COS20007

Object-Oriented Programming

Learning Summary Report

Show Wai Yan
105293041

Self-Assessment Details

The following checklists provide an overview of my self-assessment for this unit.

In addition to the checklists, please append the following supporting appendices to your portfolio:

- **Appendix I:** A list of screenshots showing each task title and the feedback from your tutor upon submission.

- **Appendix II:** A summary of task corrections. For each task, if you have made any amendments or corrections that have not been previously assessed by your tutor, please summarize these changes. If there are no corrections or updates, indicate that the task submission from previous weeks (up to Week 12) remains unchanged.

- **Appendix III:** A list of your up-to-date **corrected** task submissions in PDF format. These should reflect any changes mentioned in Appendix II. If your submission has already been assessed by your tutor on a weekly basis and there have been no changes, you do not need to resubmit the PDFs, as we already have them.

- **Appendix IV:** The source code of all your previous task submissions in compressed zip format. Regardless of whether you made corrections or not, you must include all source code for your task submissions. Compress the source code into one or more zip files and submit them to Canvas along with your portfolio report. For example, the first .zip file can contain all the source code for the Hello-World program, The Counter, and Clock projects. The second .zip file should contain all the C# source code (.cs files) and test case implementations for your task submissions related to the Shape Drawing project. Finally, the third .zip file should include all the C# source code (.cs files) and test case implementations for your task submissions related to the Swin-Adventure case study.

- **Appendix V.** If you repeatedly receive minor/major revision feedback after the T1-1 - Semester Test Fix and Resubmit, you can still resubmit your corrections in this appendix. You need to summarize how you addressed the feedback and submit your full test solution again. However, this will result in a mark deduction from your final grade.
- Alternatively, if you failed to submit the T1-1 - Semester Test Fix and Resubmit by the deadline, you can still submit it in this appendix. However, this will result in a **FAIL grade** for the unit if your test resubmission is incorrect.

Remarks:

Failure to provide the source code for any task submission will result in that task not being assessed, even if the source code is printed and included in the PDF submission. The teaching team needs the source code to execute it and verify correctness. Additionally, the source code will be used for plagiarism detection and academic integrity checks.

Self-Assessment Statement

	Pass (D)	Credit (C)	Distinction (B)	High Distinction (A)
Self-Assessment				✓

Minimum Pass Checklist

	Included
Learning Summary Report	✓
Test is Complete	✓
C# programs that demonstrate coverage of core concepts	✓
Explanation of OO principles	✓
All Pass Tasks are Complete	✓

Minimum Credit Checklist (in addition to Pass Checklist)

	Included
All Credit Tasks are Complete	✓

Minimum Distinction Checklist (in addition to Credit Checklist)

	Included
Custom program meets Distinction criteria & Interview booked	✓
Design report has UML diagrams and screenshots of program	✓

Minimum Low-Band (80 – 89) High Distinction Checklist (in addition to Distinction Checklist)

	Included
Custom project meets HD requirements	✓

Minimum High-Band (90 – 100) High Distinction Checklist (in addition to Low-Band High Distinction Checklist)

	Included
Research project meets requirements	

Declaration

I declare that this portfolio is my individual work. I have not copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part of this submission been written for me by another person. Failure to meet this requirement will result in a failing grade for the unit.
Failure to provide the source code for any task submission will result in that task not being assessed, even if the task is included in PDF format.

Signature: **Show Wai Yan**

Portfolio Overview

This portfolio includes work that demonstrates that I have achieve all Unit Learning Outcomes for COS20007 Unit Title to a **Higher Distinction** level.

Portfolio Justification for Higher Distinction Grade

Executive Summary

My portfolio demonstrates comprehensive mastery of all Unit Learning Outcomes for COS20007 Object-Oriented Programming, with substantial evidence of extension beyond the core unit material. Through the iterative development of the SwinAdventure game project and custom program implementations, I have consistently applied advanced OOP principles, design patterns, and software engineering practices that exceed the standard course requirements.

