Term Paper

Professional Software Development UC2PSD052

Antonious Gobrial

04.12.2022

Contents

2 Requirements Engineering and System Design

The scenario is the ***bookTrainTicket***, in the use case of an ***International-TrainTravels*** module.

2.1 (a) Functional requirements

1. User will be able to sign up for an app using their personal information. Initially, this will be done via email and password.
2. System must check that the email isn't already registered.
3. System must notify the user of invalid inputs.
4. Users who have created an account will be able to log in, input a destination, and view available routes to the desired location, as well as departure times.

2.2 (b) Non-functional requirements

- Speed is an important product requirement. The system should have the capacity to quickly respond to user commands, and maintain up-to-date data.
- Portability is an example of an organizational requirement, defined by how the system operates in one environment compared to another. The system should be capable of running successfully on all the appropriate operating systems used by users.
- 2 Security is a critical external requirement. Sensitive data must be protected to an appropriate level. Through the implementation of security features, an organization handling data should protect said data in all its states.

2.3 (c) Class diagram

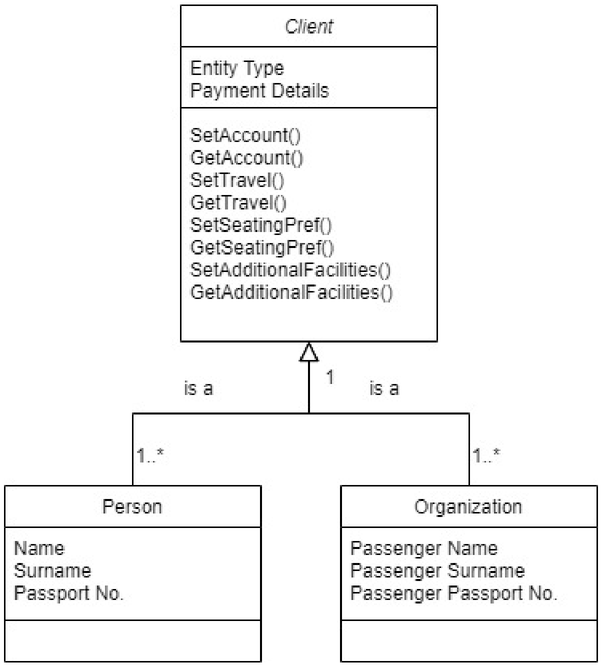


Figure 1: Class diagram showing inheritance

2.4 (d) UML and explanation for *one-to-one* vs *one-to-many* relationships

In a one to one relationship, also notated as 1:1, objects are related to a maximum of one object. Figure 2 is a UML diagram showing an example of a *one-to-one* relationship.

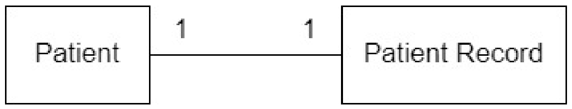


Figure 2: UML 1:1 relationship

In a one to many relationship, also notated as 1..*, objects can be mapped to an unlimited number of other objects. Figure 3 is a UML diagram showing an example of a *one-to-many* relationship.

