**COS70006**

**Object-Oriented Programming**

**Semester 2 - 2024**

**Learning Reflection Report**

**Name: Arun Ragavendhar Arunachalam Palaniyappan**

**Student ID: 104837257**

**Date: 27/10/2024**

Contents

[Acknowledgment of Country 3](#_Toc180933067)

[Introduction 3](#_Toc180933068)

[Task 1: A self-assessment on the Unit’s Learning Outcomes 3](#_Toc180933069)

[1.1 Principles of Object-Oriented Programming: Abstraction, Encapsulation, Inheritance, and Polymorphism 3](#_Toc180933070)

[1.2 Design, Development, Testing, and Debugging in an IDE 5](#_Toc180933071)

[1.3 Using Collection Classes for Object Management 7](#_Toc180933072)

[1.4 Constructing Diagrams and Descriptions 8](#_Toc180933073)

[1.5 Applying Good Practices 10](#_Toc180933074)

[1.6 Independent Research 11](#_Toc180933075)

[Task 2: Reflections on Learning 12](#_Toc180933076)

[2.1 The Most Important lessons I Learned 12](#_Toc180933077)

[2.2 The Resources and Things That supported and Helped Me Most 12](#_Toc180933078)

[2.3 Challenges I Encountered 13](#_Toc180933079)

[2.4 Key Topics, Concepts, and Tools Mastered 13](#_Toc180933080)

[2.5 My Progress throughout the course of the Unit 13](#_Toc180933081)

[2.6 Applying What I Learned to Future Projects 14](#_Toc180933082)

[2.7 What I Would Do Differently if I Retook the Course 14](#_Toc180933083)

[2.8 Other Comments and Final Reflections 14](#_Toc180933084)

[3. Conclusion 14](#_Toc180933085)

[4. References 15](#_Toc180933086)

# Acknowledgment of Country

I comprehend that Swinburne University of Technology in Melbourne, Australia, is located on land that was traditionally tended by the Wurundjeri People of the Kulin Nation. I have a great deal of respect for the Wurundjeri People's ongoing ties to their ancestral lands, culture, and traditions as a Swinburne student. I want to express my thanks to every one of the university's Aboriginal and Torres Strait Islander employees, students, partners, and guests. Honouring the Wurundjeri People's deep cultural heritage, rich history, and spiritual connection to this country is an honour.

# Introduction

This report captures my experiences and reflections from COS70006, a course focused on mastering Object-Oriented Programming (OOP) concepts using Java. Throughout the unit, I engaged in hands-on projects and weekly tasks that introduced new concepts progressively. The main project involved building a Car Parking Management System, initially developed as a console based application and later upgraded with a graphical user interface (GUI). Each week’s tasks, which included practical exercises and coding challenges, reinforced essential OOP principles like abstraction, encapsulation, inheritance, and polymorphism. These incremental assignments not only deepened my understanding of each concept but also provided the opportunity to apply them in real-world scenarios, resulting in a comprehensive learning experience.

# Task 1: A self-assessment on the Unit’s Learning Outcomes

## 1.1 Principles of Object-Oriented Programming: Abstraction, Encapsulation, Inheritance, and Polymorphism

In the COS70006 course, I focused on mastering four core principles of Object-Oriented Programming (OOP): abstraction, encapsulation, inheritance, and polymorphism. These principles are the backbone of designing modular and maintainable software. Let me break down how each of these concepts was applied in my Car Parking Management System project and in the weekly tasks:

**Abstraction**

* Abstraction allows developers to simplify complex systems by concentrating only on the essential characteristics. In my project, I applied abstraction by creating a base class called Vehicle, which acted as a template for more specific classes like Car and Bike. This approach let me define common attributes and methods for all vehicles, without worrying about unnecessary specifics at this level **(Arnold et al., 2005).**

**Code Example:**

A computer screen shot of a program

Description automatically generated

In the above example, the Vehicle class captures shared properties like registration number and owner name. The displayInfo () method is left abstract, allowing subclasses to implement their specific versions of it. This keeps the core idea consistent while letting each type of vehicle define its own way of displaying information.