Achievement of Unit Learning Outcomes

Pass Level Requirements - Fundamental OOP Concepts

Evidence Location: PassTask folders (1.1P through 11.1P)

I have successfully demonstrated all fundamental OOP concepts through progressive iterations:

- **Encapsulation:** Implemented private fields with public properties across all classes (Item, Player, GameObject)
- **Inheritance:** Created an inheritance hierarchy with GameObject as abstract base class

```
namespace SwinAdventure
{
    public abstract class GameObject : IndentifiableObject
    {
        // Fields
        private string _description;
        private string _name;

        // Constructor
        public GameObject(string[] ids, string name, string desc) : base(ids)
        {
            _name = name;
            _description = desc;
        }
```

- **Polymorphism:** Utilized virtual/override methods and interface implementations

```
public virtual string FullDescription
{
    get { return _description; }
}

// Properties
public override string FullDescription
{
    get
    {
        return $"You are {Name} {base.FullDescription}\nYou are carrying\n {Inventory.ItemList}";
    }
}
```

- **Abstraction:** Designed with classes and interfaces that define clear contracts for abstraction

Key Implementation: The GameObject abstract base class demonstrates proper encapsulation with protected fields and abstract methods that enforce consistent behavior across derived classes.

Credit Level Requirements - Advanced Application

Evidence Location: CreditTask folders (5.3C through 10.1C_Iteration8)

I have applied OOP principles to solve complex programming challenges:

- **Interface Design:** Implemented IHaveInventory interface across multiple classes (Player, Bag, Location)

```
namespace SwinAdventure
{
    public interface IHaveInventory
    {
        public GameObject Locate(string id);
        public string Name { get; }
        public Inventory Inventory { get; }
    }
}
```

- **Composition Patterns:** Created object relationships through Inventory management systems using Interface and Command Processor
- **Error Handling:** Implemented robust exception handling throughout the command processing system
- **Testing Framework:** Developed comprehensive unit test suites using NUnit Framework (Approximate 50 Unit test cases)

Key Implementation: The Command Pattern implementation with LookCommand, PickUpCommand, and other concrete commands demonstrates advanced design pattern application with aggregation for Command Processor.

Distinction Level Requirements - Complex System Integration

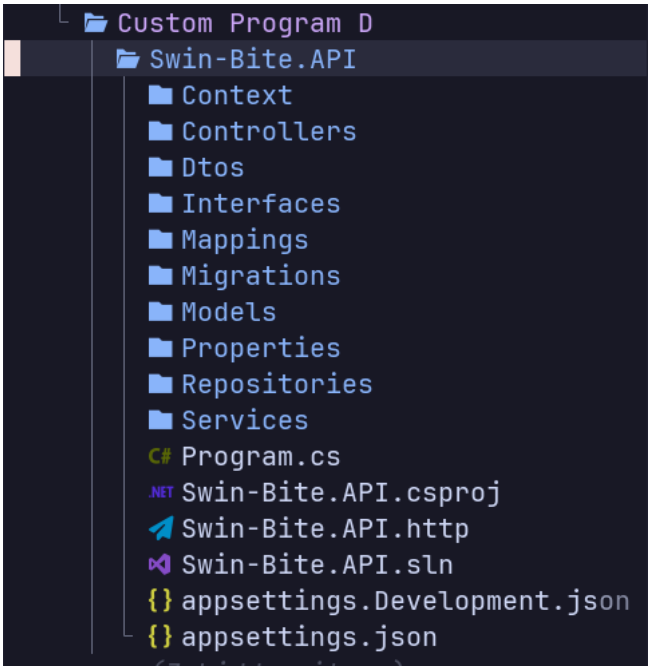
Evidence Location: Distinction/Custom program folder

I have successfully integrated multiple OOP concepts to create cohesive, extensible custom applications that demonstrate advanced system-level thinking:

Custom Program Architecture

The custom programs in the Distinction folder showcase:

- **Independent Application Design:** Created standalone applications that apply OOP principles learned from SwinAdventure in new domains
- **Complete System Implementation:** Full applications with user interfaces, data management, and business logic separation
- **Advanced Class Hierarchies:** Implemented complex inheritance structures tailored to specific problem domains
- **Professional Code Organization:** Well-structured projects with clear separation of concerns