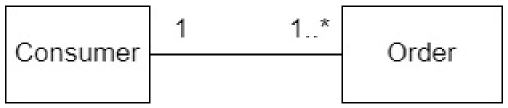


Figure 3: UML 1:M relationship

2.5 (e) Composite use-case diagram for client with five interactions

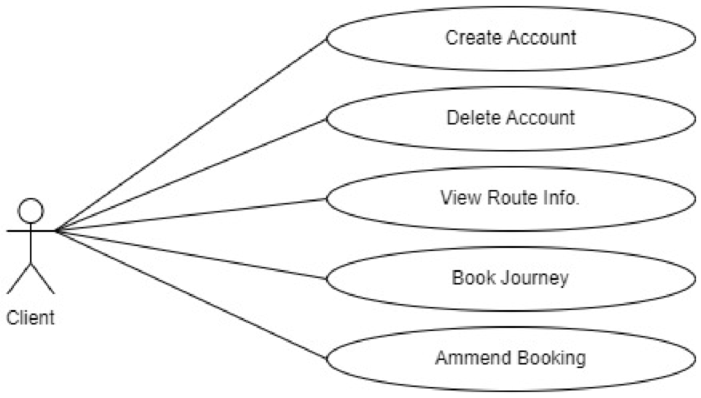


Figure 4: Use-case diagram for client

2.6 (f) UML for *bookTrainTicket*

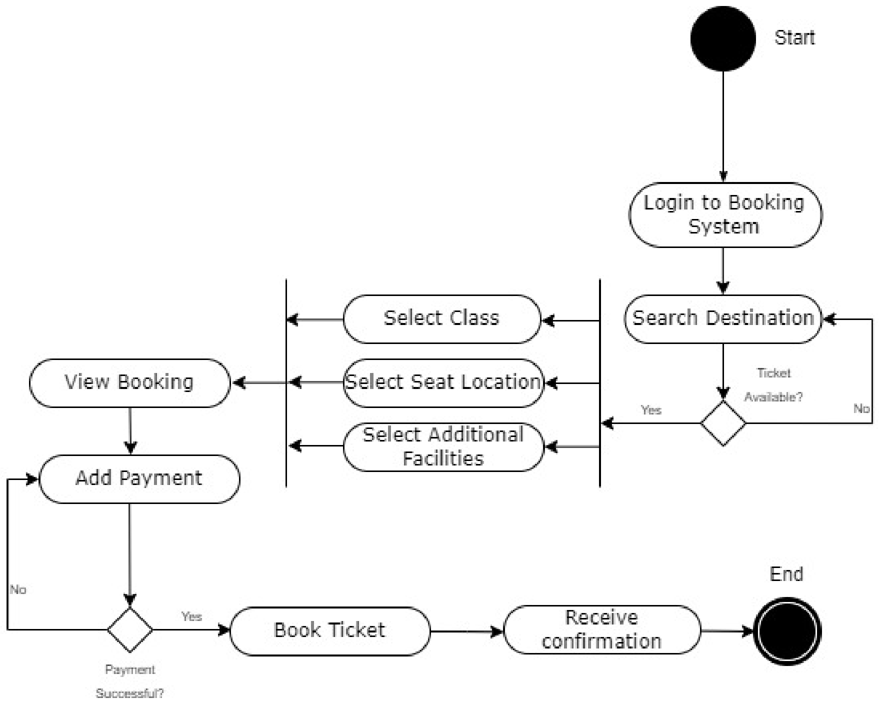


Figure 5: UML diagram for ***bookTrainTicket***

2.7 (g) Sequence diagram depicting client logging into booking system portal

Sequence diagrams are used to show the flow of messages. These are either inputs from actors, or outputs from the objects contained within a system. The message will always have a direction - arrows are used to indicate where the message was initiated, and its destination.

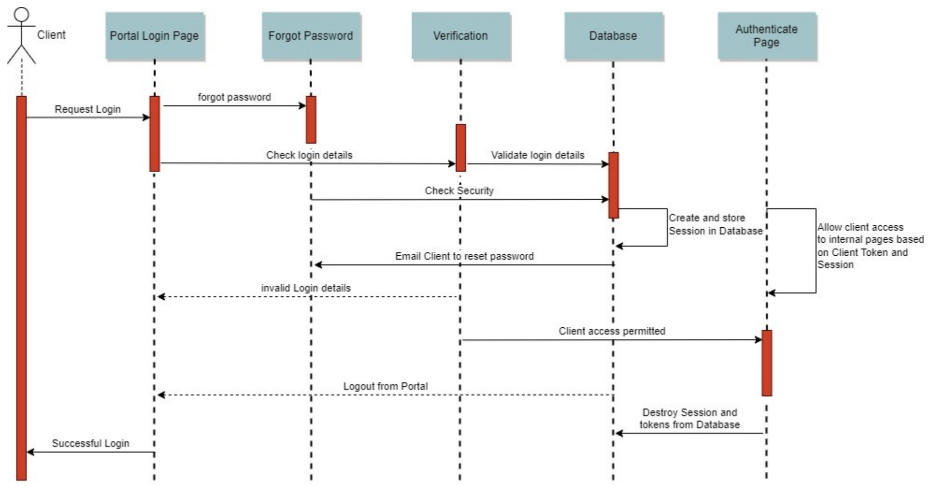


Figure 6: Sequence diagram for booking system portal

3 Software Testing Methodologies

3.1 (a) Scenario Testing

1. **Scenario:** On weekdays, Greg travels to and from his job, in a neighbouring city. His day starts with a 15 minute walk to the train station, then a hour train journey, followed by another 15 minute walk to reach his office. For 5 years Greg has purchased his train ticket at the train station. However, recently there have been many train cancellations which results in Greg being late for work. Greg has decided he needs to know if the train service is operating, before he leaves his home. Greg has decided its best to download and install the Booking app, so he can make sure he's never late for work again.

   Test Case 1

   Objectives: Retrieve up-to-date train information

   Pre-requisites: Live booking system, good internet connection.

   Test Data: Search of desired destination, from current location.

   Test Steps: Input destination, Input time of travel, click on search.

   Expected Results: Journey displayed with correct start and end points, at the correct time of day.

   Test Case 2

   Objectives: Notify client of delayed or cancelled service.

   Pre-requisites: Live booking system, good internet connection, push messages to client permitted, client saves a service to be notified about if it is delayed or cancelled.