**Encapsulation**

* Encapsulation involves bundling data and methods that work on that data into a single unit or class and controlling access to it. I used encapsulation to safeguard attributes like parking slot IDs and occupancy status within the ParkingSlot class. This ensured that these values could not be altered directly and were only accessed or modified through specific methods **(Arnold et al., 2005)**.

**Code Example:**

A screenshot of a computer program

Description automatically generated

Here, direct access to the attributes like slotID or isOccupied is restricted, allowing the class to maintain its integrity and enforce rules consistently.

**Inheritance**

* Inheritance allows the creation of new classes based on existing ones, facilitating code reuse and extending functionalities. In my project, I designed a base class Car and then created specialized classes like ElectricCar and PetrolCar that inherited common properties and behaviours from Car. This not only reduced code redundancy but also enabled polymorphic behaviour **(Horstmann & Cornell, 2013).**

**Code Example:**

A computer screen shot of a program code

Description automatically generated

**Polymorphism**

* Polymorphism is about having a single interface to represent different types of objects. In my project, I used polymorphism by allowing objects of different subclasses to be treated as objects of the base class Vehicle. This approach enabled flexibility in handling multiple types of vehicles without having to write repetitive code **(Horstmann & Cornell, 2013)**.

**Code Example:**

A computer screen with colorful text

Description automatically generated

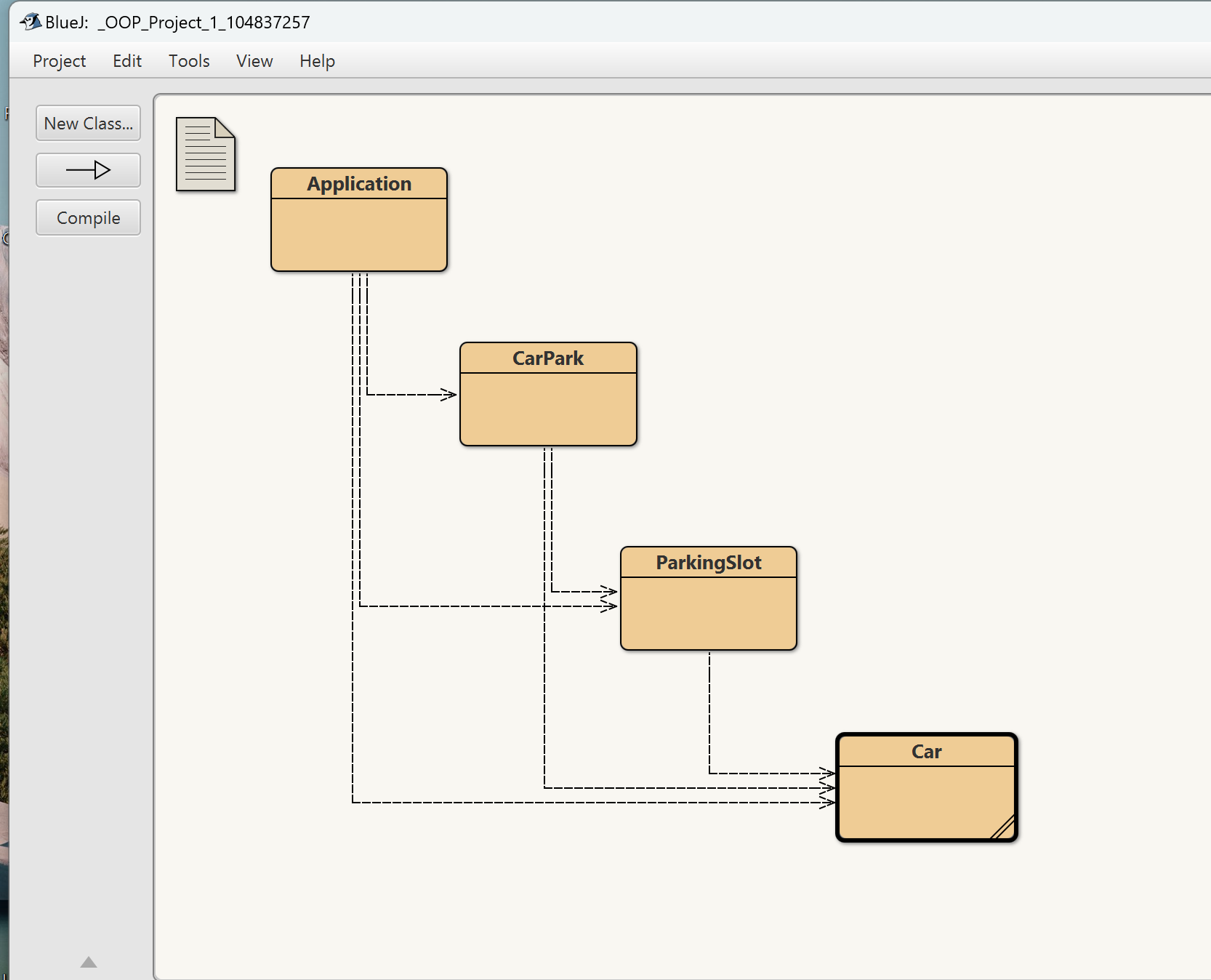
In this scenario, the displayVehicleInfo method uses the Vehicle base class as its parameter, which allows it to accept any subclass of Vehicle and dynamically call the appropriate method based on the object’s actual type. This showcases how polymorphism simplifies code while enhancing flexibility.

