j < intersection.length; j++) //loop for the length of the intersection array checking it
            //against numbers set A pos A
            {
                if (numbers[i] != intersection[j]) //if there not equal then add one to counter
                {
                    notEqualCounter = notEqualCounter + 1;
                }
            }

            if(notEqualCounter == intersection.length) //if the counter reaches the length of the intersection array
            //there means there is no numbers in there
            {
                setDifference[setDiffFoundCounter] = numbers[i]; //therefore make set difference equal to numbers A
                setDiffFoundCounter = setDiffFoundCounter + 1;
            }
        }
    }

```

```

    }
}

return setDifference;
}

/**
 * Function to display the set difference
 * @param i int used as a for loop counter
 * @param j int j used as a alternative for loop counter
 * @param amountA int used to hold the amount of numbers in a set
 * @param amountB int used to hold the amount of numbers in a set
 * @param numbersA int array holds the values of the number set
 * @param numbersB int array holds the values of the number set
 */
static void displaySetDifference(int i, int j, int amountA, int amountB, int[] numbersA, int[] numbersB)
{
    int[] intersection; //array to store the intersection
    int[] setDifferenceA, setDifferenceB;

    intersection = getIntersection(i, j, amountA, amountB, numbersA, numbersB); //get intersection to work what
values are common
    setDifferenceA = getSetDifference(i, j, amountA, intersection, numbersA); //get set difference for set A and
then B
    setDifferenceB = getSetDifference(i, j, amountB, intersection, numbersB);
    outputSetDifference(i, setDifferenceA, 'A'); //output to console set A and B set differences
    outputSetDifference(i, setDifferenceB, 'B');
}

/**
 * Function to output set difference to the console
 * @param i int used as for loop counter
 * @param setDifference int array stores the set difference passed in
 * @param listLabel char used to display the set currently on
 */
static void outputSetDifference(int i, int[] setDifference, char listLabel)
{
    System.out.printf("\r\n---Set %c---\r\n", listLabel);

    for(i = 0; i < setDifference.length; i++)
    {
        if(i < setDifference.length - 1) //if number is less than one less than the amount of numbers in set
then...
        {
            System.out.printf(" %d ", setDifference[i]); //...print with comma
        }
        else
        {
            System.out.printf(" %d", setDifference[i]); //else print without meaning last number
        }
    }
}
}

```

J-Unit Tests

```

package Part_One_Assignment;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class Tests
{
    private Main main = new Main();

    private int[] numbers = new int[2];

    @Test
    void testMenuReturn() //Used to test if menu returns expected value
    {
        int option = 3;
        assertEquals(3, main.displayMenu(option, false)); //compare the expected value 3 with the return
    }

    @Test

```

```

void testNumberAmountReturn() //test to return the amount of numbers function
{
    assertEquals(0, main.enterAmountOfNumbers('A', true)); //compare expected value 0 to return
}

@Test
void testEnterNumbers() //Test for testing the return of array entered by the user
{
    assertEquals(new int[]{1, 2}, main.enterNumbers(0, 2, 'A', numbers, false));
}

@Test
void testDuplicateNumbers() //Test used to test duplicate numbers function
{
    boolean duplicate = false; //make duplicate false to begin with
    int[] numbersDuplicate = {2, 1}; //use a array to simulate numbers array
    //test if it returns true with comparing number to check if it is in numbersDuplicate array
    assertTrue(main.checkForDuplicate(0, 2, numbersDuplicate, 1));
}

@Test
void verifyQuitReturnTest() //Test used to verify the quit function returning expected results
{
    assertTrue(main.verifyQuit(false, false));
}

@Test
void verifyQuitReturnTestFalse() //Test used to test the opposite with a false expected quit
{
    assertFalse(main.verifyQuit(false, false));
}

@Test
void checkForMinValueTest() //Test used to check if the getMin function actually returns the min
{
    int[] numbersMinCheck = {10, 15, 20, 25, 5, 30, 35, 40, 45, 50}; //set an array of random numbers
    //expect 5 as the min from the list and assert the return equal to this
    assertEquals(5, main.getMin(0, 10, numbersMinCheck, numbersMinCheck[0]));
}

@Test
void checkForMaxValueTest() //Test used to check if the getMax function actually returns the max
{
    int[] numbersMaxCheck = {15, 25, 30, 65, 50}; //set an array of random numbers
    //expect 5 as the max from the list and assert the return equal to this
    assertEquals(65, main.getMax(0, 5, numbersMaxCheck, numbersMaxCheck[0]));
}

@Test
void checkCorrectAverage() //Test to check if a correct average has been calculated
{
    int[] numbersAverageCheck = {1, 10, 7, 12, 5};
    assertEquals(7.0, main.calculateAverage(0, 5, numbersAverageCheck, 'A', 0));
}

```

```

    }

    /**@Test
    void checkForCorrectOverlap() //Test to check if a true overlap is found
    {
        int[] numbersAOverlap = {3, 1, 2, 5, 4};
        int[] numbersBOverlap = {1, 2, 4, 6, 7};

        assertTrue(main.checkOverlap(0, 5, 5, numbersAOverlap, numbersBOverlap));
    }

    @Test
    void checkForFalseOverlap() //Test to check if a true overlap is found
    {
        int[] numbersAOverlap = {3, 1, 2, -1, -2};
        int[] numbersBOverlap = {3, 5, 4, 6, 7};

        assertFalse(main.checkOverlap(0, 5, 5, numbersAOverlap, numbersBOverlap));
    }*/

    @Test
    void testForTrueIntersection() //Test created to test if the intersection returns expected value
    {
        int[] intersectionA = {3, 4, 5, 2, 1};
        int[] intersectionB = {12, 10, 15, 3, 5, 11};
        int[] intersect = {3, 5}; //use this as expected assert value from the above generated lists
        assertEquals(intersect, main.getIntersection(0, 0, 5, 6, intersectionA, intersectionB));
    }

    @Test
    void testForSetDifference() //Test for checking the return of set difference function
    {
        int[] numbers = {1, 2, 3, 4, 5}; //create inputs and expected parameter as set difference
        int[] intersection = {1, 3, 4};
        int[] setDifference = {2, 5};
        assertEquals(setDifference, main.getSetDifference(0, 0, 5, intersection, numbers));
    }