   Test Data: Services affected by delay or cancellation, push notification to clients using these services.

   Test Steps: Update service, notify client.

   Expected Results: Customer receives a message when the service they use is affected.

2. **Scenario:** Greg has now installed the app which allows him to book his train to and from work. He has viewed his regular train and saved it as a favourite. He isn't sure whether he wants to pay for the monthly or annual pass but will just pay for his tickets individually. He is also unsure which card to use, so he will decide that when he is purchasing each ticket.

   Test Case 1

   Objectives: Payment is successful.

   Pre-requisites: Valid card, chosen a valid ticket.

   Test Data: Card details, payment verification and receipt.

Test Steps: Fill in card details, click on purchase.

Expected Results: Client gets confirmation of booking and payment, client receives a payment receipt.

Test Case 2

Objectives: Payment is unsuccessful.

Pre-requisites: Payment card which doesn't pass approval, chosen a valid ticket.

Test Data: Card details, payment declined message, method to trigger next attempt.

Test Steps: Fill in card details, click on purchase.

Expected Results: Client receives message of unsuccessful payment, and given the option of another attempt, max 3 attempts.

3. **Scenario:** Greg has been happily using the Booking system for a few months now, so he decides to buy his weeks worth of tickets, on the weekend before. But Greg falls sick on a Monday afternoon and is sent home, his boss tells him to take the rest of the week off. Greg wants to cancel the remaining tickets for that week and get a refund.

   Test Case 1

   Objectives: Cancel tickets and get a refund.

   Pre-requisites: Tickets are refundable, tickets are not within 24 hours of departure time.

   Test Data: Booking details, payment details.

   Test Steps:View booking, cancel booking, receive cancellation confirmation, receive refund.

   Expected Results: Client is able to cancel ticket(s) and will receive a refund to the payment method used.

   Test Case 2

   Objectives: Cancel tickets without a refund.

   Pre-requisites: Tickets are within 24 hours of departure.

   Test Data: Booking details, payment details.

   Test Steps: View booking, confirm cancellation without refund, cancel booking, receive cancellation confirmation.

   Expected Results: Client is able to cancel ticket(s) upon confirming they won't get a refund.

4. **Scenario:** Greg has been doing well in his job and has received a promotion. However, this means will be travelling alot more. On any given

day, he will travel to multiple locations. He doesn't know more than a few days in advance so he will be booking his tickets as he needs them.

Test Case 1

Objectives: Client books multiple tickets at the same time

Pre-requisites: Journey start can be edited, multiple tickets can be added to a basket before purchase.

Test Data: Live booking system, good internet connection, shopping basket.

Test Steps: Client inputs start and end point of a journey, selects ticket, adds ticket to basket, repeats process several times.

Expected Results: Client creates a basket with multiple tickets that snake around the country, and performs one payment action.

Test Case 2

Objectives: Client is warned of booking a ticket which intersects the journey time of another ticket, when attempting to add it to the basket.

Pre-requisites: Basket object is operational, ticket travel time clashes with another tickets travel time.

Test Data: Live booking system, good internet connection, shopping basket, compare current booking to those in the basket.

Test Steps: Client inputs start and end point of a journey, selects ticket, attempts to add ticket to basket, receives warning that ticket clashes with another ticket in the basket, ammends ticket, adds ticket to basket.

Expected Results: Client is warned when attempting to add a ticket which clashes with an existing ticket in the basket. The client can decide whether to continue with adding the ticket to the basket, or to ammend first.

3.2 (b) 2 validation and 2 defective test that can be performed on AC

3.2.1 Validation testing

1. Check if the AC unit is able to reduce the temperature of a room during summer.
2. Check if the AC unit is able to increase the temperature of a room during winter.

3.2.2 Defective testing

1. Check the performance of an AC unit when the coolant level is brought to below the recommended level.

2. Check the performance of an AC unit when the airflow is restricted. Starting at a low restriction, increment towards complete restriction to test for performance of the system as well as error handling carried out by the system to detect such an issue.

3.2.3 Assumptions

- The room is insulated to common building standards, as well as checked that no drafts are present.
- The installation of the unit was carried out by a professional.
- The GUI of the AC will display a temperature, but for comparison, the room temperature should be measured independently of the system.
- The coolant system has no leaks, and the unit is otherwise under normal operation.
- The components involved in air circulation, are all functioning correctly.

3.3 (c) Difference between unit testing, component testing and system testing

The three stages of development testing are *unit testing, component testing,* and *system testing.* All these testing methods are involved in finding defects or bugs in software (Sommerville, 2016).

