Study Journal

Final Report

Lucas George Sipos

Student ID: 24292215

A study journal in part fulfilment of module of

COMP47480 Contemporary Software Development

Module Coordinator: Associate Professor Mel Ó Cinnéide

UCD School of Computer Science
University College Dublin

April 24, 2025

Table of Contents

Chapter 1: Agile Principles

In this lab session, we came up with a team of three to talk about Agile principles that needed to be pointed out all over the three parts. The primary focus was on understanding the principle by working on to figure it out on our own, along with eight criteria that would cover each of the twelve Agile principles in an iterative and collaborative approach.

1.1 Work Done

During the lab, our team worked closely together to come up with ideas for the following task:

Part 1: Brainstorming

We discussed and identified key criteria that contribute to the task, focusing on the role of a development team. Drawing from our UCD studies and other prior experiences, we've managed to come up with 15 ideas. Some of the key factors included effective communication, and clear role distribution, meaning everyone had to come up with at least 3 or 4 criteria. Feedback from classmates if a criterion wasn't as suited as everyone would want to, adaptability to changing good ideas into slightly better ones, and efficient time management, even though it was the most challenging part of this laboratory, at least for our team.

Part 2: Select Eight Key Criteria

Once we had our full list of success criteria, we carefully narrowed it down to the eight most important ones. Everyone shared their thoughts on what really makes a software project successful, and the conversation was insightful. Key elements like "clear coding standards", "testing", "CI/CD", and "documentation" were generally agreed upon, but certain areas caused controversy, particularly the need for finding a balance between covering a large ground and having the perfect combination of criteria. In the end, we came to an agreement, to make sure our choice represented both our experiences and our personal thoughts on this subject.

Part 3: Are Your Criteria Agile, and What Did You Miss?

After selecting our eight key criteria, we assessed their alignment with the 12 Agile Principles. Each criterion was carefully mapped to the relevant principles, ensuring that they supported Agile methodologies in practice. Most principles were well-covered, but there was one instance where only a single criterion aligned with a specific principle. This required a deeper discussion to explain its relevance and justify our selection.

Since every concept was taken into consideration, we concentrated on explaining how each criterion matched Agile values. We also considered the important aspects of these decisions, which increased our own understanding of Agile's fundamental principles. The process itself was accurate, and the conversation improved our justification for selecting each criterion.

1.2 Reflections

This lab session offered insightful information about Agile principles and how they are applied in the real world. We were able to understand the importance of iterative collaboration, particularly during the brainstorming and selection stages, thanks to the practical approach. One important lesson learned was how productive teamwork and organized dialogue helped us improve our concepts and create a carefully analyzed set of success criteria.

In the long run, we could improve on multiple aspects of our well-balanced decision-making process. An alternative approach would have been to set aside less time to meeting the list of "perfect" criteria. Although careful consideration ensured the level of quality of our choices, it also resulted in time-consuming discussions that could have been shortened. In the future, choosing a more iterative approach, where we continuously improve criteria instead of searching for the immediate perfect solution, might have been more successful. Agile itself promotes adaptability and incremental development.

In addition, there were also strong connections throughout this experiment and other areas of software engineering, particularly Software Development Methodologies and User-Centered Design (UCD). Agile's emphasis on teamwork and iterative feedback loops reflects important UCD ideas, proving that software is about constant change based on stakeholder and user demands rather than just generating code. Similar resemblances have been found between talks of criteria such as CI/CD and documentation and real-world engineering methodologies, where finding the right combination of maintainability and agility is a crucial task.

Furthermore, it was really intriguing when we related our criteria to Agile principles. It made us reconsider whether we might have overlooked certain details or if our decisions actually reflected Agile values. Because it forced us to consider the concept's significance and provide justification for our choice, the conversation regarding the principle with only one matching condition turned out to be intriguing.

Overall, this exercise strengthened our understanding of Agile and its practical implications. It highlighted the importance of structured decision-making, teamwork, and reflection in software development skills that will be essential in our future careers. The challenges we faced, such as differing opinions, time constraints, and balancing broad coverage with specificity, are common in professional software development, making this a valuable learning experience beyond the classroom setting.