}

```

Page Left Blank

PART TWO

Table of Contents

Table of Contents	I
Table of Figures	Error! Bookmark not defined.
Table of Figures	II
1. Introduction	1
2. Background Information	1
2.1. High Cohesion	1
2.2. Loose Coupling	2
2.3. Data Hiding and Information Hiding	3
3. Discussion	3
4. Programming Task	5
4.1. Task Brief	5
4.2. Problem Solving Methodology	6
5. Conclusion	7
References	8
Bibliography	10
Appendix	11
Problem Solving Mind Map	11
Git Log	13
Final Code	16
J-Unit Tests	27

Table of Tables

Table 1: Breakdown of Tasks Problems and Prioritization..... 6

1. Introduction

The purpose of this report is to present a discussion on the importance of *high cohesion*, *loose coupling*, *data hiding* and *information hiding* to software engineering. The report is structured with a background to all terms firstly introduced. Within the main body consists a discussion of the importance of each of the mentioned terms. Also provided within the main body is the process of solving a set programming task in Java, with the task being a console based connect four game. The problem-solving process undertaken to solve this task is also outlined within this section. All findings and concepts are concluded at the end of the report.

2. Background Information

This section of the report presents definitions from literature of *high cohesion*, *loose coupling*, *data hiding* and *information hiding*.

2.1. High Cohesion

Cohesion in software engineering relates to the extent of association between elements within a module [1] [2]. Authors Agarwal, Tayal and Gupta [3] define cohesion as:

“... the measure of the closeness of the relationships between ... components. “

Cohesion is measured as an ordinal metric with a module described as having high or low *cohesion*. *Low cohesion* is used to describe a class which has many functions that do not relate closely together, whereas *high cohesion* is used to describe a class where the majority of its functions are closely related to one and other [4].

There are various types/levels of *cohesion* which are as follows:

- **Coincidental:** When there is no relationship between any elements within a module. It is where a module will perform a random set of functions which are loosely related.
- **Logical:** When all a modules elements perform similar operations.
- **Temporal:** This is when a module contains functions which are dependent on execution. Functions are related by the fact that all functions are run at the same time.

- **Procedural:** Where a module has a set of functions which are related via a sequence. An example would be to decode a message where a function must be executed before another.
- **Communicational:** This is when all functions within a module work off the same data input or output.
- **Sequential:** This is when a modules functions output is used as an input for another function within that module.
- **Functional:** When a modules functions all cooperate with one and other to perform a single function [3].

The types define the amount of *cohesion* within a piece of software. *Coincidental cohesion* is defined as *low cohesion* and functional is defined as *high cohesion*.

2.2. Loose Coupling

Coupling in software engineering relates to the extent of independence within connections between modules measuring the strength of the relationship [3]. Defined by Chidamber and Kemerer [5]:

“... any evidence of a method of one object using methods or instance variables of another object constitutes coupling.”

Like *cohesion*, *coupling* is an ordinal measure with a module being described as having high (*tight*) or low (*loose*) *coupling*. *Tight coupling* describes software in which modules are highly dependent on one another. *Loose coupled* software is the opposite, meaning modules and classes are less dependent on one another [4].

There are various types/levels of *coupling* which are described as follows:

- **Content:** This is when a module can share and/or change the content of another module.
- **Common:** When multiple modules communicate through global data.
- **Control:** This is when a function of a module can control a function from another module's behaviour.
- **External:** When two modules share a data format or communication protocol between each other.
- **Message:** This is when modules and components use message passing as a means of communicating.

- **Stamp:** This is where a common data structure is shared between modules.
- **Data:** This is when two modules interact with one another through passing parameters [6] [7] [8].

2.3. Data Hiding and Information Hiding

Data hiding in software engineering is a term used in *Object-Oriented Programming (OOP)*. It refers to a technique which is used to hide data members and objects internally within a module [9].

Information hiding in software engineering is the hiding of information (processed data) within an internal structure within software i.e. an object or class. By hiding information, it reduces access other classes have to the information [10].

The main difference between *data hiding* and *information hiding* is within the difference in definition between data and information [10]. A way to achieve data hiding in an *OOP* environment is to use *encapsulation* [4].

Encapsulation is an *OOP* concept which is the principle of combining methods which use the same data into one module, this being known as a class. By making a specific piece of data only useable to an individual class means you can specify and control the access of other functions, thus allowing permissions to be made [11].

3. Discussion

This section of the report presents a discussion of the importance of the researched terms to software development.

High cohesion, *loose coupling*, *data hiding*, and *information hiding* are terms which have increased in importance in modern software engineering. This is due to the emergence of *OOP* techniques and languages such as *C++* and *Java* [12].

High cohesion in software engineering is a valuable quality metric which software developers strive to achieve in their developed software. Software designed with *high cohesion* can lead to high reusability of code [13]. This is due to designs which inherit *high cohesion* contain methods and functions which perform a smaller set of tasks, allowing developers to find common functionality between methods [14].

Software with higher *cohesion* can also improve the ease of maintenance and locating of bugs [15]. Functions which are split up into manageable sizes can make debugging easier. Testing is also made clearer as testers and developers can test smaller portions of code individually rather than testing dependent methods and functions together. This leads to produced software being more reliable and more manageable [7] [16]. It is particularly important in *Test Driven Development (TDD)* methodologies where tests are continuously developed and refactored.

Loose coupling is another quality metric which is desirable in software development. Software functions and methods which are less dependent on one another is important as if any changes are made within a function, it will not affect others [17]. This becomes particularly useful in organisations which employ AGILE methodologies where constantly changing user requirements is apparent e.g. the DVLA.

Many web services use multiple layers and modules. For a web service it is important that *loose coupling* is present within implementation. If a website is too dependent on certain artefacts or modules, then it could risk going down. *Loosely coupled* systems can also allow different layers of a web based service to be edited, meaning that the *user interface (UI)* layer can be changed without effecting the data processing or *back-end* layer [12] [18].

A *loosely coupled* system can also become more manageable and maintainable for developers. As modules are less dependent on one another it becomes clearer to locate what functions are causing bugs and to eventually rectify them [13].

Reusability of code is another large benefit of a *loosely coupled* systems. It means that software does not have to be redesigned every time a new function is created. It also allows services to potentially use a whole system to provide functionality to the creation of a new service.

Loose coupling will also give software a *platform agnostic* property. This means that the designed software is not dependant on the platform and will generally run on any platform [19].

The main goal of software is to feature both *loose coupling* and *high cohesion* together. However, a problem arises for developers as both contradict each other. Therefore, a balance needs to be decided upon the implementing software. The most well-known strategy for balancing these properties is known as the *Law of Demeter (LoD)* [13].

LoD is used as a guideline for use in *OOP* which is to help reduce coupling and higher cohesion within a system. There are many other guidelines which can be used to guide and project quality control into software development.

Data hiding and *information hiding* are also important factors in software development. By keeping variables within a class private it allows changes to be made to a variable ensuring there is no effect on any other classes' functionality [20]. This would not be true if for example a global variable is used between classes as there maybe multiple assignments of the variable within various classes.

As objects are not dependent on each other with a high amount of *data hiding*, it allows objects and methods to work in isolation, meaning that there is little dependency between objects [3]. If a method corrupts a variable, a system with a high level of *data hiding* would not allow the corrupt data to be passed between methods. This means that there is a lower chance of bugs and errors causing other methods to fail.

Data hiding and *information hiding* promotes both high cohesion and low coupling among objects are important considerations in producing quality software [21].

4. Programming Task

This section of the report presents the process taken to produce the final solution to the given programming task. The use of version control and *J-unit* testing is also presented within the section.

4.1. Task Brief

The task provided was to build a connect four console application in *Java*. The specification required a game of connect four to be played between two players with a winner being determined when a player connects four of their counters in a horizontal, vertical or diagonal line. However, the specification required modified rules which introduced a blitz move, which can clear a selected column and a time bomb, which can destroy counters located one piece around it horizontally, vertically and diagonally.

4.2. Problem Solving Methodology

Firstly, the problem was broken down into smaller more manageable problems. The problems which emerged from the task's requirements are outlined below (*table 1*).

Problem	Priority
• How are the rules going to be displayed to the user?	1
• How is the game board going to be displayed?	3
• How is the user going to enter a counter move?	4
• How is the user going to select a bomb or blitz?	13
• How is the user going to specify a move for the bomb?	15
• How is the user going to specify a move for the blitz?	14
• How are the counters going to drop?	8
• How is a row win going to be checked?	9
• How is a column win going to be checked?	10
• How is a diagonal win going to be checked?	11
• How is the game going to end once a player has won	12
• How is the bomb going to be tracked for turns?	17
• How is the program going to distinguish between players?	7
• How is the game board going to be updated with a move?	6
• How is the game board going to be created?	2
• How is the board going to be updated for a blitz?	16
Additional Problems	
• How is the user input going to be validated?	5
• How is the board updated with a String variable?	6.1
• How is the bomb going to explode the corners and edges?	18
• How are the counters going to drop above a bomb explosion?	19

Table 2: Breakdown of Tasks Problems and Prioritization

The problems outlined were prioritised with one being solved first. Prioritisation was made based around the difficulty of the problems. A mind map was drawn out of the problems with ideas around how each of the problems could be solved including data types and potential functions to be used.

After the mind map was drawn out each problem was attempted to be solved with a *test-driven development* approach taken. This meant a *J-unit* test was written to fail with the software refactored until it passed.

Version control was also used to provide backups (previous versions of code). *Bit bucket* was used to create a remote repository and *git* was used for controlling a local repository and used as the version control. The master branch was mainly used in the project; however, branches were created for more complex tasks such as win checking features and time bomb features. Evidence of both using *J-unit* for testing and *git* for version control is shown in the *appendix* with a full log of all commits performed.

5. Conclusion

High cohesion, loose coupling, data hiding, and information hiding are all considered important principles in software engineering. The research literature found shows a correlation between all principles with all four being highly desirable in software implementation. They are all especially important in *OOP*. Both *high cohesion* and *loose coupling* can make designed software easier to manage, maintain and adapt changes too. This being due to methods and functions becoming less dependent on one another and having a defined purpose. Unfortunately, both contradict one another, therefore a balancing strategy is required to ensure that both get implemented in a balanced way. There are many guidelines which are available with the most well-known being *LoD*.

Data hiding and *information hiding* are usually implemented using the principle of *encapsulation*, which ensures that data within objects is less dependent on each other's data. Thus, meaning changes are easier to implement. *Data hiding* and *information hiding* bring with them *high cohesion* and *loose coupling*. Overall all the aspects researched provide a basis in which quality software developed should strive towards.

The connect four programming task was solved with all requirements being completed. It is believed that using problem solving techniques beneficial in completing the task. Both problem identification and mind mapping techniques were found the most helpful. Overall it is believed the programming task is completed to a high standard. However, for future work the solution can be improved by making use of classes in java instead of using functions within one large class. This will improve the cohesion and coupling of the designed software.

References

- [1] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, 1979.
- [2] P. Dr. (Prof.) Rajendra and V. Prof. Govind, *Software Engineering*, Khanna Publishing, 2016.
- [3] B. B. Agarwal, S. P. Tayal and M. Gupta, *Software Engineering and Testing*, illustrated ed., Jones & Bartlett Learning, 2010.
- [4] M. Steve, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed., Microsoft Press, 2004.
- [5] R. C. Shyam and F. K. Chris, "Towards a Metrics Suite for Object-Oriented Design," in *OOPSLA '91*, 1991.
- [6] J. Eder, G. Kappel and M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems," 1995.
- [7] M. Hitz and M. Behzad, "Measuring coupling and cohesion in object-oriented systems," pp. 25-27, 1995.
- [8] M. Anup, "Question: Explain Coupling and Cohesion. Explain different types of coupling and cohesion.," 11 January 2018. [Online]. Available: <http://www.ques10.com/p/21518/explain-coupling-and-cohesion-explain-different-typ/>. [Accessed 2 January 2019].
- [9] S. Yogesh and M. Ruchika, *Object-Oriented Software Engineering*, PHI Learning Pvt. Ltd., 2012.
- [10] O. Effy, *Management Information Systems*, 6th ed., Cengage Learning, 2008.
- [11] J. Thorben, "OOP Concept for Beginners: What is Encapsulation," 30 November 2017. [Online]. Available: <https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/>. [Accessed 30 December 2018].
- [12] K. Doug, *Loosely Coupled: The Missing Pieces of Web Services*, RDS Strategies LLC, 2003.
- [13] A. M. Leszek, "From Hubs Via Holons to a Meta-Architecture - "AD-HOC" Approach," in *Software Engineering Techniques: Design for Quality*, K. Sacha, Ed., Springer Science & Business Media, 2006, pp. 1-14.
- [14] P.-J. Meilir, *Fundamentals of Object-Oriented Design in UML (Object Technology Series)*, 1st ed., Addison Wesley, 1999.
- [15] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*.

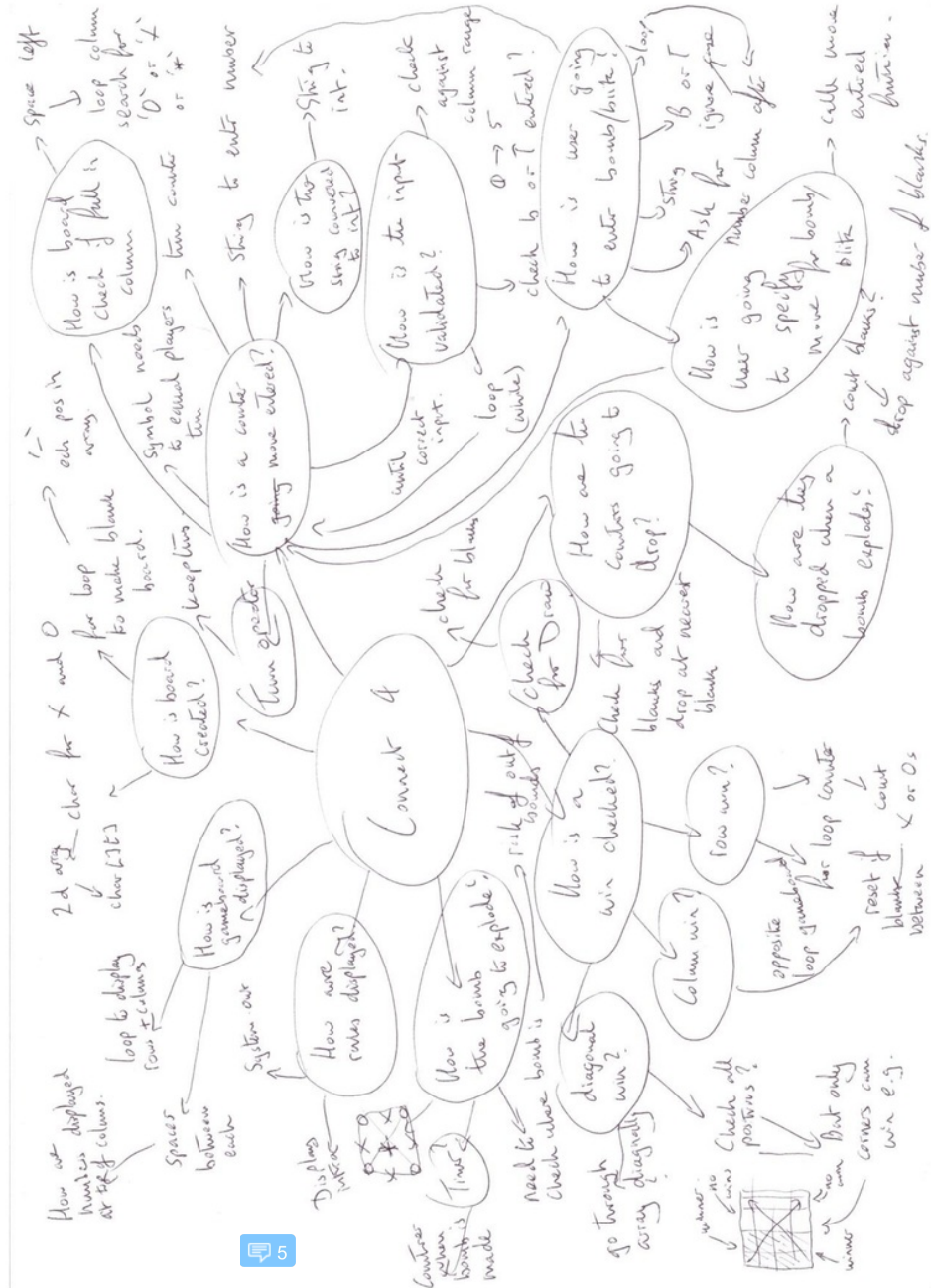
- [16] J. K. Lee, J. S. Seung, D. K. Soo, H. Woo and H. H. Dong, "Component identification method with coupling and cohesion," in *IEEE Eighth Asia-Pacific Software Engineering Conference*, 2001.
- [17] M. P. Papazoglou, T. Paolo, D. Schahram and L. Frank, "Service-Oriented Computing: State-of-the-Art and Open Research Issues," in *IEEE Computer*, 2007.
- [18] C. Pautasso and W. Erik, "Why is the web loosely coupled?: a multi-faceted metric for service design," in *Proceedings of the 18th international conference on World wide web*, 2009.
- [19] B. Rajkumar, *Mastering Cloud Computing*, Tata McGraw-Hill Education, 2013.
- [20] W. Pree, "Component-based software development-a new paradigm in software engineering?," in *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, Hong Kong, 1997.
- [21] R. Mali, *FUNDAMENTALS OF SOFTWARE ENGINEERING, FIFTH EDITION*, PHI Learning Pvt. Ltd., 2018.

Bibliography

1. P. Lorenzo, "The 5 Problem-Solving Skills of Great Software Developers," 13 November 2017. [Online]. Available: <https://www.coderhood.com/5-problem-solving-skills-great-software-developers/>. [Accessed 3 December 2018].
2. G. L. Milind, Software Testing, Tata McGraw-Hill Education, 2009.
3. G. Shekhar and S. Rahul, Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5, Apress, 2017.
4. S. Rajiv and S. Vivek, Java Programming by Example, CUP Archive, 1998.
5. K. Somasundaram, Introduction to Java Programming, Jaico Publishing House, 2014.
6. I. Vanderfeesten and W. M. V. d. A. Hajo A. Reijers, "Evaluating workflow process designs using cohesion and coupling metrics.," *Computers in Industry*, vol. 59, no. 5, pp. 420-437, 2008.

Appendix

Problem Solving Mind Map



Bomb Explode Corners

Char $[LiLi]$;

Hand-drawn grid for the 8-queens problem. The grid is 8x8 with columns labeled 0-7 and rows labeled 0-7. Queens are placed at (0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), and (7,7). The main diagonal is marked with 'x'. The anti-diagonal is marked with 'x'. The top-right 4x4 area is shaded with diagonal lines. The bottom-left 4x4 area is marked with 'x' in a grid pattern.

← can only win from this position no need to check whole board

Can do down from top

Bomb Explode Corners

[illegible]

So need to limit corners of bonds
→ need to limit row and column boundary bonds.

Git Log