- **Unit testing** is where the functionality of an individual program unit is tested. An example of unit testing would be isolating part of the login system software, to test the code which checks if an email address is already registered to a client, and how it handles errors.
- **Component testing** is also a type of software testing, however it concerns the usability of each individual component. Components may be entities such as object classes or functions; or even groups of these entities. An example of component testing, to refer to the train booking system, would be to test how data flows between the components of the system which are involved in returning the correct price of a ticket once a client has selected it.
- **System testing** is the testing of the integration of an entire system. It is concerned with showing that a system meets functional and non-functional requirements, as well as testing system properties. This is often a multi-stage process, the subsystems are tested before the integrated system is tested (Sommerville, 2016). An example of system testing, would be performance testing a car prototype that has been manufactured. Initially the components of the car will have undergone unit testing - to establish if each component is functioning as intended, as well as component testing

- how subsystems come together and that they perform as expected. The integration of these subsystems creates the final product i.e. the car. The car will then be tested for performance, reliability, crash-testing and so on. This is system testing.

4 Software Development Life Cycle

4.1 (a) Waterfall vs Agile

The pros and cons of the **waterfall method** are as follows:

Pros:

- Pre-defined objectives result in effective use of time to complete tasks.
- Timescales are pre-defined and kept, to ensure efficiency.
- Testing is simplified using this model.
- The output is accurate as per the requirement.

Cons:

- Difficulty faced when defining the proper needs.
- Lack of flexibility.
- Requires the most time for the delivery of the product.

The pros and cons of the **Agile method** are as follows:

Pros:

- Responds very well to changing objectives.
- Accepts uncertainty.
- Has faster review cycles.
- Highly flexible in releasing features.

Cons:

- Moves forward with a lack of complete understanding of the problem.
- More flexibility can lead to poor behaviour or performance.
- Lacks predictability.

The main difference between the waterfall and the agile methods, can be summed up as follows: the waterfall method is a linear life cycle approach, whereas the agile method is a continuous iteration approach to development and testing in the software development life cycle. The agile method allows more flexibility but in the waterfall their is an structured methodology of the software development.

4.2 (b) Project where Spiral Development Life Cycle Model is preferred over Validation and Verification Life Cycle Model

The spiral model, also referred to as the evolutionary process model, was pioneered by Barry Boehm as an incremental and cyclical approach to the development process. The key factor which makes the spiral model different from other software process models is its explicit recognition of risk. Risk is simply what can go wrong. Each loop is divided into 4 sectors (Sommerville, 2016):

- Objective setting
- Risk assessment reduction
- Development and validation
- Planning

The validation and verification life cycle model (V-model) is made up of its two subcategories: validation testing, and verification testing. Beginning with verification testing, it is the process of analysing predetermined software specifications. These include documents, code, design and programs - to ensure the product meets these specifications. The tester develops a product by carrying out the following activities:

- Identifying business requirements
- Identifying system requirements
- Design Review
- Code walkthrough

Validation testing, also referred to as dynamic testing, entails functional and non-functional testing. It is the process of ensuring that products satisfy the intended purpose, set out by the client. Functional tests include:

- Unit testing
- Integration testing
- System testing

Non-functional testing includes:

- User acceptance testing

Take for example the creation of a new software application like a booking system for a train service. The benefit of a spiral model is that it will allow for the creation of a functional application relatively quickly, so that it can be launched. The same application, developed using the V-model, will require more time before launch due to rigidity in the case of utilization, and a lack of flexibility in terms of design - the V-model by definition doesn't suit projects that aim to release early prototypes.

It is also appropriate to use the spiral model, as it is designed for projects where systems are divided into subsystems that can be developed incrementally, while allowing the core features to remain operational. In this case, requirements are expected to evolve over time, and the application must handle updates while operational to some degree. This renders the V-model unsuitable as it is not designed for small scale updates, rather it is suited for large updates which result in the application being offline.

5 Version Control

5.1 (a) 4 reasons for a version control system

1. Keeping track of changes - who made them, and when they made them.
2. Allowing collaboration between multiple developers
3. Facilitating backups and increasing productivity vs manual version control.
4. Bug fixes are shared - ensures maintenance of different software versions.

5.2 (b) Centralized (CVCS) vs Distributed (DVCS) version control systems

5.2.1 3 advantages of DVCS over CVCS

1. Code base is distributed therefore doesn't have a single point of failure.
2. Branching allows the system to overcome slow network connections.
3. For large team collaboration, CVCS has difficulty finding stable moments to push changes. Hence, DVCS has greater stability.

5.2.2 Examples of centralized and distributed version control systems

- Subversion (SVN) is an example of a CVCS.
- Git is an example of a DVCS.