Chapter 2: Podcast with Karl McCabe

Karl McCabe's journey in software engineering is a fascinating case study of how technology evolves in high-scale environments like Amazon and Meta [All23]. His discussion touches on key themes that resonate deeply with anyone in software development, particularly around scaling, ownership, and balancing speed with quality.

2.1 What resonated with me

One of the main points that popped up out of this podcast was how crucial it is to develop software fast, even if doing so short-term ends up in a chaotic system. Karl McCabe described how teams had been encouraged by both Amazon and Meta to allow themselves loose by creating quick solutions, prioritizing functionality in front of long-term architectural perfection. This resonated with me because it captures an essential trade-off that many businesses must make: establishing a balance between maintainability and speed.

The ownership culture of these large organizations was another important aspect, in my personal view. One effective strategy that guarantees accountability and reliability is the concept that engineers are in charge of monitoring their code from development to production. Because engineers are the ones who will have the responsibility of fixing the code if something goes wrong, this paradigm forces developers to develop more robust code, as revealed by Karl's example of engineers being on-call for their own services. I believe that more companies should use this strategy since it fits in neatly with the DevOps mentality.

Another highlight for me was the conversation around incident reviews and a no-blame culture. Engineers have the opportunity to grow and improve in a safe atmosphere when failures are analyzed objectively and not as an opportunity of placing blame. These companies manage large-scale software in an original and efficient way through utilizing failures as opportunities to improve their processes and systems instead of punishing mistakes.

At last, the information regarding the evolution of software quality over time was extremely valuable. Karl pointed out that although quality is frequently a secondary issue in the early stages of a company's development, architectural discipline becomes necessary as the systems become more complex. It is obvious from this natural process why companies need adjustments to their strategy as they expand.

2.2 What surprised me

One of the most surprising aspects of this discussion was the lack of a single, structured development methodology across teams. I had assumed that companies as large as Amazon and Meta would enforce a standardized development framework, but Karl's explanation showed that teams are remarkably independent in choosing their own methodologies. Some teams use Agile approaches like Scrum or Kanban, while others operate with minimal formal processes. Teams can personalize what they produce to meet their particular needs because to this level of independence, but it also indicates that different divisions inside of the company might develop software in very different ways.

Another surprising point was Karl's description of how Amazon initially rejected AWS EC2 as an internal side project. He explained that the EC2 team had to build their own solutions because they couldn't get support from the rest of the company. Even though AWS initially started out as a weird experiment, it is currently one of Amazon's largest and most valuable businesses. This demonstrates how successful companies frequently begin as small, independent initiatives and how innovative thinking can occasionally originate from places that are not expected.

I was also surprised by the differences in culture between Amazon and Meta. While both are programming-driven companies, Amazon has a more top-down, structured management style, whereas Meta operates in a more engineer-driven, primarily distributed. The fact that Meta allows engineers to modify each other's code freely was unexpected, as most companies have strict ownership rules to prevent unnecessary changes. This collaborative approach can be beneficial for fostering innovation, but it also introduces risks in terms of consistency and accountability.

Lastly, the discussion on code freezes and manual control in large-scale deployments was unexpected. In theory, continuous integration and deployment (CI/CD) pipelines should make software releases seamless, but Karl explained that companies still enforce code freezes during critical periods, such as Black Friday or New Year's Eve, to minimize risk. This suggests that despite advancements in automation, human monitoring is still a crucial factor in high-stakes environments.

Chapter 3: Test-Driven Development

We implemented Conway's Game of Life [Wika] using strict test-driven development, creating functionality for cell evolution while maintaining commit discipline with RED/GREEN/REFACTOR markers.

3.1 Work Done

We implemented:

1. Grid initialization and current generation retrieval
2. Neighbour counting logic with Manhattan distance checks
3. Next generation computation implementing survival/revival rules
4. Custom equality check for game states

Test cases covered:

1. Empty grid persistence across generations
2. Single-cell underpopulation (Rule 1)
3. Stable 2x2 block preservation (Rule 3)
4. Pattern evolution with mixed neighbour counts (Rule 2/4)

Commits followed RED-GREEN-REFACTOR pattern with 18 atomic commits showing TDD progression, granulating the work as much as possible.