```
User-Admin@DESKTOP-VCV1P9C MINGW64 /c/Programming and Software Testing/Part
Two/part_two_assignment (master)
```

```
$ git log
```

```
commit 7f3893f290fea0783404c6fe4517a0700cda0e1e (HEAD -> master)
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Tue Jan 1 11:36:45 2019 +0000
```

Refined print out new line statements

```
commit 5eea7d6929965bf6e381834a568f13b443db4459
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Tue Jan 1 11:31:05 2019 +0000
```

Refined bomb counter to explode on correct player move

```
commit 289d79d7185291ec75d125e4b5c66cb601d6d735
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Tue Jan 1 11:26:29 2019 +0000
```

Added and updated comments to main function turn operator

```
commit 88af19579d96f2ca74d1b99e6c3370f8ddff39e2 (performTimeBombFeature)
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 18:35:14 2018 +0000
```

Added test and function for checking if the game ends in a draw

```
commit a6bd97ea3cad5b25048c6740cecf8971c2d3318
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 18:21:39 2018 +0000
```

Ensured bomb explode function passed two previous tests and called into turn operator function

```
commit 0570b9a6293c8bb355cd7bbc23e59bb1e1acfba3
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 18:17:07 2018 +0000
```

Added test and function for counter gravity

```
commit 378c5575abaa19e119fcfd57afcd22656e94630a
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 14:51:57 2018 +0000
```

Added test and function for selecting which boundary explode function is required

```
commit 988cce3c83a6f5748a7b2d61f60f88e5714cc360
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 14:23:24 2018 +0000
```

Added tests and functions for exploding bomb at the corners of the board

```
commit fd2fd49b9f325ff97e5059dd0c4ac44ad591120c
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 13:50:30 2018 +0000
```

Added tests and functions for exploding column and row boundaries

```
commit 1f2eaac135ab17931a3838165058c661c0b6eba3
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 13:15:39 2018 +0000
```

Added tests and functions for checking if a bomb position had already been performed and for checking if a bomb remains on board

```
commit 4c26ac8f8aec9f82ba4c1da766b298f494ef6e1
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 12:52:07 2018 +0000
```

Added test and function for bomb exploding away from boudary

```
commit 63f9fd43e37a3805faa1d0ccbd377240e5fe383f
```

```
Author: AdamDavies07 <adamdavies07@btinternet.com>
```

```
Date: Mon Dec 31 12:31:43 2018 +0000
```

Added tests and function for bomb explode and check bomb position (bomb explode function failed tests due to lack of functionality)

commit 65a9994f216a780e527fa3783a9cfc4ca511042a
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Mon Dec 31 12:09:16 2018 +0000

Added test and function for bomb counter/timer

commit 73c0bd9f1afd10cdc1fe9cef42b763ddd8091887
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Mon Dec 31 11:52:20 2018 +0000

Added test and function for getting player bomb position

commit 40db7a94d8fd7174bad50cdf437d1087951759b0
(origin/performTimeBombFeature)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Fri Dec 28 12:14:14 2018 +0000

Added bomb placement test and function

commit e65df2c1da8e75a1fa6afcd927516e4d713998f6
(origin/performBlitzMoveFeature, origin/master, performBlitzMoveFeature)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 23:56:24 2018 +0000

Added display number of blitzs and bombs each player has remaining

commit 1814c6f068c965cc5d774f4547bf918affdc3c73
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 23:42:55 2018 +0000

Added functionality and tests for blitz function

commit fed55f9e7b655b4b5d1d6b26230b2bad7ee13991
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 22:04:19 2018 +0000

Added test and function for perform move and added code for user to enter abomb and blitz move without functionality

commit 727a1e414e18b32b6daf41603a773d232b08b666 (origin/checkForWinFeature, checkForWinFeature)
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 19:08:35 2018 +0000

Added function to display which player won the game

commit a69b20c930bbd86092df8fc081509d22ec88398c
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 19:01:55 2018 +0000

Added diagonal win check tests and functions

commit 42bfde9bcd58d3627aa5f5d5f842988abe55ccdc
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 16:33:53 2018 +0000

Added test and function for column winner check

commit 59eae1ee60e621cd37c161e5a9c21d6e155c43d8
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 16:19:47 2018 +0000

Added tests and function for checking a row win

commit 75857a368417f94894f62f456d0bbc84e897e2de
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Thu Dec 27 14:52:46 2018 +0000

Added test and function for checking spaces left in column and created more efficient turn function

commit 51d88088a0b2777bcb2d78bf86a9d28373f8d9e9
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 20:13:41 2018 +0000

Added test and function for placing counter on top of previous counters

```
commit ce29b4019565496ca385f3fb256eed24a018328e
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 19:14:10 2018 +0000
```

Added test and function to update game board (needs updating to multiple rows only on single row)

```
commit ae9e9b4a7f2071ffbbe508501e67d25b1013c211
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 18:41:54 2018 +0000
```

Added move counter and turn operator function with appropriate

```
commit 84136ab20c9f41d97abbc6878a299e38c15a4eb6
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 18:15:53 2018 +0000
```

Added player input validation tests and function

```
commit be98655246552a1b40581c6a80684818d8093761
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 17:22:33 2018 +0000
```

Added test and function for user input

```
commit 6e49a6e0dceb20ab9933af65e555f4d9b28333f3
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 16:56:24 2018 +0000
```

Added function to display blank game board

```
commit d2543fdf3facc36f24e1a1abea764d43cd6bf43f
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 16:31:14 2018 +0000
```

Added test and function for creating a blank game board

```
commit 645c1b126ae9785f0cb415c0b5090ce3dea7518b
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 15:47:30 2018 +0000
```

Added function to display a short intro to game

```
commit b854f7df448eaca5701fcfbf4252ae0b68ae48f9
Author: AdamDavies07 <adamdavies07@btinternet.com>
Date: Sat Dec 22 14:54:19 2018 +0000
```

Initial commit (added source and test files)

(END)

Final Code

```
package Connect_Four;
import java.util.*;

public class Main
{
    static Scanner stdin = new Scanner(System.in); //Scanner defined as static stdin for all functions to use

    public static void main(String[] args)
    {
        char[][] gameBoard = new char[7][6]; //char 2d array to hold gameboard with a hidden row for bomb and
        blitz info

        displayIntroduction(); //display introduction to the game
        createGameBoard(0, 0, gameBoard); //create a blank gameboard
        displayGameBoard(0,0, gameBoard); //display the blank gameboard
        turnOperator(gameBoard); //run the game going through each players turn
    }

    /**
     * Function which just displays an introduction to the game
     */
    static void displayIntroduction()
    {
        System.out.printf("-----Connect-4-----\r\n");
        System.out.printf("-----Instructions-----\r\n");
        System.out.printf("    -Player One is X    |\r\n");
        System.out.printf("    -Player Two is O   |\r\n");
        System.out.printf("    -T is for Time Bomb |\r\n");
        System.out.printf("    -B is for Blitz     |\r\n");
        System.out.printf("    -Must Connect 4 to Win! |\r\n");
        System.out.printf("-----\r\n");
    }

    /**
     * Function which creates a blank game board
     * @param i int used as a for loop counter
     * @param j int used as a alternative for loop counter
     * @param gameBoard char 2d array used to hold the entire game board
     * @return gameBoard char array which will hold blank game board
     */
    static char[][] createGameBoard(int i, int j, char[][] gameBoard)
    {
        for(i = 0; i < 6; i++) //nested for loop used to enter '-' into each array position this loops through
        columns
        {
            for(j = 0; j < 6; j++) //for loop to loop through rows
            {
                gameBoard[i][j] = '-';
            }
        }

        for(i = 0; i < 4; i++)
        {
            gameBoard[6][i] = '1';
        }

        return gameBoard; //return gameboard
    }

    /**
     * Function which displays the game board
     * @param i int used as a for loop counter
     * @param j int used as a alternative for loop counter
     * @param gameBoard char 2d array which holds the overall gameboard
     */
    static void displayGameBoard(int i, int j, char[][] gameBoard)
    {
        int displayCounter; //display counter used for displaying the numbers above columns

        for(i = 0; i < 6; i++) //nested for loop used to print out rows and columns
        {
            System.out.printf("\r\n");

            for(j = 0; j < 6; j++)
            {
                numbers
                if((i == 0) && (j == 0)) //if i is 0 and j is 0 it means your at start of grid print out column
                {
                    for(displayCounter = 0; displayCounter < 6; displayCounter++)
                    {
                        System.out.printf(" %d ", displayCounter); //prints numbers above columns
                    }

                    System.out.printf("\r\n");
                }
                System.out.printf(" %c ", gameBoard[i][j]); //print gameboard array
            }
        }
    }

    /**
     * Function which asks the user to enter a move
     * @param playerMove String which holds the player move value
     * @param moveMessage String which displays the move message to the player
     * @return String players move
     */
    static String playerMoveInput(String playerMove, String moveMessage, boolean bombBlitz, char[][] gameBoard, int
    playerBlitz, int playerBomb)
    {
    }
}
```

```

        boolean moveValid = false;           //move valid initially set as false

        while(moveValid == false)           //while loop used to check if move is valid keep looping if move is not valid
        {
            System.out.printf("\r\n%s", moveMessage);    //Ask player to enter the move
            playerMove = stdin.nextLine();

            moveValid = validateUserInput(playerMove, bombBlitz, gameBoard, playerBlitz, playerBomb);

        }

        return playerMove;           //return player move
    }

    /**
     * Function which validates the user input is within gameboard boundaries
     * @param playerMove String which holds the player move
     * @return boolean validMove which is true if the move entered is in range
     */
    static boolean validateUserInput(String playerMove, boolean bombBlitz, char[][] gameBoard, int playerBlitz, int playerBomb)
    {
        if(validatePlayerRange(playerMove)) //use validate player range function to check is player move is within range
        {
            return true;
        }
        else if (validateBombOrBlitzEntry(playerMove, gameBoard, bombBlitz, playerBlitz, playerBomb))
        {
            return true;
        }
        else //else move is invalid return false
        {
            System.out.printf("\r\nInvalid move!\r\nPlease try again.");
            return false;
        }
    }