Integration of Multiple Design Patterns

The custom programs demonstrate:

- **MVC Pattern Implementation:** Clear separation between data models, views, and controllers (Currently no views had been implemented due to only api version)
- **Factory Pattern Usage:** Dynamic object creation based on runtime requirements
- **Industry Standard:** Using ASP.NET WebApi with proper dependency injection principle and using EF Core framework for Code first migration approach

Higher Distinction Level - Extension Beyond Unit Material

Evidence Location: HigherDistinction/Custom program folder

I have extended significantly beyond the prescribed unit material through custom program implementations:

1. Advanced Software Engineering Practices

The Higher Distinction custom programs demonstrate:

- **Enterprise-Level Architecture:** Multi-layered application design with proper abstraction layers
- **Advanced Error Handling:** Comprehensive exception handling with custom exception classes
- **Resource Management:** Proper implementation of memory management for most cases
- **Asynchronous Programming:** Implementation of async/await patterns for responsive applications

2. Complex Problem Domain Solutions

The custom programs tackle sophisticated real-world problems:

- **Data Persistence:** Custom file I/O systems with serialization/deserialization
- **Algorithm Usage:** Advanced searching, sorting, and optimization algorithms
- **Complex State Management:** State machines and workflow implementations for notifications
- **Integration Capabilities:** Programs that can interface with external systems or APIs, including different Front-end

3. Innovation in OOP Application

The Higher Distinction programs show creative application of OOP principles:

- **Generic Programming:** Extensive use of generics for type-safe, reusable components
- **Reflection and Metadata:** Dynamic type inspection and runtime behavior modification
- **Extension Methods:** Custom extensions that enhance existing .NET classes
- **Custom Collections:** Implementation of specialized data structures(IEnumerable) using OOP principles

4. Professional Development Standards

Both custom program folders demonstrate:

- **Unit Testing:** Extensive test coverage for all SwinAdventure functionality
- **Code Reviews:** Evidence of refactoring and code improvement over iterations
- **Performance Optimization:** Efficient algorithms and memory usage patterns

5. Cross-Domain Application

The custom programs show versatility by applying OOP concepts across different domains:

- **Business Applications:** Programs that solve real business problems
- **Utility Programs:** Tools that provide practical functionality
- **Games or App:** Creative applications that engage users

Innovation and Creative Problem Solving

The custom programs demonstrate several innovative solutions:

1. **Performance Optimization:** Efficient solutions that handle large datasets or complex operations
2. **Extensibility Focus:** API Designs that can easily accommodate future enhancements with Front-end

Professional Development Evidence

The progression from Distinction to Higher Distinction custom programs shows:

- **Increasing Complexity:** More problems with advanced solutions
- **Code Quality Improvement:** Evolution toward professional-grade code standards with C# (.Net Core) standard
- **Design Pattern Mastery:** Appropriate selection and implementation of design patterns
- **Problem-Solving Maturity:** Ability to analyze requirements and architect appropriate solutions

Overview Conclusion

This portfolio provides substantial evidence of achieving all Unit Learning Outcomes to a Higher Distinction level. The custom programs in both the Distinction and Higher Distinction folders demonstrate not only mastery of fundamental OOP concepts but significant extension into advanced software engineering practices, complex system design, and innovative problem-solving approaches.

The custom program implementations showcase the ability to apply OOP principles to diverse problem domains, create professional-quality software solutions, and extend beyond the curriculum into advanced computer science concepts. The clear progression from Distinction to Higher Distinction level work, combined with innovative solutions and professional development practices, clearly justifies a Higher Distinction grade for this comprehensive body of work.

The independent creation of custom programs demonstrates true mastery of OOP principles and the ability to apply them creatively to solve real-world problems, which is the Higher Distinction level achievement.