## 1.2 Design, Development, Testing, and Debugging in an IDE

Throughout this unit, I relied heavily on the integrated development environment BlueJ to organize, build, test, and refine the system. Using these tools helped streamline the entire process and ensured that my code adhered to key object-oriented principles **(BlueJ Team, n.d.)**.

**Design**

Before writing any code, I focused on carefully planning the project’s structure. I used UML diagrams to visually outline the relationships between different classes, which provided a clear blueprint for the system. These diagrams were crucial in mapping out how classes like Car, ParkingSlot, and CarPark would interact and share responsibilities. Having this visual representation made it easier to identify potential issues or improvements early on **(BlueJ Team, n.d.)**.



**Development**

Once the design was in place, I proceeded with the development phase, taking an incremental approach. Instead of attempting to build the entire system at once, I tackled one feature at a time. For example, I started by implementing basic class definitions and methods, like adding a parking slot or parking a car. BlueJ proved particularly useful in this phase because it allowed me to test individual methods in isolation. This modular approach helped ensure that each feature was functioning correctly before integrating it into the larger system **(BlueJ Team, n.d.)**.

**Testing and Debugging**

* During testing, I created JUnit tests to verify the correctness of critical functions, such as addParkingSlot() and parkCar(). Writing test cases helped me detect and resolve issues early on, while also ensuring that any changes made later wouldn’t break existing functionalities. I also used debugging tools to set breakpoints and step through the code, which allowed me to closely monitor the program’s flow and pinpoint any errors. Additionally, implementing try-catch blocks was crucial in handling runtime exceptions gracefully, especially when dealing with user inputs and file operations **(Gamma & Beck, 2004)**.

**Example JUnit Test:**

**A screen shot of a computer program

Description automatically generated**

By systematically testing and debugging the system using these tools, I was able to maintain a stable and reliable codebase, which ultimately resulted in a well-structured and efficient Car Parking Management System.