    /**
     * Function which is used as conditional validate for bomb and blitz entry
     * @param playerMove String which holds the players entered move
     * @param gameBoard char 2d array which holds current gameboard
     * @param bombBlitz boolean which holds if the user has entered a bomb or blitz
     * @param playerBlitz int is the value and used as position ref to where the players blitz value is stored in array
     * @param playerBomb int is the value and used as position ref to where the players bomb value is stored in array
     * @return boolean true if the condition is met
     */
    static boolean validateBombOrBlitzEntry(String playerMove, char[][] gameBoard, boolean bombBlitz, int playerBlitz, int playerBomb)
    {
        return ((playerMove.equalsIgnoreCase("B")) && (bombBlitz == true) && (gameBoard[6][playerBlitz] == '1')) || ((playerMove.equalsIgnoreCase("T")) && (bombBlitz == true) && (gameBoard[6][playerBomb] == '1'));
    }

    /**
     * Function which holds conditional playermove statements
     * @param playerMove String which holds the entered player move
     * @return boolean which returns true if the condition is met
     */
    static boolean validatePlayerRange(String playerMove)
    {
        return ((playerMove.equalsIgnoreCase("0")) || (playerMove.equalsIgnoreCase("1")) || (playerMove.equalsIgnoreCase("2")) || (playerMove.equalsIgnoreCase("3")) || (playerMove.equalsIgnoreCase("4")) || (playerMove.equalsIgnoreCase("5")));
    }

    /**
     * Function which runs through the players turns checking if there is a winner or if the game ends in a draw
     * @param gameBoard char 2d array which holds the current gameboard
     */
    static void turnOperator(char[][] gameBoard)
    {
        String playerOneMove = "-"; playerTwoMove = "-"; //Strings which hold the players move as a string
        int playerOneBlitz = 0, playerTwoBlitz = 1, playerOneBomb = 2, playerTwoBomb = 3, playerOneBombActivated = 4, playerTwoBombActivated = 5; //ints which are used as position references for the hidden row in gameboard array
        int turnCounter = 1; //int which holds the overall games counter
        int playerOneBombCount = 2, playerTwoBombCount = 2; //ints which hold the bomb timers for each players bomb

        int[] playerOneBombPos = {10, 10}, playerTwoBombPos = {10, 10}; //int arrays which hold the location of players bomb with default values set as 10,10
        char symbol = '-'; //char which holds the current games counter symbol
        boolean win = false, draw = false; //boolean which end the game if made true holds if there is a win or draw has already been found

        boolean gotPlayerOneBombPos = false, gotPlayerTwoBombPos = false; //boolean which hold whether a bomb has already been found

        while((win == false) && (draw == false)) //while both draw and win are false perform player turns
        {
            if((win == false) && (draw == false)) //if both are false perform turns
            {
                System.out.printf("\r\n\r\n---Player One's Turn---\r\n");
                displayBlitzBombRemaining(gameBoard, playerOneBlitz, playerOneBomb); //display the number of bombs and blitz player one has
                gameBoard = playerTurn(gameBoard, playerOneMove, "Please enter move player one: ", turnCounter, symbol, playerOneBlitz, playerOneBomb, playerOneBombActivated); //perform player ones turn
                playerOneBombPos = getBombPosition(gameBoard, 0, 0, playerOneBombPos, playerTwoBombPos, gotPlayerOneBombPos); //get bomb position if bomb is selected
                playerTwoBombCount = bombCounter(playerTwoBombCount, gameBoard, playerTwoBombActivated); //set counter for player two if bomb is activated
                gameBoard = checkForBombExplode(playerTwoBombCount, gameBoard, playerTwoBombActivated, playerTwoBombPos); //explode bomb if it is ready for explosion
                win = checkForWin(gameBoard, 'X', win); //check for player win
                turnCounter = turnCounter + 1; //turn counter plus 1
            }
        }
    }

```

```

        draw = checkForDraw(draw, 0, 0, gameBoard); //check for a draw
    }

    if((win == false) && (draw == false)) //if both are false perform moves as no winner or draw has
    been found
    {
        System.out.printf("\n\n\n---Player Two's Turn---\n\n");
        displayBlitzBombRemaining(gameBoard, playerTwoBlitz, playerTwoBomb); //display the number of bombs
        and blitz player two has
        gameBoard = playerTurn(gameBoard, playerTwoMove, "Please enter move player two: ", turnCounter,
        symbol, playerTwoBlitz, playerTwoBomb, playerTwoBombActivated); //perform player two's turn
        gotPlayerTwoBombPos = getBombPosition(playerTwoBombPos); //check if a bomb position has been gotten
        playerTwoBombPos = getBombPosition(gameBoard, 0, 0, playerTwoBombPos, playerOneBombPos,
        gotPlayerTwoBombPos); //get bomb position if bomb is selected
        playerOneBombCount = bombCounter(playerOneBombCount, gameBoard, playerOneBombActivated); //set
        counter for player one if bomb is activated
        gameBoard = checkForBombExplode(playerOneBombCount, gameBoard, playerOneBombActivated,
        playerOneBombPos); //explode bomb if it is ready for explosion
        win = checkForWin(gameBoard, 'O', win); //check for player win
        turnCounter = turnCounter + 1; //turn counter plus 1
        draw = checkForDraw(draw, 0, 0, gameBoard); //check for a draw
    }

    checkPlayerWin(turnCounter, draw); //check for win or draw and display relevant player win
}

/**
 * Function which repeats player move functions
 * @param gameBoard char 2d array which holds the gameboard
 * @param playerMove String which holds the value of the entered players move
 * @param moveMessage String which holds which player to enter the move
 * @param turnCounter int which holds the turn number currently the game is on
 * @param symbol char which holds the game symbol
 * @return char 2d array of the gameboard
 */
static char[][] playerTurn(char[][] gameBoard, String playerMove, String moveMessage, int turnCounter, char
symbol, int playerBlitz, int playerBomb, int playerBombActivated)
{
    int moveType = 0;
    int playerMoveInt = 0; //set playerMoveInt to 0 which is the converted player move to an integer
    boolean spaceLeft = false; //initialise spaces left on board to false
    boolean blitz;

    while(spaceLeft == false) //while there are no spaces left in column repeat player move entry
    {
        blitz = false;
        playerMove = playerMoveInput(playerMove, moveMessage, true, gameBoard, playerBlitz, playerBomb);

        if(playerMove.equalsIgnoreCase("B"))
        {
            playerMove = playerMoveInput(playerMove, "Please enter column to blitz: ", false, gameBoard,
            playerBlitz, playerBomb);
            moveType = 1;
            gameBoard[6][playerBlitz] = 'O';
            blitz = true;
        }
        else if(playerMove.equalsIgnoreCase("T"))
        {
            playerMove = playerMoveInput(playerMove, "Please enter a column for the bomb: ", false, gameBoard,
            playerBlitz, playerBomb);
            gameBoard[6][playerBomb] = 'O';
            moveType = 2;
        }

        playerMoveInt = convertPlayerInputToInt(playerMove, playerMoveInt); //convert string to int to move symbol
        into array.
        spaceLeft = checkForRoom(gameBoard, playerMoveInt, blitz);
    }

    gameBoard = performMove(moveType, gameBoard, turnCounter, symbol, playerMoveInt, playerBombActivated);
    displayGameBoard(0, 0, gameBoard);

    return gameBoard;
}

/**
 * Function which performs all relevant moves the player can perform
 * @param moveType int used as a ref to what move the player has made
 * @param gameBoard char 2d array used to hold the current gameboard
 * @param turnCounter int holds the current turn the game is on
 * @param symbol char holds the current symbol used by the player
 * @param playerMoveInt int is the player move
 * @return gameBoard char 2d array which holds the updated gameboard
 */
static char[][] performMove(int moveType, char[][] gameBoard, int turnCounter, char symbol, int playerMoveInt, int
playerBombActivated)
{
    switch(moveType) //switch statement which checks against the move type int to perform relevant move
    {
        case 0: //standard move
            gameBoard = updateGameBoard(gameBoard, turnCounter, symbol, playerMoveInt); //update the gameboard
            once a valid move is entered
            break;

        case 1: //blitz move
            gameBoard = updateBlitzGameBoard(0, playerMoveInt, gameBoard);
            break;

        case 2: //bomb move
            gameBoard = updateBoardBomb(gameBoard, playerMoveInt, playerBombActivated);
            break;
    }

    return gameBoard;
}

```

```

/**
 * Function to check which players turn it currently is
 * @param turnCounter int which keeps track of the number of turns in the game
 * @param symbol char which holds the symbol of the return symbol
 * @return char symbol which is return value which holds the appropriate player token
 */
static char checkTurn(int turnCounter, char symbol)
{
    if(turnCounter % 2 == 0)           //if the number is even (calculated using modulus) then return O
    {
        symbol = 'O';
    }
    else                               //else return X
    {
        symbol = 'X';
    }

    return symbol;
}

/**
 * Function which updates the gameboard from player position entered
 * @param gameBoard char 2d array which holds the game board
 * @param turnCounter int which stores the current turn number
 * @param symbol char which holds the current player token in play
 * @param playerMoveInt int which is the players move entered and converted into an integer
 * @return char 2d array of the updated game board
 */
static char[][] updateGameBoard(char[][] gameBoard, int turnCounter, char symbol, int playerMoveInt)
{
    symbol = checkTurn(turnCounter, symbol); //use check turn function to get the players symbol
    gameBoard = placeCounter(gameBoard, 0, playerMoveInt, symbol); //place the counter in the next available space
    return gameBoard; //return gameboard
}

/**
 * Function which converts the player move from a string to an integer
 * @param playerMove string which is the original player move value
 * @param playerMoveInt player move int which is the converted player value
 * @return int playerMoveInt which will be an integer equivalent of the original string player move
 */
static int convertPlayerInputToInt(String playerMove, int playerMoveInt)
{
    playerMoveInt = Integer.parseInt(playerMove); //convert string to integer
    return playerMoveInt;
}

/**
 * Function for placing counter in the gameboard on top of previous moves
 * @param gameBoard char 2d array which holds gameboard data
 * @param i int used as for loop counter
 * @param playerMoveInt int player move entry
 * @param symbol char which holds the symbol token for the player
 * @return updated gameboard with counter placed at player position
 */
static char[][] placeCounter(char[][] gameBoard, int i, int playerMoveInt, char symbol)
{
    for(i = 5; i >= 0; i--) //start at bottom of board and place...
    {
        if(gameBoard[i][playerMoveInt] == '-') //...counter where the first blank position is
        {
            gameBoard[i][playerMoveInt] = symbol; //make gameboard position i and player column entry equal
            break; //break out when blank space found
        }
    }

    return gameBoard;
}

/**
 * Function to check if there is space on the gameboard in the entered column
 * @param gameBoard char 2d array which holds the current game board
 * @param playerMoveInt int which holds the value of intended player move
 * @return boolean which is true if there is room and false if column is full
 */
static boolean checkForRoom(char[][] gameBoard, int playerMoveInt, boolean blitz)
{
    if(blitz == false)
    {
        if ((gameBoard[0][playerMoveInt] == 'O') || (gameBoard[0][playerMoveInt] == 'X')) //check if the top of
the gameboard is allocated with a X or O at the column entered by player
        {
            System.out.printf("No room left in column!\r\nPlease try again!\r\n");
            return false;
        }
        else //else there is space on the board
        {
            return true;
        }
    }

    return true;
}

/**
 * Function which checks if all win functions return win as true
 * @param gameBoard char 2d array which holds the current gameboard
 * @param symbol char which holds the current symbol of the current players turn
 * @param win boolean which is set to true if a winner is found
 * @return win boolean which is true if a winner is found
 */
static boolean checkForWin(char[][] gameBoard, char symbol, boolean win)

```