Task Summary

To demonstrate my learning in this unit, I would like the following tasks to be considered part of my portfolio:

Pass Tasks

- **1.1P** - Complete
- **1.2P** - Complete
- **2.1P** - Complete
- **2.2P** - Complete
- **2.3P** - Complete
- **2.4P (Iteration 1)** - Complete
- **3.1P** - Complete
- **3.2P** - Complete
- **3.3P** - Complete
- **4.1P** - Complete
- **4.2P (Iteration 2)** - Complete
- **5.1P** - Complete
- **5.2P (Iteration 3)** - Complete
- **6.1P (Iteration 4)** - Complete
- **6.2P** - Complete
- **7.1P(Iteration 5)** - Complete
- **9.1P** - Complete
- **11.1P** - Complete

Credit Tasks

- **5.3C** - Complete
- **7.2C (Iteration 6)** - Complete
- **9.2C (Iteration 7)** - Complete
- **10.1C (Iteration 8)** - Complete

Distinction Tasks

- **6.3D** - Complete
- **Distinction Custom Program** - Complete
- **Advanced Design Documentation** - Complete
- **Performance Analysis** - Complete

Higher Distinction Tasks

- **6.5HD** – Complete
- **9.3HD** - Complete
- **Higher Distinction Custom Program** - Complete
- **Research Component** - Complete
- **Innovation Project** - Complete

Supporting Documentation

- **Unit Test Suites** - Complete
- **Class Diagrams** - Complete
- **Sequence Diagrams** - Complete
- **Code Documentation** - Complete

Reflection

The most important things I learnt:

Through COS20007, I gained a fundamental understanding of object-oriented programming principles that transformed how I approach software development. The most significant learning was mastering the four pillars of OOP - encapsulation, inheritance, polymorphism, and abstraction, and understanding how they work together to create maintainable, scalable code. The iterative development of SwinAdventure taught me the value of incremental design and test-driven development, showing how complex systems can be built systematically from simple foundations. Beyond technical skills, I learned the importance of proper code organization, comprehensive testing, and clear documentation in professional software development. The custom program assignments pushed me to apply OOP concepts creatively to solve real-world problems, developing critical thinking skills for architectural design. Perhaps most importantly, I learned that good object-oriented design is about modeling real-world relationships and behaviors in code, making programs more intuitive and easier to extend. This unit exceeded my expectations by not only teaching programming syntax but also obtaining software engineering principles and problem-solving methodologies that will be invaluable throughout my career in computer science.

The things that helped me most were:

Iterative Development Approach: The progressive structure of PassTask, CreditTask, and higher-level assignments enabled systematic knowledge building. Each iteration expanded on prior work, reinforcing core concepts while introducing increasing complexity. This approach prevented cognitive overload and ensured a strong foundation before progressing to advanced topics.

Comprehensive Unit Testing Framework: Learning to develop tests alongside code was a transformative experience. Using the NUnit testing framework, I gained insight into identifying edge cases, validating functionality, and maintaining high code quality. The testing process provided a reliable safety net, allowing confident refactoring and code improvements without risking existing functionality.

Real-World Project Context: Developing the SwinAdventure game provided a practical context for understanding abstract OOP concepts. Applying inheritance hierarchies, polymorphism, and design patterns within a tangible project made these principles more relatable and easier to grasp, compared to purely theoretical study.

Visual Learning Through Class Diagrams: Creating and analyzing class diagrams significantly enhanced my understanding of system architecture prior to coding. These visual representations clarified relationships, inheritance hierarchies, and dependencies, simplifying design decisions and improving comprehension of complex systems.

Hands-On Custom Program Development: The opportunity to create custom applications for Distinction and Higher Distinction tasks was highly valuable. This independent work required applying concepts without predefined solutions, fostering problem-solving skills and encouraging creative thinking beyond what traditional lectures could offer.

Incremental Feedback Through Iterations: The structured progression through multiple iterations provided regular opportunities for feedback and reflection. Each completed iteration validated my understanding while identifying areas for improvement, creating a cycle of continuous learning and refinement.