## 1.3 Using Collection Classes for Object Management

**A yellow and black text

Description automatically generated**

**(Schildt, 2018)**

One of the most important components of developing scalable software systems is efficiently managing many objects. I made use of the Java Collections Framework in my Car Parking Management System to effectively arrange and retrieve data. I chose some collection classes that provide the best performance and flexibility based on the needs of the system:

**ArrayList**

I decided to store and retrieve parking spaces using an ArrayList. I was able to quickly create, remove, and access parking spaces based on user activities thanks to ArrayList's dynamic nature. Given that the amount of parking spaces may change based on the demands and requirements, this was quite helpful **(Schildt, 2018).**

**HashMap**

I used HashMap to map slot IDs to the corresponding parked autos. This decision made it possible to conduct fundamental tasks like adding, finding, and deleting parked automobiles in continuous time. I was able to keep the unique ID of each parking space as the key and the associated Car object as the value thanks to HashMap's key-value structure **(Schildt, 2018).**

**Code Example**:

A screen shot of a computer program

Description automatically generated

In this code, ArrayList is used to maintain an ordered list of parking slots, while HashMap is employed to efficiently associate each slot ID with a specific car. This combination of collections enables the system to handle dynamic parking requirements while ensuring fast lookups and updates **(Schildt, 2018)**.

By choosing these specific collection classes, I was able to achieve a balance between performance and flexibility, ensuring that the Car Parking Management System could handle various operations efficiently.

## 1.4 Constructing Diagrams and Descriptions

Object-oriented programming relies heavily on visualising and explaining a system's structure. Throughout the development of the Car Parking Management System, I used many types of diagrams to properly portray both the static architecture and the dynamic interactions of the system elements. These visual tools helped me understand how different pieces are linked together and how data flows across the system **(Larman, 2004; McConnell, 2004).**

**Class Diagrams**

A class diagram was necessary for establishing my project's main structure. It highlighted the essential classes—CarPark, ParkingSlot, and Car—and displayed their linkages and interactions **(Larman and McConnell, 2004).**

The diagram depicted the relationships between the system's core classes, which aided my understanding of how to arrange the code properly.

The class diagram clearly shows how classes are related to one another. A CarPark, for example, comprises a list of ParkingSlot objects, each of which can be filled by a Car. This connection is vital for understanding data flow and system functions. The diagram also shows key techniques for managing the parking lot, such as addSlot(), parkCar(), and findByCarRegNo().

**Figure 1: UML Class Diagram for the Car Parking Management System**

**A screenshot of a computer

Description automatically generated**

This diagram showed the relationships between the system's core classes, which helped me understand how to arrange the code properly.

The class diagram clearly shows the links between the classes. For example, a CarPark object comprises a list of ParkingSlot objects, each of which may be occupied by a Car. This link is critical for understanding the flow of data and processes inside the system. The picture also includes crucial methods such as addSlot(), parkCar(), and findByCarRegNo(), which are essential for managing the parking lot.

**Sequence Diagrams**

I made sequence diagrams to show the sequential interactions between objects during processes, such as parking and unparking cars, in addition to the class diagram. Sequence diagrams made it easier to understand how methods and data are transferred at runtime by visualising the dynamic flow of messages across classes **(Larman, 2004; McConnell, 2004).**

**Example Sequence: Parking a Car in a Slot**

**User Input:** The user provides details such as the slot ID and car information.

**System Actions:** The CarPark class receives the parking request and calls the method findBySlotID(), which identifies the target ParkingSlot.

**Method Execution:** If the slot is available, the parkCar() method is invoked on the identified ParkingSlot, associating it with the provided Car object.

**Status Update:** The system updates the status of the ParkingSlot to reflect that it is now occupied.

**Purpose of Using Diagrams**

* These diagrams played a critical role in planning and refining the system’s design. By visually representing the relationships and interactions between classes, I was able to ensure that each component fulfilled its intended role. This approach not only facilitated better organization but also minimized the chances of overlooking key connections or dependencies within the system **(Shan & Humer, 2020)**.