```

    {
        int winCounter = 0;

        if(checkWinSituations(gameBoard, symbol, winCounter))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}

/**
 * Function which checks the gameboard for a row win
 * @param gameBoard 2d char array which holds current gameboard
 * @param i int used as a for loop counter
 * @param j int used as an alternative for loop counter
 * @param symbol char which holds the current players symbol of previous move
 * @param winCounter int which keeps a count of how many symbols are in a row
 * @return win boolean if a row winner is found it returns true
 */
static boolean checkForRowWin(char[][] gameBoard, int i, int j, char symbol, int winCounter)
{
    for(i = 0; i < 6; i++) //loop around 6 columns
    {
        winCounter = 0; //resetting the win counter after each row checked
        for(j = 0; j < 6; j++) //loop around each row
        {
            if(gameBoard[i][j] == symbol) //if the symbol in column i and row j is equal to the last symbol
entered then ...
            {
                winCounter = winCounter + 1; //add one to win counter
                if(winCounter == 4) //if the win counter is equal to four four in a row found
                {
                    return true; //return win as true
                }
            }
            else //if symbol is not sequentially one after each other reset win counter
            {
                winCounter = 0;
            }
        }
    }

    return false; //return false if winner not found
}

/**
 * Function which checks the gameboard for a column win
 * @param gameBoard 2d char array which holds current gameboard
 * @param i int used as a for loop counter
 * @param j int used as an alternative for loop counter
 * @param symbol char which holds the current players symbol of previous move
 * @param winCounter int which keeps a count of how many symbols are in a column
 * @return win boolean if a column winner is found it returns true
 */
static boolean checkForColumnWin(char[][] gameBoard, int i, int j, char symbol, int winCounter)
{
    for(i = 0; i < 6; i++) //loop around each row
    {
        winCounter = 0; //resetting the win counter after each column position is checked for each row
        for(j = 0; j < 6; j++) //loop around all columns checking that there are four previously entered
symbols in a row
        {
            if(gameBoard[j][i] == symbol)
            {
                winCounter = winCounter + 1; //increment win counter if symbols are in column
                if(winCounter == 4) //if four in a row are found then return winner as true
                {
                    return true;
                }
            }
            else
            {
                winCounter = 0; //reset win counter if there is a another symbol within a row of symbols
            }
        }
    }

    return false; //return false if winner is not found
}

/**
 * Function which checks for a diagonal winner from the game board
 * @param gameBoard char 2d array which holds the current gameboard layout
 * @param i int used as a for loop counter
 * @param j int used as an alternative for loop counter
 * @param symbol char which holds the current player symbol
 * @return true if a winner is found
 */
static boolean checkForDiagonalWin(char[][] gameBoard, int i, int j, char symbol)
{
    if((checkForDiagonalWinLeft(gameBoard, 0, 0, symbol)) || (checkForDiagonalWinRight(gameBoard, 0, 0, symbol)))
//if there is a diagonal winner from right or left checks then return true
    {
        return true;
    }

    return false;
}
}

```

```

/**
 * Function which checks for a diagonal win from the top left corner of the board to the bottom right
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int used as a for loop counter
 * @param j int used as a alternative for loop counter
 * @param symbol char which holds the current symbol by previous move
 * @return true if a diagonal win is found
 */
static boolean checkForDiagonalWinLeft(char[][] gameBoard, int i, int j, char symbol)
{
    for (i = 0; i < 3; i++) //start at row 0 and increment to three
    {
        for (j = 0; j < 3; j++) //only need to search 9x9 grid as no other potential winners
        {
            //increase each column checking the top left winner square of the board
            if ((gameBoard[j][i] == symbol && gameBoard[j + 1][i + 1] == symbol //if gameboard position i and
j (increased to 3 are in a diagonal row then...
&& gameBoard[j + 2][i + 2] == symbol && gameBoard[j + 3][i + 3] == symbol))
            {
                return true; //...winner is found
            }
        }
    }
    return false;
}

/**
 * Function which checks for a diagonal win from the top right corner of the board to the bottom left
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int used as a for loop counter
 * @param j int used as a alternative for loop counter
 * @param symbol char which holds the current symbol by previous move
 * @return true if a diagonal win is found
 */
static boolean checkForDiagonalWinRight(char[][] gameBoard, int i, int j, char symbol)
{
    for (i = 0; i < 3; i++) //loop around the rows
    {
        for (j = 5; j >= 3; j--) //loop around each column at bottom right corner of board
        {
            if ((gameBoard[j][i] == symbol && gameBoard[j - 1][i + 1] == symbol //check bottom right
position moving up towards top left diagonally
&& gameBoard[j - 2][i + 2] == symbol && gameBoard[j - 3][i + 3] == symbol)) //checking if
there are four matching symbols in a row
            {
                return true; //if there is return true win
            }
        }
    }
    return false;
}

/**
 * Function which holds the conditional statement check for each win condition
 * @param gameBoard char 2d array holds the current gameboard
 * @param symbol char which holds the current player symbol from previously made move
 * @param winCounter int which is the wincounter for column and row checks
 * @return true if the condition is met
 */
static boolean checkWinSituations(char[][] gameBoard, char symbol, int winCounter)
{
    return ((checkForRowWin(gameBoard, 0, 0, symbol, winCounter)) ||
(checkForColumnWin(gameBoard, 0, 0, symbol, winCounter)) ||
(checkForDiagonalWin(gameBoard, 0, 0, symbol)));
}

/**
 * Function which checks which player has one using the turn counter
 * @param turnCounter int which holds the total turns made in the game
 * @param draw boolean which is true if there is a draw game
 */
static void checkPlayerWin(int turnCounter, boolean draw)
{
    if (draw == true) //if draw is true then there is a draw game
    {
        System.out.printf("\n\nDraw Game!\n\n");
    }
    else if (turnCounter % 2 == 0) //if the turn is even then player one wins
    {
        System.out.printf("\n\n---Player 1 Wins!---\n\n");
    }
    else //if odd then player two wins
    {
        System.out.printf("\n\n---Player 2 Wins!---\n\n");
    }
}

/**
 * Function which performs a blitz on the column
 * @param i for loop counter
 * @param playerMoveInt int which holds player move
 * @param gameBoard char 2d array holds current gameboard
 * @return char 2d array gameboard which stores updated gameboard
 */
static char[][] updateBlitzGameBoard(int i, int playerMoveInt, char[][] gameBoard)
{
    for (i = 0; i < 6; i++)
    {
        gameBoard[i][playerMoveInt] = '-';
    }
    return gameBoard;
}

```

```

    * Function which displays the player with the number of bombs and blitzs they have remaining
    * @param gameBoard char 2d array which holds the gameboard
    * @param playerBlitz which is a ref to the player blitz value in the array
    * @param playerBomb int which is a ref to the player bomb value in the array
    */
    static void displayBlitzBombRemaining(char[][] gameBoard, int playerBlitz, int playerBomb)
    {
        System.out.printf("\r\nBlitz: %e", gameBoard[6][playerBlitz]);
        System.out.printf("\r\nTime Bomb: %e", gameBoard[6][playerBomb]);
    }

    /**
     * Function which updates the gameboard to place a bomb in the players intended column
     * @param gameBoard char 2d array which holds the current gameboard
     * @param playerMoveInt int which holds the current player move
     * @param playerBombActivated int which is the position reference in the gameboard array to the hidden row
     * @return char 2d array gameboard which holds the updated gameboard
     */
    static char[][] updateBoardBomb(char[][] gameBoard, int playerMoveInt, int playerBombActivated)
    {
        gameBoard = placeCounter(gameBoard, 0, playerMoveInt, '*'); //call place counter function and make the
        //symbol as the bomb symbol
        gameBoard[6][playerBombActivated] = '1'; //make players bomb 1 at the secret row in the gameboard array
        //using player bomb activated as a pos ref
        return gameBoard;
    }

    /**
     * Function which gets the current bomb position from the player
     * @param gameBoard char 2d array which holds the current gameboard
     * @param i int used as a for loop counter
     * @param j int used as a alternative for loop counter
     * @param currentPlayerBombPos int array used to store the coordinates for the current players bomb
     * @param previousPlayerBombPos int array used to store previous bomb location
     * @param gotPlayerBombPos boolean which will be made false if the player has not previously used a bomb
     * @return int array playerbomb position which holds the current players bomb coordinates
     */
    static int[] getBombPosition(char[][] gameBoard, int i, int j, int[] currentPlayerBombPos, int[]
    previousPlayerBombPos, boolean gotPlayerBombPos)
    {
        if(gotPlayerBombPos == false) //if player bomb position has been got before do not perform code
        {
            for (i = 0; i < 6; i++) //loop around each row and ...
            {
                for (j = 0; j < 6; j++) //each column of the gameboard
                {
                    if ((gameBoard[i][j] == '*') && ((previousPlayerBombPos[0] != i) || (previousPlayerBombPos[1] !=
                    j))) //if a bomb symbol is found in a column and it is not equal to the previous players bomb
                    {
                        currentPlayerBombPos[0] = i; //then make array pos 0 equal to rows and...
                        currentPlayerBombPos[1] = j; //make array pos 1 equal to columns
                        return currentPlayerBombPos; //return bomb position
                    }
                }
            }
        }
        return currentPlayerBombPos;
    }

    /**
     * Function which decreases the timer for the relevant bomb
     * @param playerBombCount int which is the counter for the current players bomb
     * @param gameBoard char 2d array which holds the current gameboard information
     * @param playerBomb int which is the reference position to the hidden row in gameboard to check if bomb is
    activated
     * @return int players bomb counter/timer
     */
    static int bombCounter(int playerBombCount, char[][] gameBoard, int playerBomb)
    {
        if((gameBoard[6][playerBomb] == '1') && (playerBombCount >= 0)) //if the bomb is activated and bomb
        //counter is greater than 0
        {
            playerBombCount = playerBombCount - 1; // - 1 from bomb timer
        }
        return playerBombCount; //return bomb timer
    }

    /**
     * Function which controls how the bomb will explode by using bomb ref as a reference for boundary check
     * @param playerBombPos int array which holds the players bomb location on the board
     * @param gameBoard char 2d array which holds the current gameboard
     * @return 2d char array gameboard which returns updated gameboard
     */
    static char[][] bombExplode(int[] playerBombPos, char[][] gameBoard)
    {
        int bombRef = 0; //int which holds the bomb reference number from check bomb pos
        bombRef = checkBombPosition(playerBombPos); //call check bomb position function to check if the bomb is on
        //a boundary
        if(bombRef == 1) //if bomb is on the boundry then call bomb explodes boundary selector
        {
            gameBoard = bombExplodeBoundarySelect(gameBoard, playerBombPos);
        }
        else if(bombRef == 2) //if bomb ref is 2 then bomb is away from boundary
        {
            gameBoard = bombExplodeCentre(gameBoard, 0, 0, playerBombPos);
        }
        gameBoard = checkFloatingCounters(gameBoard, 0, 0);
        System.out.printf("\r\n\r\n---Bomb Explode---\r\n");
        displayGameBoard(0, 0, gameBoard); //display bomb exploded
    }

```