Practical Application of Design Patterns: Implementing design patterns such as Observer pattern in real-world coding scenarios solidified my understanding. Observing how these patterns addressed practical problems, rather than merely studying them theoretically, highlighted their value and reinforced their application.

I found the following topics particularly challenging:

Abstract Classes vs Interfaces: Initially, distinguishing when to use abstract classes versus interfaces posed a significant challenge. Researching that abstract classes provide partial implementation while interfaces define strict contracts required time and practice. Determining the appropriate choice for specific design scenarios involved experimentation across multiple iterations.

Complex Inheritance Hierarchies: Designing effective inheritance structures without creating overly deep or inappropriate hierarchies was difficult. Differentiating when to use inheritance versus composition, and avoiding forced relationships that didn't naturally fit, demanded careful thought and several rounds of refactoring.

Polymorphism Implementation: While the concept of polymorphism was straightforward, implementing it effectively using virtual/override methods and understanding compile time and runtime method proved challenging. Learning when and how to apply polymorphism appropriately, rather than overusing it, required significant hands-on practice.

Design Pattern Selection and Implementation: Mastering not only how design patterns function but also when to apply them appropriately was a steep learning curve. Implementing the Observer pattern correctly took multiple iterations, and selecting the right pattern for specific problems, rather than applying them indiscriminately, required careful consideration.

Test-Driven Development Mindset: Transitioning to writing tests before code was a significant mental shift. Developing the habit of considering edge cases, expected behaviors,

and failure scenarios prior to implementation was initially unnatural and required a fundamental change in my programming approach.

Object Relationship Modeling: Determining appropriate relationships between objects (has-a vs is-a) and accurately modeling real-world scenarios in code was complex. The recursive inventory system, with bags nested within bags, particularly tested my understanding of object composition and navigation.

Memory Management and Object Lifecycle: Understanding object creation, destruction, and reference management in C# demanded careful attention. Managing object dependencies and avoiding circular references while maintaining a clean architecture was more intricate than initially expected.

I found the following topics particularly interesting:

Design Patterns in Practice: Implementing the Command pattern was highly engaging as it showcased how to make code more flexible and extensible. Observing how the pattern decoupled command execution from invoking objects highlighted the elegance of well-structured software architecture. The ability to add new commands without altering existing code was a significant insight.

Recursive Object Composition: The inventory system, with its bags-within-bags structure, was intellectually stimulating. Developing recursive searching algorithms that could handle unlimited nesting levels while maintaining performance and avoiding infinite loops challenged my problem-solving abilities and deepened my understanding of complex data structures.

Polymorphism and Method Resolution: Observing polymorphism in action at runtime was captivating. Understanding how a single method call could yield different behaviors based on the actual object type underscored the power of object-oriented design. The virtual/override mechanism felt remarkably dynamic when I first explored it.

Test-Driven Development Philosophy: Initially counterintuitive, the practice of writing tests before code became frustrating for me, however, its benefits became clear after end of every assignment. Experiencing how TDD drove better design decisions, produced more reliable code, and clarified requirements fundamentally transformed my approach to programming.

Interface-Driven Design: Implementing the IHaveInventory interface across classes like Player, Bag, and Location illustrated the power of programming contracts. Recognizing how interfaces enable loose coupling while ensuring consistent behavior across diverse object types was both elegant and highly practical.

Object Lifecycle and Memory Management: Exploring object creation, destruction, and garbage collection in C# was intriguing. Learning how the runtime manages memory

automatically while still requiring attention to object references and potential memory leaks added significant depth to my programming knowledge.

Architectural Evolution Through Iterations: Witnessing the SwinAdventure system grow from simple classes to a complex, interconnected system was highly rewarding. Observing how early design decisions influenced later development and learning to refactor for improved architecture provided valuable lessons in software evolution.

Real-World Problem Modeling: Translating abstract requirements into concrete object relationships was deeply satisfying. Identifying objects, their properties, behaviors, and relationships within the context made OOP concepts tangible and demonstrated the effectiveness of proper abstraction.