5.3 (c) Branching in version control systems

5.3.1 What is branching?

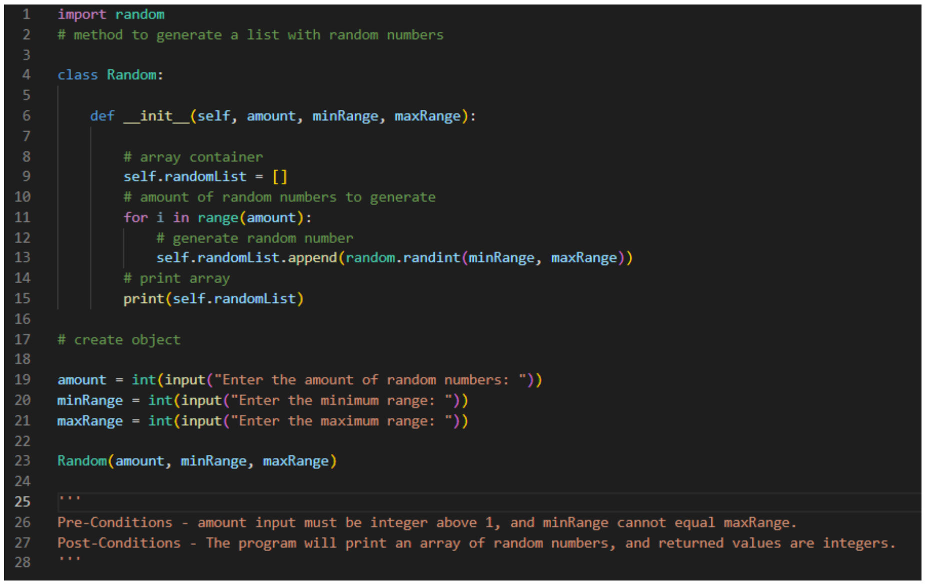
Branching allows developers to work on code at the same time, without duplicating effort. A branch is a copy of a codeline, managed by a version control system (Zolkifli, Ngah, & Deraman, 2018).

5.3.2 3 uses of branching

1. Physical branching: branches created for files, components and subsystems.
2. Functional branching: branches of a systems functional configuration - created for features, logical changes and any unit of deliverable functionality.
3. Organizational branching: branches of a teams work effort - created for groups, roles, tasks and subprojects (Appleton, Berczuk, Cabrera, & Orenstein, 1998).

6 Working with Git

6.1 (a) Random integer generator

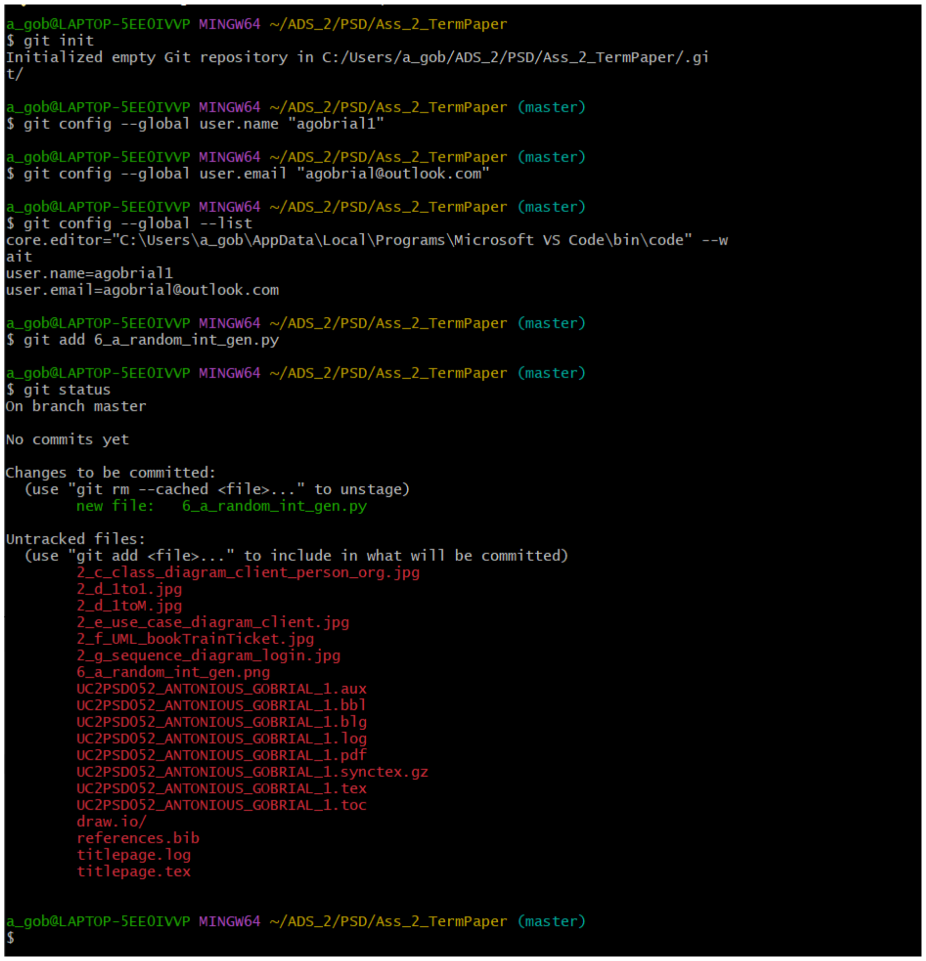


```python
import random
# method to generate a list with random numbers

class Random:

    def __init__(self, amount, minRange, maxRange):

        # array container
        self.randomList = []
        # amount of random numbers to generate
        for i in range(amount):
            # generate random number
            self.randomList.append(random.randint(minRange, maxRange))
        # print array
        print(self.randomList)

# create object

amount = int(input("Enter the amount of random numbers: "))
minRange = int(input("Enter the minimum range: "))
maxRange = int(input("Enter the maximum range: "))

Random(amount, minRange, maxRange)

'''
Pre-Conditions - amount input must be integer above 1, and minRange cannot equal maxRange.
Post-Conditions - The program will print an array of random numbers, and returned values are integers.
'''
```