```

        return gameBoard;
    }

    /**
     * Function which checks the position of the bomb
     * @param playerBombPos int array which holds the bombs location
     * @return bombPos int which will hold the value for bomb explode to use
     */
    static int checkBombPosition(int[] playerBombPos)
    {
        if((playerBombPos[0] == 0) || (playerBombPos[0] == 5) || (playerBombPos[1] == 0) || (playerBombPos[1] == 5))
        //if bomb is on the boundary return 1
        {
            return 1;
        }
        else //if the bomb is away from boundaries/edges of board return 2
        {
            return 2;
        }
    }

    /**
     * Function which explodes the bomb when it is away from the edge of the board
     * @param gameBoard 2d char array which holds the current gameboard
     * @param i int which is used as a for loop counter
     * @param j int which is used as an alternative for loop counter
     * @param playerBombPos int array which holds the location to the current players bomb
     * @return char 2d array which returns updated gameboard
     */
    static char[][] bombExplodeCentre(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        for(i = playerBombPos[0] - 1; i <= playerBombPos[0] + 1; i++) //loop around column starting at -1 from
        bomb position and going till +1 of bomb position
        {
            for(j = playerBombPos[1] - 1; j <= playerBombPos[1] + 1; j++) //loop around rows doing the same as
            columns
            {
                gameBoard[i][j] = '-'; //replace each area of gameboard the bomb is near with a blank space
            }
        }
        return gameBoard;
    }

    /**
     * Function which checks if a bomb has already been used
     * @param playerBombPos int array which holds the current players bomb position
     * @return true if a bomb has already been located for that player
     */
    static boolean gotBombPosition(int[] playerBombPos)
    {
        if((playerBombPos[0] != 10) && (playerBombPos[1] != 10)) //check if the default value of the players
        bomb position has been changed
        {
            return true;
        }
        return false; //return false if it has not changed
    }

    /**
     * Function which checks if a bomb is remaining on the board
     * @param gameBoard char 2d array which holds the gameboard
     * @param i int used as a for loop counter
     * @param j int used as a alternative for loop counter
     * @param playerBombPos int array used as the location for the current players bomb
     * @return boolean true if a bomb is remaining
     */
    static boolean checkIfBombRemains(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        for (i = 0; i < 6; i++) //loop around rows
        {
            for (j = 0; j < 6; j++) //loop around columns of gameboard and check...
            {
                if((gameBoard[i][j] == '*') && (playerBombPos[0] == i) && (playerBombPos[1] == j)) //if there is a
                bomb within the location of the current bomb position
                {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     * Function which explodes on the lower row boundary (row 5 of array)
     * @param gameBoard char 2d array which holds the current gameboard
     * @param i int which is used as a for loop counter
     * @param j int which is used as a alternative for loop counter
     * @param playerBombPos int array which holds the players bomb location
     * @return gameboard 2d char array which holds the updated gameboard
     */
    static char[][] bombExplodeBoundaryLowerRow(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        //Stops looping going below array position 5 so limits nothing more than 5 so it does not add blanks to hidden
        array row
        for(i = playerBombPos[0] - 1; i <= playerBombPos[0]; i++) //loop around rows starting at -1 of bomb pos
        and looping till centre
        {
            for(j = playerBombPos[1] - 1; j <= playerBombPos[1] + 1; j++) //loop around columns starting at -1
            across until +1 of bomb centre is reached
            {
                gameBoard[i][j] = '-'; //replacing counters with blanks
            }
        }
    }

```

```

    }
    return gameBoard;
}

/**
 * Function which explodes on the upper row boundary (row 0 of array)
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int which is used as a for loop counter
 * @param j int which is used as a alternative for loop counter
 * @param playerBombPos int array which holds the players bomb location
 * @return gameboard 2d char array which holds the updated gameboard
 */
static char[][] bombExplodeBoundaryUpperRow(char[][] gameBoard, int i, int j, int[] playerBombPos)
{
    //Stops looping going above array position so limits nothing less than 0 so no out of bounds takes place
    for(i = playerBombPos[0]; i <= playerBombPos[0] + 1; i++) //loop around rows starting at centre of bomb
        and moving plus 1 from bomb row pos
    {
        for(j = playerBombPos[1] - 1; j <= playerBombPos[1] + 1; j++) //loop around columns starting at one -1
            across bomb pos and moving one across +1 from centre of bomb
        {
            gameBoard[i][j] = '-'; //make positions blank spaces
        }
    }
    return gameBoard;
}

/**
 * Function which explodes on the left column (column 0 of array)
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int which is used as a for loop counter
 * @param j int which is used as a alternative for loop counter
 * @param playerBombPos int array which holds the players bomb location
 * @return gameboard 2d char array which holds the updated gameboard
 */
static char[][] bombExplodeBoundaryLeftColumn(char[][] gameBoard, int i, int j, int[] playerBombPos)
{
    //loop around exploding around the bomb limiting to going towards left of bomb to stop going out of array
    bounds
    for(i = playerBombPos[0] - 1; i <= playerBombPos[0] + 1; i++) //loop around rows starting at -1 down and
        working to +1 of centre of bomb
    {
        for(j = playerBombPos[1]; j <= playerBombPos[1] + 1; j++) //loop around column starting at centre of
            bomb and looping to plus 1 of centre (to the right)
        {
            gameBoard[i][j] = '-'; //replace counters with blanks
        }
    }
    return gameBoard;
}

/**
 * Function which explodes on the right column (column 5 of array)
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int which is used as a for loop counter
 * @param j int which is used as a alternative for loop counter
 * @param playerBombPos int array which holds the players bomb location
 * @return gameboard 2d char array which holds the updated gameboard
 */
static char[][] bombExplodeBoundaryRightColumn(char[][] gameBoard, int i, int j, int[] playerBombPos)
{
    //loop around exploding around the bomb limiting to going towards right of bomb to stop going out of array
    bounds
    for(i = playerBombPos[0] - 1; i <= playerBombPos[0] + 1; i++) //loop around rows starting at -1 down and
        working to +1 of centre of bomb
    {
        for(j = playerBombPos[1] - 1; j <= playerBombPos[1]; j++) //loop around column starting at centre of bomb
            and looping to plus 1 of centre (to the left)
        {
            gameBoard[i][j] = '-'; //replace counters with blanks
        }
    }
    return gameBoard;
}

/**
 * Function which explodes bomb on the top left corner
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int which is used as a for loop counter
 * @param j int which is used as a alternative for loop counter
 * @param playerBombPos int array which holds the players bomb location
 * @return gameboard 2d char array which holds the updated gameboard
 */
static char[][] bombExplodeTopLeftCorner(char[][] gameBoard, int i, int j, int[] playerBombPos)
{
    //loop through rows and columns limiting the top of bomb position and the left of bomb position
    for(i = playerBombPos[0]; i <= playerBombPos[0] + 1; i++)
    {
        for(j = playerBombPos[1]; j <= playerBombPos[1] + 1; j++)
        {
            gameBoard[i][j] = '-'; //replace counters with blank spaces
        }
    }
    return gameBoard;
}

/**
 * Function which explodes bomb on the top right corner
 * @param gameBoard char 2d array which holds the current gameboard
 * @param i int which is used as a for loop counter
 * @param j int which is used as a alternative for loop counter

```

```

    * @param playerBombPos int array which holds the players bomb location
    * @return gameBoard 2d char array which holds the updated gameboard
    */
    static char[][] bombExplodeTopRightCorner(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        //loop through rows and columns limiting the top of bomb position and the right of bomb position
        for(i = playerBombPos[0]; i <= playerBombPos[0] + 1; i++)
        {
            for(j = playerBombPos[1] - 1; j <= playerBombPos[1]; j++)
            {
                gameBoard[i][j] = '-'; //replace counters with blank spaces
            }
        }

        return gameBoard;
    }

    /**
    * Function which explodes bomb on the bottom left corner
    * @param gameBoard char 2d array which holds the current gameboard
    * @param i int which is used as a for loop counter
    * @param j int which is used as a alternative for loop counter
    * @param playerBombPos int array which holds the players bomb location
    * @return gameBoard 2d char array which holds the updated gameboard
    */
    static char[][] bombExplodeBottomLeftCorner(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        //loop through rows and columns limiting the bottom of bomb position and the left of bomb position
        for(i = playerBombPos[0] - 1; i <= playerBombPos[0]; i++)
        {
            for(j = playerBombPos[1]; j <= playerBombPos[1] + 1; j++)
            {
                gameBoard[i][j] = '-'; //replace counters with blank spaces
            }
        }

        return gameBoard;
    }

    /**
    * Function which explodes bomb on the bottom right corner
    * @param gameBoard char 2d array which holds the current gameboard
    * @param i int which is used as a for loop counter
    * @param j int which is used as a alternative for loop counter
    * @param playerBombPos int array which holds the players bomb location
    * @return gameBoard 2d char array which holds the updated gameboard
    */
    static char[][] bombExplodeBottomRightCorner(char[][] gameBoard, int i, int j, int[] playerBombPos)
    {
        //loop through rows and columns limiting the bottom of bomb position and the right of bomb position
        for(i = playerBombPos[0] - 1; i <= playerBombPos[0]; i++)
        {
            for(j = playerBombPos[1] - 1; j <= playerBombPos[1]; j++)
            {
                gameBoard[i][j] = '-'; //replace counters with blank spaces
            }
        }

        return gameBoard;
    }

    /**
    * Function which selects which explodes boundary function to call based on bomb position
    * @param gameBoard char 2d array which holds the current game board
    * @param playerBombPos int array which holds current players bomb location
    * @return char 2d array which holds the updated gameboard
    */
    static char[][] bombExplodeBoundarySelect(char[][] gameBoard, int[] playerBombPos)
    {
        if((playerBombPos[1] >= 1) && (playerBombPos[1] <= 4) && (playerBombPos[0] == 0)) //if bomb position is
        in top row and not in corner column
        {
            gameBoard = bombExplodeBoundaryUpperRow(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[1] >= 1) && (playerBombPos[1] <= 4) && (playerBombPos[0] == 5)) //if bomb is in
        lower row and not in corner of column
        {
            gameBoard = bombExplodeBoundaryLowerRow(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] >= 1) && (playerBombPos[0] <= 4) && (playerBombPos[1] == 0)) //if bomb is in left
        column and not in bottom or top of row
        {
            gameBoard = bombExplodeBoundaryLeftColumn(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] >= 1) && (playerBombPos[0] <= 4) && (playerBombPos[1] == 5)) //if bomb is in right
        column and not in bottom or top of row
        {
            gameBoard = bombExplodeBoundaryRightColumn(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] == 0) && (playerBombPos[1] == 0)) //if bomb is in top left corner
        {
            gameBoard = bombExplodeTopLeftCorner(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] == 0) && (playerBombPos[1] == 5)) //if bomb is in top right corner
        {
            gameBoard = bombExplodeTopRightCorner(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] == 5) && (playerBombPos[1] == 0)) //if bomb is in bottom left corner
        {
            gameBoard = bombExplodeBottomLeftCorner(gameBoard, 0, 0, playerBombPos);
        }
        else if((playerBombPos[0] == 5) && (playerBombPos[1] == 5)) //if bomb is in bottom right corner
        {
            gameBoard = bombExplodeBottomRightCorner(gameBoard, 0, 0, playerBombPos);
        }
    }

```