3.2 Reflections

3.2.1 Code Coverage Insights

Using IntelliJ's coverage tool revealed:

- **Line Percentage** (100%) – we've covered every single line in the *GameOfLife* class

- **Branch Percentage** (80%) – even though it is not 100%, the objective is to get what we need from the branch percentage, meaning that only relevant branches would need to be verified. This code `return aliveNeighbors > 1 && aliveNeighbors < 4;` was not verified for `aliveNeighbors < 4`, which isn't a big deal because it's stated in the rules of the game that the cell with 1 to 3 neighbours will still be alive in the next generation, else it would be dead. We know that covering the `aliveNeighbors > 1` is correct, which concludes that the other branch is also correct. The other percentage comes from *overriding equals* built-in method (Listing 3.1) which is a widely known template for this *override*, so there is no need to test it.

Listing 3.1: Equals function

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    GameOfLife other = (GameOfLife) obj; // covered
    if (gridSize != other.gridSize) return false;
    return new HashSet<>(this.liveCells).equals(new
       HashSet<>(other.liveCells)); // covered
}
```

- **Cell class** – in this class, we override 3 methods (Listing 3.2) so that:

  1. we determine if 2 objects are logically equal (helped with object comparison and when collections use method to determine if objects are equal)
  2. we return an integer hash for an object (made *HashSet* more efficient and also 2 objects are equal if they have the same hash)
  3. we return a string representation of the object for better visualisation (improved readability and debugging speed)

Listing 3.2: Cell class additions

```java
@Override
public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()) return false;
    Cell cell = (Cell) o;
    return x == cell.x && y == cell.y;
}

@Override
public int hashCode() {
    return Objects.hash(x, y);
}

@Override
public String toString() {
    return x + ":" + y;
}
```

3.2.2 AI Tests

The AI-generated tests proved both valuable and flawed. While it expanded our test coverage, it also highlighted differences between human and machine testing approaches:

- **Overkill and Noise** – the AI's tests were redundant, like rechecking empty grids, or overly specific, such as directly testing the *countAliveNeighbors* method using *Reflection API*. This violated TDD's principle of testing public behaviour rather than internals. It even generated tests that passed accidentally; it asserted that cells revived with "exactly three live neighbours," but our code allowed resurrection with three total neighbours (excluding dead cells), meaning it revived a cell even if it had 2 neighbours because the 3 would be the dead cell itself.
- **Branch Coverage Percentage** – AI tests achieved a higher branch coverage (98% vs. our 80%) but felt less purposeful. For example, parameterised tests for all neighbour counts 0–8 forced us to handle edge cases like `neighbours = 4`, which we'd overlooked. However, these tests felt mechanical compared to our behaviour-focused ones, because the AI also tested the *equals* method for almost every branch which isn't very informative. The conclusion is that AI complements but doesn't replace human judgment; it's great for filling gaps but risks testing for coverage metrics rather than meaningful scenarios.

3.2.3 TDD comments

Strict adherence to TDD shaped both our successes and blind spots:

- **Emergent Simplicity** – the neighbour-counting logic evolved naturally through not having the time to think about a more efficient solution and getting things done as fast and as simple as possible. We started with a brute-force approach, an inefficient Moore neighbourhood [Wikc], but tests for larger grids and a small number of alive cells exposed performance issues. This led to a streamlined version still using Moore neighbourhood but improved it a little bit by removing some unnecessary iterations for the cell removal, which passed all tests while being more efficient. TDD prevented overengineering; we only optimized when tests demanded it. The worst part about it is that there are major improvements that could be made for better efficiency, such as for reviving cells, only looking for alive neighbours if the dead cell is an immediate neighbour of the alive cell.
- **The Edge Case Dilemma** – Our TDD process focused on "happy paths" defined by Conway's rules but could've missed boundary conditions or as the AI stated the "blinker" test (for recursive pattern change over generation).

3.2.4 Conclusion

Testing was shown to be a multi-layered process by the exercise. Coverage tools point to blind spots, TDD gives guidance, and AI tests bring computational robustness. When combined, they are almost like an Ensemble learning [Wikb] approach in which distinct error classes are caught by different layers. Although no strategy is flawless, this approach was useful for getting things done in a short time. I'll use this lesson in academic challenges.

Chapter 4: Mutation & Mocking

This report presents the findings and reflections on the second software testing lab, which focused on mutation testing and mocking. The laboratory task aimed to evaluate the robustness of unit tests and understand how to isolate class dependencies using mocking frameworks.

4.1 Work Done

During the lab, two primary areas were explored: mutation testing and mocking. The initial phase involved using mutation testing to evaluate the strength of our unit tests in detecting bugs. This was achieved by introducing small changes (mutations) into the *GameOfLife* class and running unit tests to verify if the changes were detected.

Manual mutation testing was first performed by altering the loop boundaries within the *GameOfLife* class. Specifically, the nested loop in lines 42 and 43 was modified, which iterates over the grid size. The original code was:

Listing 4.1: Initial code