Figure 7: Screenshot of Random integer generator

6.2 (b) Updating the project 3 times by adding appropriate commit message before updating the existing version

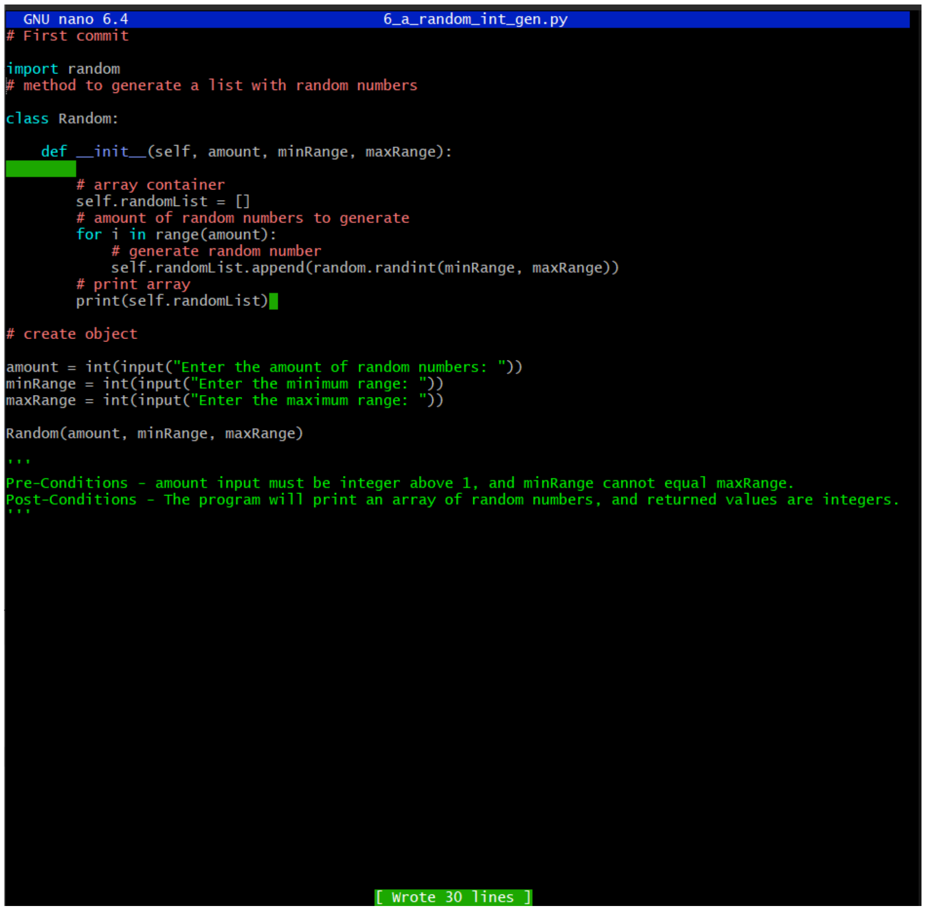
Figure 8 shows the *Git Bash* script for committing the python file to the **staging area**.



```
a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper
$ git init
Initialized empty Git repository in C:/Users/a_gob/ADS_2/PSD/Ass_2_TermPaper/.gi
t/

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git config --global user.name "agobrial1"

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git config --global user.email "agobrial@outlook.com"

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git config --global --list
core.editor="C:\Users\a_gob\AppData\Local\Programs\Microsoft VS Code\bin\code" --w
ait
user.name=agobrial1
user.email=agobrial@outlook.com

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git add 6_a_random_int_gen.py

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   6_a_random_int_gen.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        2_c_class_diagram_client_person_org.jpg
        2_d_1to1.jpg
        2_d_1toM.jpg
        2_e_use_case_diagram_client.jpg
        2_f_UML_bookTrainTicket.jpg
        2_g_sequence_diagram_login.jpg
        6_a_random_int_gen.png
        UC2PSD052_ANTONIOUS_GOBRIAL_1.aux
        UC2PSD052_ANTONIOUS_GOBRIAL_1.bbl
        UC2PSD052_ANTONIOUS_GOBRIAL_1.blg
        UC2PSD052_ANTONIOUS_GOBRIAL_1.log
        UC2PSD052_ANTONIOUS_GOBRIAL_1.pdf
        UC2PSD052_ANTONIOUS_GOBRIAL_1.synctex.gz
        UC2PSD052_ANTONIOUS_GOBRIAL_1.tex
        UC2PSD052_ANTONIOUS_GOBRIAL_1.toc
        draw.io/
        references.bib
        titlepage.log
        titlepage.tex


a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$
```