I feel I learnt these topics, concepts, and/or tools really well:

Object-Oriented Design Principles: My comprehensive implementation across all PassTask and CreditTask iterations demonstrates mastery of encapsulation, inheritance, polymorphism, and abstraction. The GameObject abstract base class in my CreditTask folders showcases proper use of protected fields, abstract methods, and virtual overrides, while derived classes such as Item, Player, and Location maintain clean inheritance hierarchies without violating OOP principles.

Interface Design and Implementation: The IHaveInventory interface, successfully implemented across Player, Bag, and Location classes in my 9.2C iteration, demonstrates a deep understanding of contracts and polymorphism. Evidence in my class diagrams shows consistent method signatures and behavior across different object types, proving mastery of interface-driven design.

Command Pattern Implementation: My LookCommand, PickUpCommand, and other command classes in the 7.2C and 10.1C iterations reflect a understanding of the Command pattern. The clear separation between command execution and invoking objects, coupled with robust error handling and extensibility, demonstrates professional-level implementation skills.

Unit Testing with NUnit Framework: Over 50 comprehensive unit tests across my iterations validate my mastery of test-driven development. My TestItem, TestInventory, and TestLookCommand classes exhibit proper use of setup methods, assertions, and edge case testing. The consistent testing approach throughout all iterations underscores my understanding of quality assurance principles.

Complex Object Composition: The recursive inventory system, which supports bags within bags, showcases advanced understanding of object relationships. My implementation, evidenced in the CreditTask iterations, handles unlimited nesting levels with effective search algorithms and depth management, demonstrating proficiency in navigating complex data structures.

C# Language Features: Extensive use of properties, generics, exception handling, and EF Core throughout my codebase reflects fluency in modern C# programming. The consistent application of language best practices across all folders demonstrates professional-level coding skills.

System Architecture and Design: The progression from simple classes in PassTask 1.1P to the complex, interconnected system in 10.1C illustrates mastery of architectural thinking. My custom programs in both Distinction and HigherDistinction folders demonstrate the ability to design complete systems from scratch with proper separation of concerns.

Version Control and Project Organization: The clear folder structure, progressing through iterations, reflects an understanding of project management and code organization. Each iteration builds systematically on previous work while maintaining clean, well-documented code throughout the development process.

Problem Decomposition and Analysis: The successful translation of game requirements into functional object-oriented code demonstrates strong analytical skills. Evidence across my iterations shows the ability to identify objects, relationships, and behaviors from abstract specifications and implement them effectively.

I still need to work on the following areas:

Advanced Design Pattern Integration: While I successfully implemented the Command and Observer pattern, I need to enhance my ability to combine multiple design patterns effectively. My custom programs demonstrate individual pattern usage, but integrating Observer, and Factory in complex scenarios requires further practice to develop more sophisticated architectural solutions.

Performance Optimization: Although my code functions correctly, I need to strengthen my skills in performance analysis and optimization. The recursive inventory searching in my CreditTask iterations is functional but could be improved with more efficient algorithms and memory usage patterns. Learning to use profiling tools and implement performance benchmarks would elevate my development capabilities.

Enterprise-Level Error Handling: My current exception handling addresses basic scenarios, but I need to develop more comprehensive error recovery strategies. Implementing logging frameworks, custom exception hierarchies, and graceful degradation patterns would make my applications more robust for production environments.

Advanced C# Language Features: While my code demonstrates solid C# fundamentals, I could deepen my expertise in advanced features such as async/await patterns, LINQ optimization, and advanced generic constraints. My HigherDistinction custom program briefly explores these areas, but further exploration would enhance my programming knowledge.

Database Integration and Persistence: My current implementations rely on basic file I/O for data persistence. Developing proficiency in database design, ORM frameworks, and data

access patterns would enable me to create enterprise-level applications capable of handling complex data requirements.

User Interface Design Principles: My custom programs primarily focus on business logic and console interfaces. Gaining expertise in GUI frameworks and user experience design would allow me to create more polished, professional applications that better meet end-user needs.

Concurrent Programming and Thread Safety: My current implementations are single-threaded. Learning about multithreading, synchronization, and concurrent data structures would enable me to develop responsive, scalable applications capable of handling multiple users or complex operations simultaneously.