## 1.5 Applying Good Practices

This COS70006 course placed a strong emphasis on the value of following best practices to guarantee the creation of software that is well-structured, scalable, and maintainable. I used the following crucial techniques throughout my Car Parking Management System project:

**Code Modularity**

One of the main principles I followed was keeping the system modular by separating the user interface and the core business logic. This separation adhered to the single-responsibility principle, which ensures that each class or module has a well-defined purpose. For example, I created a dedicated CarPark class responsible for managing parking slots and their associated operations, while the main Application class handled user inputs and outputs. This approach improved the organization of the project and made it easier to test and maintain individual components without affecting the entire system **(Bates & Sierra, 2019).**

**Comments and Documentation**

I carefully documented my code with Javadoc to make it easier to read and re use for future use cases. Each class and method included thorough annotations outlining its functionality and goal. By doing this, I made sure that the code's behaviour and goals would be understood when referred to in the future. For instance, the arguments, return types, and any exceptions that each method in the ParkingSlot class may throw were described in the documentation **(Bates & Sierra, 2019).**

**Example Javadoc Comment for a method:**

A screen shot of a computer

Description automatically generated

**Consistent Naming Conventions**

Following consistent and meaningful naming conventions was another practice that helped maintain the code’s clarity. I named variables and methods descriptively to indicate their purpose. This practice reduced ambiguity and made the code more intuitive, which is especially beneficial when working on larger projects **(Bates & Sierra, 2019).**

**Use of Design Patterns**

During the course, I explored commonly accepted design patterns like the Singleton pattern and Factory Method. Applying these patterns improved the code’s flexibility and robustness. Using a Singleton pattern to manage a single instance of the CarPark class ensured that there was a consistent state throughout the system **(Shan & Humer, 2020).**

**Example Singleton Pattern:**

A screen shot of a computer program

Description automatically generated

**Exception Handling**

Lastly, I implemented proper exception handling to deal with unexpected situations gracefully. By using try-catch blocks and throwing custom exceptions, I was able to provide informative error messages and prevent the program from crashing **(Shan & Humer, 2020).**

**Example Exception Handling:**

A computer screen shot of text

Description automatically generated

Applying these good practices throughout the project helped me create a well-organized, readable, and scalable system, which adheres to the fundamental principles of object-oriented programming.

## 1.6 Independent Research

As part of the COS70006 course, I took the initiative to explore topics beyond the curriculum to deepen my understanding of Object-Oriented Programming (OOP) and improve the structure of my projects. This self-directed learning focused primarily on advanced design patterns and their practical applications.

**Advanced Design Patterns**

During my independent research, I delved into widely used design patterns such as Singleton, Factory Method, and Observer. Understanding these patterns allowed me to design software solutions that were not only more flexible but also easier to maintain and extend **(Shan & Humer, 2020).**

**Singleton Pattern**

I discovered that situations where a single instance of a class is needed to coordinate operations throughout the system make the Singleton design especially helpful. To maintain uniformity across the application and avoid the creation of many conflicting instances, I used the Singleton pattern to handle the CarPark instance in my Car Parking Management System (Shan & Humer, 2020).

**Code Example for Singleton Pattern:**

**A screen shot of a computer program

Description automatically generated**

**Factory Method Pattern**

I also explored the Factory Method pattern, which is often used to create objects without exposing the instantiation logic. This pattern was particularly beneficial when I wanted to abstract the creation of different types of vehicles, allowing for more flexibility in expanding the project later **(Oracle, n.d.).**

**Code Example for Factory Method Pattern:**

**A screen shot of a computer code

Description automatically generated**

This pattern helped in abstracting the instantiation logic, allowing the system to scale easily without modifying existing code.

**Exploring and Reviewing Literature**

Along with studying design patterns, I also looked through several internet resources, such as forums for Java programming and research publications. This independent study gained useful insights and strengthened theoretical understanding. For instance, I learned how to create a loosely connected notification system by reading about the Observer pattern, and I want to use this knowledge in upcoming projects (Oracle, n.d.).

**Practical Application in Projects**

Applying these patterns and concepts in my assignments not only made my implementations more robust but also helped me see the importance of adaptable code design. For instance, implementing the Singleton pattern in my Car Parking Management System ensured that there was only one consistent CarPark instance throughout the program, preventing any conflicts or inconsistencies **(Horstmann, 2021; Sommerville, 2015).**

This independent research not only deepened my understanding of key OOP concepts but also introduced me to more sophisticated design techniques. These insights have prepared me to tackle more complex programming challenges in future projects.