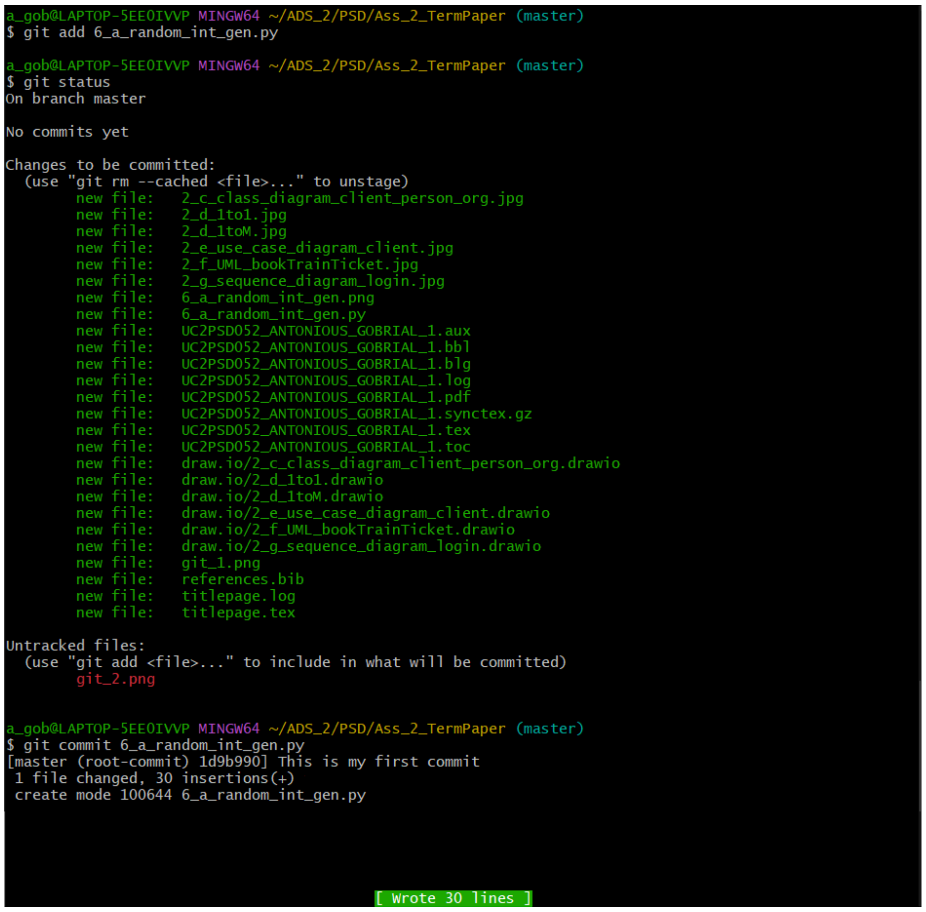
Figure 8: Committing file to the Staging area

The following figures show how the file was committed to the repository.



```
  GNU nano 6.4                              6_a_random_int_gen.py
# First commit

import random
# method to generate a list with random numbers

class Random:

    def __init__(self, amount, minRange, maxRange):

        # array container
        self.randomList = []
        # amount of random numbers to generate
        for i in range(amount):
            # generate random number
            self.randomList.append(random.randint(minRange, maxRange))
        # print array
        print(self.randomList)

# create object

amount = int(input("Enter the amount of random numbers: "))
minRange = int(input("Enter the minimum range: "))
maxRange = int(input("Enter the maximum range: "))

Random(amount, minRange, maxRange)

'''
Pre-Conditions - amount input must be integer above 1, and minRange cannot equal maxRange.
Post-Conditions - The program will print an array of random numbers, and returned values are integers.
'''

                                          [ Wrote 30 lines ]
```