```

        return gameBoard;
    }

    /**
     * Function which checks if a bomb has exploded
     * @param playerBombCount int which holds the current player bombs timer value
     * @param gameBoard char 2d array which holds the current gameboard
     * @param playerBomb int which holds the reference mark for hidden row on gameboard (if player has bomb used)
     * @param playerBombPos int array which holds the player bomb position
     * @return char 2d array gameboard which holds the updated gameboard with updated hidden row
     */
    static char[][] checkForBombExplode(int playerBombCount, char[][] gameBoard, int playerBomb, int[] playerBombPos)
    {
        boolean bombRemaining = false; //boolean which checks if a bomb remains

        bombRemaining = checkIfBombRemains(gameBoard, 0, 0, playerBombPos); //call check if bomb remains function
        and pass return to boolean bombRemaining

        if((playerBombCount == 0) && (bombRemaining == true)) //if the bomb timer is 0 and bomb remaining is true
        then...
        {
            gameBoard[6][playerBomb] = '2'; //make players bomb equal two
            gameBoard = bombExplode(playerBombPos, gameBoard); //and call bomb explode function updating the gameboard
        }

        return gameBoard;
    }

    /**
     * Function which checks for floating counters and makes counters above blanks fall
     * @param gameBoard char 2d array which holds the current gameboard
     * @param i int which is used as a for loop counter
     * @param j int which is used as a alternative for loop counter
     * @return gameboard which is updated with falling counters
     */
    static char[][] checkFloatingCounters(char[][] gameBoard, int i, int j)
    {
        int blankCounter; //blank counter keeps count of any blank counters in each column

        for(i = 5; i >= 0; i--) //loop through each column starting at right most column
        {
            blankCounter = 0; //reset blankcounter to 0

            for(j = 5; j >= 0; j--) //loop through rows strating at bottom of board
            {
                if(gameBoard[j][i] == '-') //if a blank counter is found then add one to blank counter
                {
                    blankCounter = blankCounter + 1;
                }
                else if((gameBoard[j][i] == 'X') || (gameBoard[j][i] == 'O') || (gameBoard[j][i] == '*')) &&
                (blankCounter != 0) //if a symbol counter is found then...
                {
                    gameBoard[j + blankCounter][i] = gameBoard[j][i]; //make gameboard position j + blankcounter
                    in the column equal to current symbol
                    gameBoard[j][i] = '-'; //replace where the symbol was with a blank space
                }
            }

            return gameBoard;
        }
    }

    /**
     * Function which checks if there is a draw game
     * @param draw boolean which is true if the board is full
     * @param i int which is used as a for loop counter
     * @param j int which is used as an alternative for loop counter
     * @param gameBoard char 2d array which holds the current game board
     * @return boolean which is true if board is full and there is no winner
     */
    static boolean checkForDraw(boolean draw, int i, int j, char[][] gameBoard)
    {
        for(i = 0; i < 6; i++) //check all rows
        {
            for(j = 0; j < 6; j++) //check all columns
            {
                if(gameBoard[i][j] == '-') //if there is a blank space on game board return false
                {
                    return false;
                }
            }
        }

        return true;
    }
}

```

J-Unit Tests

```
package Connect_Four;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MainTest
{
    static Main main = new Main();

    static int[] evenTurnCounter = new int[18];
    static char[][] expectedBlankBoard = { //char array used to hold a expected game board
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'}
    };

    @Test
    void testBlankGameBoard() //Test created to test if a correct sized blank gameboard created
    {
        char[][] testGameBoard = new char[6][6]; //char array used as a test game board

        assertEquals(expectedBlankBoard, main.createGameBoard(0,0, testGameBoard)); //test return againstt
        blank game board created
    }

    @Test
    void testPlayerInput() //Test created to test if the user input function returns an expected value
    {
        assertEquals("3", main.playerMoveInput("3", "One's", false, expectedBlankBoard, 1, 3));
    }

    @Test
    void testValidPlayerInput() //Test created to check for a valid user input entry
    {
        String[] validValues = {"1", "2", "3", "4", "5"}; //string array which holds all valid values
        for(int i = 0; i < 5; i++) //for loop to check all valid values
        {
            assertTrue(main.validateUserInput(validValues[i], false, expectedBlankBoard, 1, 3));
        }
    }

    @Test
    void testInvalidPlayerInput() //Test created to test for an invalid entry by the user
    {
        assertFalse(main.validateUserInput("6", false, expectedBlankBoard, 1, 3));
    }

    @Test
    void testForOTurn() //test created to test if turn check function returns a expected O for player two
    {
        for(int i = 2; i < 37; i = i + 2) //for loop to check all possible turn counter values
        {
            assertEquals('O', main.checkTurn(i, '-'));
        }
    }

    @Test
    void testForXTurn() //Test created to test if turn check function returns an expected X for player one
    {
        for(int i = 1; i < 37; i = i + 2) //for loop to check all possible turn counters
        {
            assertEquals('X', main.checkTurn(i, '-'));
        }
    }

    @Test
    void testForCorrectStringToInt() //test for correct return from string conversion function
    {
        assertEquals(2, main.convertPlayerInputToInt("2", 0));
    }

    @Test
    //test created to test for expected updated game board
    void testForUpdatedGameBoard()
    {
        char[][] expectedMoveGameBoard = { //char array used to hold a expected game board
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'X', '-', '-', '-', '-', '-'}
        };

        assertEquals(expectedMoveGameBoard, main.updateGameBoard(expectedBlankBoard, 1, 'X', 0));
    }

    @Test
    //test created to test the correct return of place counter function
    void testCorrectPositioningOfCounter()
    {
        char[][] expectedMoveGameBoard = { //char array used to hold a expected game board
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'-', '-', '-', '-', '-', '-'},
            {'O', '-', '-', '-', '-', '-'},
            {'X', '-', '-', '-', '-', '-'}
        };

        char[][] currentGameBoard = {

```

```

        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'}];

    assertEquals(expectedMoveGameBoard, main.placeCounter(currentGameBoard, 0, 0, 'O'));
}

@Test
void testRoomOnBoard() //test created to check that room check function returns true when there is space in
the column entered by the player
{
    char[][] expectedMoveGameBoard = { //char array used to hold a expected game board
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'}];

    assertTrue(main.checkForRoom(expectedMoveGameBoard, 0, false));
}

@Test
void testNoRoomOnBoard() //test created to check that room check function returns false when there is no
room left
{
    char[][] expectedMoveGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'}];

    assertFalse(main.checkForRoom(expectedMoveGameBoard, 0, false));
}

@Test
void checkForNoRowWin() //Test to check if a row win returns an expected false
{
    char[][] noWinGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', 'X', '-', '-', '-', '-'},
        {'X', 'O', 'O', 'X', 'O', 'X'}];

    assertFalse(main.checkForRowWin(noWinGameBoard, 0, 0, 'X', 0));
}

@Test
void checkForRowWin() //Test created to check an expected row win to return true
{
    char[][] winGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', 'O', '-'},
        {'O', 'X', 'X', 'X', 'X', 'O'},
        {'X', 'O', 'O', 'X', 'O', 'X'}];

    assertTrue(main.checkForRowWin(winGameBoard, 0, 0, 'X', 0));
}

@Test
void checkForNoColumnWin() //test created to test for a false column winner
{
    char[][] noWinGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', 'X', '-', '-', '-', '-'},
        {'X', 'O', 'O', 'X', 'O', 'X'}];

    assertFalse(main.checkForColumnWin(noWinGameBoard, 0, 0, 'X', 0));
}

@Test
void checkForColumnWin() //Test created to check for a column winner returns true
{
    char[][] winGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', 'X', '-', '-', '-', '-'},
        {'O', 'X', '-', '-', '-', '-'},
        {'X', 'X', 'O', '-', '-', '-'},
        {'O', 'X', 'O', '-', '-', '-'},
        {'X', 'O', 'O', 'X', 'O', 'X'}];

    assertTrue(main.checkForColumnWin(winGameBoard, 0, 0, 'X', 0));
}

@Test
void testDiagonalWinLeft() //test created to check if a expected diaognal winner from the bottom right to top
left is true
{
    char[][] winGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', 'O', '-', '-', '-', '-'},
        {'O', 'X', 'X', '-', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'O', 'O', 'X', '-'}];

```

```

        {'X', 'O', 'O', 'X', 'O', 'X'},];

    assertTrue(main.checkForDiagonalWinLeft(winGameBoard, 0, 0, 'X'));
}

@Test
void testDiagonalWinRight()    //test created to check if a expected diagonal winner from the bottom left to top
right is true
{
    char[][] winGameBoard = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', 'O'},
        {'X', 'O', '-', '-', '-', 'X'},
        {'O', 'X', 'O', 'O', 'X', 'X'},
        {'X', 'X', 'O', 'X', 'O', 'X'},
        {'O', 'X', 'O', 'O', 'X', 'O'},
        {'X', 'O', 'O', 'X', 'O', 'X'},];

    assertTrue(main.checkForDiagonalWinRight(winGameBoard, 0, 0, 'O'));
}

@Test
void testPerformMove()    //test created to check if the expected move is performed by the function
{
    char[][] expectedGameBoard = { //char array used to hold a expected game board
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', 'X', '-', '-', '-', '-'},];

    assertEquals(expectedGameBoard, main.performMove(0, expectedBlankBoard, 1, 'X', 1, 4));
}

@Test
void testBlitzColumn()    //test created to test whether a column is successfully blitzed
{
    char[][] winGameBoard = { //char array used to hold a expected game board
        {'-', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},];

    assertEquals(expectedBlankBoard, main.updateBlitzGameBoard(0, 0, winGameBoard));
}

@Test
void testBombMove()    //test created to test whether a bomb is entered at an expected column on the board
{
    char[][] expectedGameBoardBomb = { //char array used to hold a expected game board
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '-'},
        {'-', '-', '-', '-', '-', '*'},];

    assertEquals(expectedGameBoardBomb, main.updateBoardBomb(expectedBlankBoard, 5, 4));
}

@Test
void testGetBombPos()    //test created to test for correct bomb location even with previous players bomb
located
{
    char[][] gameBoardWithBomb = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', 'O', '-', '-', '-', '-'},
        {'O', 'X', 'X', '*', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'O', 'O', 'X', '*'},
        {'X', 'O', 'O', 'X', 'O', 'X'},];

    int[] currentBombPos = {10, 10};
    int[] expectedReturn = {4, 5};
    int[] previousBombPos = {2, 3};

    assertEquals(expectedReturn, main.getBombPosition(gameBoardWithBomb, 0, 0, currentBombPos,
previousBombPos, false));
}

@Test
void testForCorrectBombCounter()    //test created to check for correct return from the bomb counter
{
    char[][] gameBoardWithBomb = { //char array used to hold a expected game board
        {'O', '-', '-', '-', '-', '-'},
        {'X', 'O', '-', '-', '-', '-'},
        {'O', 'X', 'X', '*', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'O', 'O', 'X', '*'},
        {'X', 'O', 'O', 'X', 'O', 'X'},
        {'1', '1', 'O', 'O', '1', '1'},];

    assertEquals(1, main.bombCounter(2, gameBoardWithBomb, 4));
}

@Test
void testCheckBombPositionOnBoundary()    //test created to check if checkBombPosition function returns 1 when
bomb is on the edge of board
{
    int[] bombPos = {4, 0};

    assertEquals(1, main.checkBombPosition(bombPos));
}

```

```

@Test
void testCheckBombPositionAwayBoundary() //test created to check if checkBombPosition function returns 2 when bomb
is on around centre of board
{
    int[] bombPos = {4, 3};

    assertEquals(2, main.checkBombPosition(bombPos));
}

@Test
void testBombExplodeAwayBoundary()
{
    char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', '*', 'O', 'X', '-'},
        {'X', 'O', 'O', 'X', 'O', 'X'}};

    char[][] afterBomb = {//char array used to hold a expected game board after bomb explode
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', 'X', '-', '-'},
        {'O', 'O', 'X', 'O', 'X', '-'},
        {'X', 'X', 'X', 'X', 'O', 'X'}};

    int[] bombPos = {4, 2};

    assertEquals(afterBomb, main.bombExplode(bombPos, gameBoardWithBomb));
}

@Test
void testBombExplodeBoundary()
{
    char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'*', 'O', 'O', 'X', 'O', 'X'}};

    char[][] afterBomb = {//char array used to hold game board with bomb location
        {'-', '-', '-', 'X', '-', '-'},
        {'-', '-', 'X', 'O', '-', '-'},
        {'O', '-', 'X', 'X', '-', '-'},
        {'X', 'O', 'X', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'X', 'X', 'O', 'X', 'O', 'X'}};

    int[] bombPos = {5, 0};

    assertEquals(afterBomb, main.bombExplode(bombPos, gameBoardWithBomb));
}

@Test
void testBombExplodeCentre() //test created to test if the bomb explode centre function returns expected
gameboard
{
    char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', '*', 'O', 'X', '-'},
        {'X', 'O', 'O', 'X', 'O', 'X'}};

    char[][] afterBomb = {//char array used to hold game board with bomb location
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', 'X', '-'},
        {'X', '-', '-', '-', 'O', 'X'}};

    int[] bombPos = {4, 2};

    assertEquals(afterBomb, main.bombExplodeCentre(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testCheckIfBombPositionHasChanged() //test created to check if a bomb has been used
{
    int[] bombPos = {3, 4};

    assertTrue(main.getBombPosition(bombPos));
}

@Test
void testCheckIfBombRemainsTrue() //test created to check if bomb remains function returns true when a bomb
is currently on the board at players location
{
    char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'*', 'O', 'O', 'X', 'O', 'X'}};

    int[] bombPos = {5, 0};

    assertTrue(main.checkIfBombRemains(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testBombExplodeBoundaryLowerRow() //test created to check if the bomb positioned at a low boundary
explodes only what is expected
{
    char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},

```