# Task 2: Reflections on Learning

## 2.1 The Most Important lessons I Learned

One of the most valuable lessons I took away from this course was the importance of thoroughly planning and designing before diving into coding. Working on different projects showed me just how crucial it is to invest time in creating a clear blueprint with well-structured diagrams and defined relationships between classes. For instance, in the Car Parking Management System, planning the class structures in advance helped me visualize how various classes would interact, what their attributes would be, and what their roles were. This forward-thinking approach not only made the coding process smoother but also kept my code organized and flexible for future changes. Adopting this mindset has significantly boosted my problem-solving skills and reshaped the way I approach more challenging programming tasks.

## 2.2 The Resources and Things That supported and Helped Me Most

Throughout this course, I found that a few key strategies and resources greatly enriched my learning experience:

* **Hands-On Projects and Step-by-Step Labs**: Building the Car Parking Management System incrementally, starting from a basic console application and evolving it into a full graphical user interface (GUI), provided me with hands-on experience in grasping core Object-Oriented Programming (OOP) concepts. Each stage of development required me to use principles like abstraction, encapsulation, inheritance, and polymorphism. The step-by-step approach of these projects allowed me to refine my work overtime, reinforcing the theoretical knowledge I gained with practical application.
* **Utilizing External Resources**: To enhance my learning, I regularly consulted external resources like the official Java documentation, programming communities like Stack Overflow, and various video tutorials. These resources offered alternative explanations and practical examples that complemented the course material. For instance, exploring Stack Overflow discussions helped me identify and avoid common mistakes in Java Swing when building my project’s GUI. Additionally, diving into tutorials on design patterns broadened my perspective on structuring code more effectively.

## 2.3 Challenges I Encountered

One of the toughest parts of this course was shifting from a basic console application to a more sophisticated graphical user interface (GUI). Building a user-friendly and interactive GUI without relying on drag-and-drop tools meant I had to deeply understand Java’s Swing components, learn how to handle events using ActionListeners, and dynamically manage layouts. Each component came with its own set of difficulties, like arranging the interface elements cohesively with layouts such as BorderLayout and GridBagLayout, and effectively responding to user interactions.

Debugging the GUI was another major challenge. Unlike console-based outputs, errors in the graphical interface often resulted in unexpected behaviours that were harder to trace. To address this, I took a methodical approach to debugging by using print statements, setting breakpoints, and testing individual components one at a time to find and fix the issues.

2.4 Key Topics, Concepts, and Tools Mastered  
Throughout this course, I became highly proficient in fundamental OOP concepts, particularly **encapsulation** and **inheritance**. Encapsulation became almost instinctive as I frequently used private access modifiers to protect internal states and provided public methods for controlled access. Developing the Car Parking Management System also helped me deepen my knowledge of the **Java Collections Framework**. By utilizing **ArrayList** and **HashMap**, I learned to effectively handle collections of objects depending on the system’s requirements, striking a balance between flexible storage, quick access, and efficient lookups.

Additionally, I became skilled at using **JUnit** for testing and for debugging. Writing unit tests allowed me to confirm the correctness of each part of my code, while setting breakpoints in Eclipse helped me trace the program’s execution flow more clearly and identify issues effectively.

## 2.5 My Progress throughout the course of the Unit

Reflecting on this journey, I can see a clear improvement in my skills over time. At the start, I found it challenging to understand how polymorphism worked in practice and struggled with applying interfaces effectively. However, through consistent practice, completing regular assignments, and receiving constructive feedback, my understanding of these concepts gradually deepened by around Week 5. I became more comfortable writing code that was both flexible and reusable, using interfaces to define shared behaviours across different classes.

Transitioning to GUI development, though initially difficult, turned out to be a major growth experience. It broadened my knowledge of building user-friendly and interactive applications with Java Swing. By the end of the course, I felt more prepared to tackle more complex projects that involve combining core logic with user-facing features.