```java
42 for (int i = 0; i < gridSize; i++) {
43     for (int j = 0; j < gridSize; j++) {
```

Upon performing the manual mutation testing, it was discovered that our test cases did not catch the error when the grid size was altered.

Initially, we refactored the code by adding a conditional `if` statement to handle potential out-of-bounds conditions. However, we discovered that the `if` condition itself could be mutated in a way that our tests would fail to detect the resulting bug, leading to the *"survival"* of the boundary mutation.

The use of PIT [Hen] further automated the mutation testing process. PIT generated reports that identified surviving mutations and highlighted areas where the unit tests needed a little improvement.

Consequently, we refactored the code by introducing two helper methods: *isWithinAxis* and *isWithinBounds*, which provided a reusable boundary check.

Listing 4.2: Helper methods

```java
private boolean isWithinAxis(int n) {
    return n >= 0 && n < gridSize;
}

private boolean isWithinBounds(Cell cell) {
    return isWithinAxis(cell.getX()) && isWithinAxis(cell.getY());
}
```

The `for` loops were then revised, incorporating the following validation:

Listing 4.3: Modified code

```java
for (int i = 0; isWithinAxis(i); i++) {
    for (int j = 0; isWithinAxis(j); j++) {
```

The final PIT results indicated that all significant mutations were successfully killed, except for a timeout mutation related to the *isWithinAxis* method, which was expected due to the infinite loop created by the mutation:

`replaced boolean return with true for isWithinAxis -> TIMED_OUT`

In the second phase, Mockito was used to perform mocking tests on the *Ballot* class. The *Ballot* class was tested against different voting scenarios by simulating *Voter* behavior using Mockito's `mock()`, `when()`, and `verify()` methods. Three primary scenarios were covered:

- Majority votes YES, resulting in `Result.YES`
- Majority votes NO, resulting in `Result.NO`
- Equal number of YES and NO votes, resulting in `Result.DRAW`

The *Mockito* framework allowed us to isolate the logic of the *Ballot* class and verify that the correct result was communicated to all voters.

4.2 Reflections

This lab provided valuable insights into mutation testing and mocking. The manual mutation testing phase highlighted the importance of boundary checks and demonstrated how minor changes in code can expose weaknesses in unit tests. The automated approach using PIT proved highly effective in systematically identifying test suite gaps. In hindsight, the introduction of helper methods such as *isWithinAxis* improved code reusability and maintainability. An alternative approach could involve parameterised tests to cover a broader range of boundary values, further strengthening the test suite.

Mocking with *Mockito* was particularly interesting because it allowed us to decouple class dependencies and test main logic needed independently. This approach could be applied and it may be recommended to other projects where external services or database connections need to be simulated.

The practical exercise also reinforced the importance of comprehensive unit tests in software development. By integrating both mutation testing and mocking, the overall quality and reliability of the codebase was significantly improved. Furthermore, the techniques learned in this lab are closely related to other software testing methodologies, such as test-driven development (TDD) and integration testing. These concepts will be particularly useful in future projects where verifying software correctness is critical.

In conclusion, this lab exercise deepened our understanding of mutation testing and mocking, while emphasizing the need for robust unit tests in software projects. The combination of manual and automated techniques provided a comprehensive evaluation of the code's correctness and strength.

Chapter 5: Podcast with Roland Tritsch

Roland Tritsch, who spoke to our class [All25], offered a unique perspective on software engineering. As the sound engineer, I was particularly engaged. This chapter highlights the key insights I gained from his talk, covering craftsmanship, learning, and AI, and how they challenged my understanding of the field.

5.1 What Resonated with Me

The conversation with Roland Tritsch highlighted the importance of craftsmanship in software engineering, a perspective that truly resonated with me. His analogy comparing software craftsmanship to traditional crafts like carpentry was particularly accurate to today's world. The idea that a software engineer should take pride in their work to the extent of metaphorically signing their name on it encourages a sense of ownership and quality that is often overlooked in fast-paced development environments. The emphasis on maintainability and the cost of ownership was something that stood out to me. Tritsch's perspective that writing code is only the beginning of a system's lifecycle is a powerful reminder of the long-term responsibility software engineers have. The explanation of how functional programming, with its immutability and side effect free functions, leads to more testable and maintainable systems reinforced the value of thoughtful design. This approach aligns with the broader goal of reducing technical debt and building systems that can evolve without excessive overhead.