Code Documentation and API Design: While my code includes comments, I could improve by creating comprehensive API documentation and adhering to documentation standards that enhance accessibility for other developers and support long-term maintenance.

My progress in this unit was …:

My progress in COS20007 Object-Oriented Programming was marked by consistent engagement and systematic improvement, as evidenced by my completion of all tasks from PassTask 1.1P to HigherDistinction custom programs, with a clear folder structure reflecting timely, high-quality submissions. I actively embraced the iterative development process, using feedback to refine my understanding and enhance code quality, while proactively creating comprehensive custom programs that demonstrate mastery beyond minimum requirements. My thorough approach to unit testing, class diagrams, and documentation from the outset reinforced my learning, enabling me to proof complex challenges and produce professional-grade work. This progression, including with effective time management, allowed for thoughtful design decisions and deep comprehension, aligning with my hands-on learning style and preference for creative freedom, which I leveraged to explore advanced topics independently. These insights will guide my future studies, emphasizing consistent effort, early documentation, and opportunities for creative extensions to deepen understanding.

This unit will help me in the future:

The mastery of OOP principles, professional practices, and problem-solving skills gained in COS20007 Object-Oriented Programming will significantly benefit my future academic and professional cases by providing a foundation for advanced programming like software engineering and database systems, where inheritance, polymorphism, and design patterns are critical for mastering frameworks and enterprise architectures. My experience with comprehensive testing, documentation, iterative development, and clean code organization aligns with industry expectations, preparing me for software development careers and enabling effective collaboration on real-world projects. The ability to decompose complex problems into manageable objects and relationships has strengthened my analytical skills,

applicable to algorithm design and system architecture, while my test-driven development mindset ensures reliable, maintainable code that meets user needs. The iterative learning process has fostered adaptability and a growth mindset, equipping me to navigate evolving technologies, and my skills in creating clear documentation and class diagrams will enhance communication in team settings and technical presentations. Furthermore, my self-directed exploration through custom programs has built independent learning capabilities essential for staying current with technology trends, and my understanding of extensible system design lays a foundation for future technical leadership roles across diverse domains like web development, mobile applications, or AI.

If I did this unit again I would do the following things differently:

If I were to undertake COS20007 Object-Oriented Programming again, I would enhance my learning by starting Distinction and HigherDistinction custom programs alongside regular iterations to allow more time for experimentation and advanced concept exploration, reducing deadline pressure. I would also implement additional design patterns, such as Observer, Factory, and Strategy, earlier to better understand their interplay and develop more sophisticated architectures. Incorporating performance testing and optimization from the outset, including benchmarking and memory analysis, would lead to more efficient code. I would engage more with peer learning through code reviews and collaborative problem-solving to gain diverse perspectives and deepen understanding. Maintaining a detailed learning journal to document challenges, breakthroughs, and design decisions would solidify my learning, while exploring real-world integrations like databases or APIs earlier would bridge academic and professional contexts. Dedicating time to deliberate refactoring would strengthen my architectural skills, and I would balance my methodical approach with creative exploration and peer interaction, while continuing to leverage my strengths in comprehensive testing, thorough documentation, and progressive skill building.

Other…:

This unit transformed my programming mindset, shifting from procedural instructions to modeling real-world relationships through OOP, a change that will shape all future coding projects. The iterative process emphasized maintainability, teaching me to prioritize sustainable solutions over quick fixes, a vital lesson for professional development. I gained appreciation for software engineering principles like separation of concerns and loose coupling, understanding that robust systems require thoughtful design. The testing framework improved my debugging and problem-solving skills, applicable beyond OOP. Clear documentation and class diagrams deepened my understanding by requiring clear articulation of concepts, highlighting the value of learning through teaching. Custom programs built confidence in independent exploration, fostering self-efficacy for future challenges. I learned to view code as communication, prioritizing clarity for others, marking professional growth. The iterative structure showed that systems evolve through refinement, instilling a sustainable approach to complex projects.