```

        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'X', 'X', 'O', 'X'},
        {'1', '1', '1', '1', '1', '1'}];

char[][] afterBomb = { //char array used to hold game board with bomb location
    {'O', '-', 'O', 'X', '-', '-'},
    {'X', 'O', 'X', 'O', '-', '-'},
    {'O', 'X', 'X', 'X', '-', '-'},
    {'X', 'X', 'O', 'X', '-', '-'},
    {'O', 'O', 'X', 'X', 'O', 'X'},
    {'1', '1', '1', '1', '1', '1'}];

int[] bombPos = {5, 2};

assertArrayEquals(afterBomb, main.bombExplodeBoundaryLowerRow(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testBombExplodeBoundaryUpperRow() //test created to check if the bomb positioned at an upper row explodes
only what is expected
{
    char[][] gameBoardWithBomb = { //char array used to hold game board with bomb location
        {'O', '-', 'X', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    char[][] afterBomb = { //char array used to hold game board with bomb location
        {'O', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    int[] bombPos = {0, 2};

    assertArrayEquals(afterBomb, main.bombExplodeBoundaryUpperRow(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testBombExplodeBoundaryLeftColumn() //test created to check if bomb positioned at left column (column 0)
explodes what is expected
{
    char[][] gameBoardWithBomb = { //char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'X', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    char[][] afterBomb = { //char array used to hold game board with bomb location
        {'O', '-', '-', 'X', '-', '-'},
        {'-', '-', 'X', 'O', '-', '-'},
        {'-', '-', 'X', 'X', '-', '-'},
        {'-', '-', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    int[] bombPos = {2, 0};

    assertArrayEquals(afterBomb, main.bombExplodeBoundaryLeftColumn(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testBombExplodeBoundaryRightColumn() //test created to check if bomb positioned at right column (column 5)
explodes what is expected
{
    char[][] gameBoardWithBomb = { //char array used to hold game board with bomb location
        {'O', '-', 'O', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '*'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    char[][] afterBomb = { //char array used to hold game board with bomb location
        {'O', '-', 'O', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    int[] bombPos = {4, 5};

    assertArrayEquals(afterBomb, main.bombExplodeBoundaryRightColumn(gameBoardWithBomb, 0, 0, bombPos));
}

@Test
void testBombExplodeTopLeftCorner() //test created to check if bomb at top left corner (pos 0, 0) explodes as
expected
{
    char[][] gameBoardWithBomb = { //char array used to hold game board with bomb location
        {'X', 'O', 'O', 'X', '-', '-'},
        {'X', 'O', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', 'O'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

    char[][] afterBomb = { //char array used to hold game board with bomb location
        {'-', '-', 'O', 'X', '-', '-'},
        {'-', '-', 'X', 'O', '-', '-'},
        {'O', 'X', 'X', 'X', '-', '-'},
        {'X', 'X', 'O', 'X', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', 'O'},
        {'O', 'O', 'O', 'X', 'O', 'X'}];

```

```

        int[] bombPos = {0, 0};

        assertEquals(afterBomb, main.bombExplodeTopLeftCorner(gameBoardWithBomb, 0, 0, bombPos));
    }

    @Test
    void testbombExplodeTopRightCorner() //test created to check if bomb at top right corner (pos 0, 5) explodes as
    expected
    {
        char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', 'X', '*'},
            {'X', 'O', 'X', 'O', 'X', 'X'},
            {'O', 'X', 'X', 'X', 'O', 'O'},
            {'X', 'X', 'O', 'X', 'X', 'O'},
            {'O', 'X', 'X', 'O', 'X', 'X'},
            {'O', 'O', 'O', 'X', 'O', 'X'};
        };

        char[][] afterBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', 'O', 'O'},
            {'X', 'X', 'O', 'X', 'X', 'O'},
            {'O', 'X', 'X', 'O', 'X', 'X'},
            {'O', 'O', 'O', 'X', 'O', 'X'};
        };

        int[] bombPos = {0, 5};

        assertEquals(afterBomb, main.bombExplodeTopRightCorner(gameBoardWithBomb, 0, 0, bombPos));
    }

    @Test
    void testbombExplodesBottomLeftCorner() //test created to check if bomb at bottom left corner (pos 5, 0) explodes
    as expected
    {
        char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'O', 'X', 'X', 'O', 'X', '-'},
            {'*', 'O', 'O', 'X', 'O', 'X'};
        };

        char[][] afterBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'-', '-', 'X', 'O', 'X', '-'},
            {'-', '-', 'O', 'X', 'O', 'X'};
        };

        int[] bombPos = {5, 0};

        assertEquals(afterBomb, main.bombExplodeBottomLeftCorner(gameBoardWithBomb, 0, 0, bombPos));
    }

    @Test
    void testbombExplodesBottomRightCorner() //test created to check if bomb at bottom right corner (pos 5, 5)
    explodes as expected
    {
        char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'O', 'X', 'X', 'O', 'X', 'O'},
            {'O', 'O', 'O', 'X', 'O', '*'};
        };

        char[][] afterBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'O', 'X', 'X', 'O', '-', '-'},
            {'O', 'O', 'O', 'X', '-', '-'};
        };

        int[] bombPos = {5, 5};

        assertEquals(afterBomb, main.bombExplodeBottomRightCorner(gameBoardWithBomb, 0, 0, bombPos));
    }

    @Test
    void testbombExplodeBoundarySelect() //test created to check if the correct boundary function is selected
    and performed
    {
        char[][] gameBoardWithBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'O', 'X', 'X', 'O', 'X', 'O'},
            {'O', 'O', 'O', 'X', 'O', '*'};
        };

        char[][] afterBomb = {//char array used to hold game board with bomb location
            {'O', '-', 'O', 'X', '-', '-'},
            {'X', 'O', 'X', 'O', '-', '-'},
            {'O', 'X', 'X', 'X', '-', '-'},
            {'X', 'X', 'O', 'X', '-', '-'},
            {'O', 'X', 'X', 'O', '-', '-'},
            {'O', 'O', 'O', 'X', '-', '-'};
        };

        int[] bombPos = {5, 5};

        assertEquals(afterBomb, main.bombExplodeBoundarySelect(gameBoardWithBomb, bombPos));
    }

    @Test
    void testIfCountersDrop() //test created to test for counter drops
    {
        char[][] gameBoardFloating = {//char array used to hold game board with bomb location
            {'X', 'X', 'X', '-', '-', '-'},
            {'O', '-', 'O', 'O', '-', '-'},
            {'X', 'X', '-', '-', 'X', '-'},
            {'-', '-', '-', '-', 'O', '-'},
            {'O', '-', '-', 'X', '-', '-'};
        };
    }

```

```

        {'O', 'O', '-', '-', '-', '-'],};
    char[][] afterDrop = { //char array used to hold game board with bomb location
        {'-', '-', '-', '-', '-', '-'},
        {'X', '-', '-', '-', '-', '-'},
        {'O', '-', '-', '-', '-', '-'},
        {'X', 'X', '-', '-', '-', '-'},
        {'O', 'X', 'X', 'O', 'X', '-'},
        {'O', 'O', 'O', 'X', 'O', '-'}];

    assertEquals(afterDrop, main.checkFloatingCounters(gameBoardFloating, 0, 0));
}

@Test
void testForDrawGame() //test created to test for a draw game
{
    char[][] gameBoard = { //char array used to hold game board with bomb location
        {'O', 'O', 'X', 'X', 'O', 'X'},
        {'X', 'O', 'O', 'O', 'X', 'X'},
        {'O', 'X', 'O', 'X', 'O', 'O'},
        {'X', 'X', 'O', 'X', 'X', 'O'},
        {'O', 'X', 'X', 'O', 'X', 'X'},
        {'*', 'O', 'O', 'X', 'O', 'X'}];

    assertTrue(main.checkForDraw(false, 0, 0, gameBoard));
}
}

```

Adam Davies - Programming and Testing Assignment - Submission

GRADEMARK REPORT

FINAL GRADE

88/100

GENERAL COMMENTS

Instructor

PART ONE

=====

Engagement with Literature Skills (25) **22**

Knowledge and Understanding Skills (10) **8**

Cognitive and Intellectual Skills (10) **8**

Transferable Skills (5) **4**

Practical Application Skills (50) **46**

Part 1 Software Testing

A good discussion of software testing terminology.

Good coverage of the application of testing at the DVLA.

Provided a nice evaluation of the testing process at DVLA.

Could have been more clearly written in some parts.

Code Part1 Number Sets

a. Display numbers in sets COMPLETE

b. Determine max and min COMPLETE

c: Calculate average COMPLETE

d: Determine overlap COMPLETE

e: Intersection COMPLETE

f: Set difference COMPLETE

Use of JUnit YES

Use of Git YES

Use of Java Doc YES Code executes correctly. Well written and structured code. Some unused variables should be removed. Good use of comments.

PART TWO

=====

Engagement with Literature Skills (25) **14**

Knowledge and Understanding Skills (10) **7**

Cognitive and Intellectual Skills (10) **7**

Transferable Skills (5) **3**

Practical Application Skills (50) **46**

A good number of sources used, but several of them books which are non-seminal and severely out of date or non-credible web sources. At level 7, try to rely more on journal articles and up-to-date books.

Good exploration of the concepts - well-researched and described.

Some errors how sentences are constructed in particular.

Code Part2 Connect 4

Display board COMPLETE

Enter column COMPLETE

Check column position is empty COMPLETE

Drop counter to position COMPLETE

Check win/lose COMPLETE

Blitz feature COMPLETE

Time bomb feature COMPLETE

Use of JUnit YES

Use of Git YES

Use of Java Doc YES

Game executes correctly. Well written and structured code. Good use of comments

Overall Comments

Only one branch used in Git Good testing.

Marks 46/50

Marks Part One: **88**

Marks Part Two: **77**

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10



Comment 1

is required



Comment 2

they are not legally allowed drive their car on the road



Comment 3

How do they enter a number?

Which then begs the question, how to be prevent them entering anything that is not a number?

Can they enter duplicate numbers?

How do we store the numbers?

...and so on.



Comment 4

This problem needs to be defined or sub-divided. What do we mean by overlap....check for identical numbers?

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48



Comment 5

I like this!!

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65

PAGE 66

PAGE 67

PAGE 68

PAGE 69

PAGE 70