Figure 9: Commenting first modification on the file



```
a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git add 6_a_random_int_gen.py

a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   2_c_class_diagram_client_person_org.jpg
        new file:   2_d_1to1.jpg
        new file:   2_d_1toM.jpg
        new file:   2_e_use_case_diagram_client.jpg
        new file:   2_f_UML_bookTrainTicket.jpg
        new file:   2_g_sequence_diagram_login.jpg
        new file:   6_a_random_int_gen.png
        new file:   6_a_random_int_gen.py
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.aux
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.bbl
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.blg
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.log
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.pdf
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.synctex.gz
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.tex
        new file:   UC2PSD052_ANTONIOUS_GOBRIAL_1.toc
        new file:   draw.io/2_c_class_diagram_client_person_org.drawio
        new file:   draw.io/2_d_1to1.drawio
        new file:   draw.io/2_d_1toM.drawio
        new file:   draw.io/2_e_use_case_diagram_client.drawio
        new file:   draw.io/2_f_UML_bookTrainTicket.drawio
        new file:   draw.io/2_g_sequence_diagram_login.drawio
        new file:   git_1.png
        new file:   references.bib
        new file:   titlepage.log
        new file:   titlepage.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git_2.png


a_gob@LAPTOP-5EE0IVVP MINGW64 ~/ADS_2/PSD/Ass_2_TermPaper (master)
$ git commit 6_a_random_int_gen.py
[master (root-commit) 1d9b990] This is my first commit
 1 file changed, 30 insertions(+)
 create mode 100644 6_a_random_int_gen.py



                                          [ Wrote 30 lines ]
```

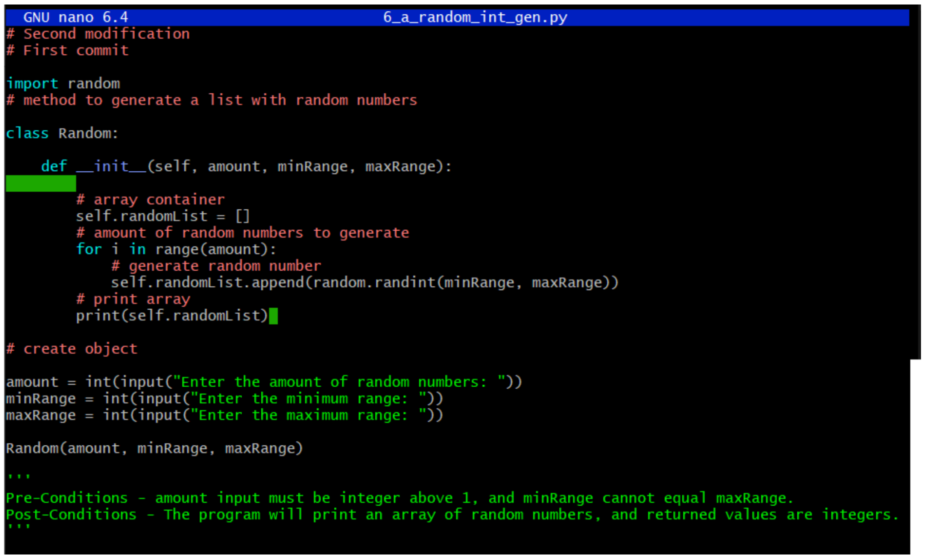
Figure 10: Committing to repository

References

Appleton, B., Berczuk, S., Cabrera, R., & Orenstein, R. (1998). Streamed lines: Branching patterns for parallel software development. In *Proceedings of plop* (Vol. 98, p. 14).

Sommerville, I. (2016). Software engineering 10. *Harlow: Pearson Education Limited*.

Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version control system: A review. *Procedia Computer Science*, *135*.



```
  GNU nano 6.4                             6_a_random_int_gen.py
# Second modification
# First commit

import random
# method to generate a list with random numbers

class Random:

    def __init__(self, amount, minRange, maxRange):

        # array container
        self.randomList = []
        # amount of random numbers to generate
        for i in range(amount):
            # generate random number
            self.randomList.append(random.randint(minRange, maxRange))
        # print array
        print(self.randomList)

# create object

amount = int(input("Enter the amount of random numbers: "))
minRange = int(input("Enter the minimum range: "))
maxRange = int(input("Enter the maximum range: "))

Random(amount, minRange, maxRange)

'''
Pre-Conditions - amount input must be integer above 1, and minRange cannot equal maxRange.
Post-Conditions - The program will print an array of random numbers, and returned values are integers.
'''
```

Figure 11: Second modification