## 2.6 Applying What I Learned to Future Projects

The knowledge and skills I have gained from this course will be directly useful in future projects and my career. Building the Car Parking Management System from the ground up and later upgrading it with a graphical interface gave me a solid foundation in designing modular and scalable applications in any object-oriented programming language. Moreover, mastering concepts like separating business logic from the user interface, using design patterns, and handling user interactions will be essential for creating reliable and user-friendly software solutions in my future work.

## 2.7 What I Would Do Differently if I Retook the Course

If I were to take this course again, I would make a point to dive deeper into challenging topics like GUI development and thoroughly explore design patterns. Utilizing additional resources, such as video tutorials on Java Swing and examples of design patterns in action, would have given me an early advantage in improving my implementations. I would also aim to have more regular feedback sessions with the instructor to clear up doubts sooner and gain valuable insights on best practices.

## 2.8 Other Comments and Final Reflections

Overall, this unit offered a well-rounded and fulfilling learning experience. The blend of theoretical lessons, hands-on projects, and independent research allowed me to develop a complete understanding of Object-Oriented Programming (OOP). Building the Car Parking Management System not only strengthened my technical abilities but also gave me valuable experience in project management, planning, and refining my work through iterative development. I now feel more prepared to take on more complex programming challenges and motivated to keep building on this strong foundation.

3. Conclusion  
Completing the COS70006 course has been a highly enriching and transformative journey that significantly deepened my understanding of Object-Oriented Programming (OOP) principles and improved my proficiency in Java. The course’s structured format, which effectively combined theoretical learning with hands-on practice, allowed me to fully engage with the material. Working on projects like the Car Parking Management System not only solidified my grasp of core OOP concepts but also highlighted the importance of creating scalable and maintainable software solutions.

The course’s focus on planning and documentation was particularly impactful. By regularly creating UML diagrams, providing thorough code comments, and separating business logic from the user interface, I was able to develop well-structured and reliable programs. These practices have ingrained a habit of thoughtful planning and organization, which will be essential as I tackle more advanced projects in the future.

Reflecting on this experience, I can clearly see how much my problem-solving abilities have improved. Navigating challenges such as moving from a console-based application to a fully functional GUI pushed me beyond my comfort zone and encouraged me to explore new programming techniques. This experience has given me the confidence to approach complex problems with a structured mindset and the resilience to overcome technical hurdles.

Looking forward, I am eager to continue refining these skills and applying them to real-world projects. The insights and experience I’ve gained from this course have provided me with a solid foundation, and I feel ready to take on more advanced coursework and career opportunities in software development.

# 4. References

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language* (4th ed.). Addison-Wesley Professional.

Bates, B., & Sierra, K. (2019). *Head first Java* (3rd ed.). O’Reilly Media.

BlueJ Team. (n.d.). *BlueJ tutorial*. BlueJ. <https://www.bluej.org/tutorial/tutorial.pdf>

Coursera. (n.d.). *Object-oriented programming in Java specialization*. <https://www.coursera.org/specializations/object-oriented-programming>

Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

Gamma, E., & Beck, K. (2004). *JUnit pocket guide*. O'Reilly Media.

Horstmann, C. S., & Cornell, G. (2013). *Core Java Volume I—Fundamentals* (9th ed.). Prentice Hall.

Horstmann, C. S. (2021). *Modern Java in action: Lambdas, streams, functional and reactive programming* (3rd ed.). Manning Publications.

Larman, C. (2004). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Prentice Hall.

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

McConnell, S. (2004). *Code complete: A practical handbook of software construction* (2nd ed.). Microsoft Press.

Oracle. (n.d.). *JavaFX documentation*. <https://openjfx.io/javadoc/17/overview-summary.html>

Schildt, H. (2018). *Java: The complete reference* (11th ed.). McGraw-Hill Education.

Shan, Z., & Humer, C. (2020). *Java design patterns*. Packt Publishing.

Sommerville, I. (2015). *Software engineering* (10th ed.). Pearson.

TutorialsPoint. (n.d.). *Java tutorial*. <https://www.tutorialspoint.com/java/index.htm>