Additionally, his insight on learning organizations struck a chord. The notion that successful engineering teams are fundamentally learning communities highlights the importance of continuous growth and collaboration. This perspective encourages not just individual learning, but the sharing of knowledge across the team to avoid barrier knowledge and foster collective improvement. Tritsch's emphasis on fostering an environment where knowledge is continuously exchanged and refined aligns with the belief that great software is built by great teams, not just great individuals.

Moreover, the emphasis on software craftsmanship as a mindset rather than a set of rigid practices was inspiring. It suggested that the path to becoming a better software engineer is not solely about mastering tools and languages, but about cultivating an attitude of care, attention to detail, and pride in one's work. This broader view of craftsmanship encourages a holistic approach to software development, where technical excellence is balanced with empathy for users and consideration for long-term sustainability.

5.2 What Surprised Me

The discussion's strong support for pair programming was one unexpected finding. Tritsch highlighted pair programming as a crucial tool for promoting cooperation, knowledge transfer, and better code quality, despite the fact that it is frequently perceived as a specialized activity. Given that pair programming is not commonly used in many firms, his recommendation to set aside at least an hour each day for it was surprising. I reexamined the practice's possible benefits after learning that pair programming fosters a culture of community ownership and aids in dismantling knowledge silos.

Another surprising point was the emphasis on code coverage as both a quality metric and a cultural tool. Tritsch's view that code coverage should not just verify testing completeness but also encourage better engineering practices provided a new perspective on how automated tools can shape team behaviour. The mention of using code coverage tools to identify redundant tests and optimize the test suite added a practical dimension to what is often seen as a purely quantitative metric. His reflections on refactoring also offered valuable insights. Often viewed as a technical necessity rather than a strategic activity, Tritsch framed refactoring as a continuous investment in the health and longevity of the codebase. This perspective positioned refactoring not as an occasional cleanup effort, but as a regular practice that maintains the system's adaptability and durability. It was particularly interesting to hear how refactoring can align with business goals, ensuring that the software remains responsive to evolving requirements.

Lastly, his perspective on generative AI was thought-provoking. Instead of viewing AI as a threat to software engineering jobs, Tritsch framed it as a tool for augmentation. The idea that AI-augmented engineers will outpace those who resist the technology was both surprising and motivating. It reframed AI as an opportunity to broaden the scope of a software engineer's role rather than diminishing its value. This vision of AI as a collaborative partner rather than a competitor offered a more optimistic and empowering view of the future of software development.

Furthermore, the discussion on the evolution of programming paradigms provided a broader historical context that deepened my understanding of the field. Tritsch's explanation of how functional programming builds on the principles of structured programming and object-oriented programming highlighted the continuous quest for better ways to manage complexity. This historical perspective underscored the importance of staying curious and adaptable, as the best practices of today are often built on the lessons of the past.

In conclusion, Roland Tritsch's podcast was enlightening. It confirmed some views, challenged others, and reinforced that software engineering is a human craft focused on building sustainable systems, not just writing code.

Chapter 6: SOLID Principles

During this lab session, I worked through a series of exercises aimed at identifying and resolving violations of the SOLID principles in object-oriented design. Each exercise presented a small codebase with one or more design flaws, and my task was to refactor the code while improving maintainability, extensibility, and overall software quality. In particular, I explored principles such as Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion, the Law of Demeter, and the avoidance of concrete superclasses.

6.1 Work Done

Exercise 1: Postage Stamps

The original `PostageStamp` class was tightly coupled to the `Square` class, which violated both the **Open/Closed Principle (OCP)** and the **Dependency Inversion Principle (DIP)**. It was not extensible to support shapes other than squares, such as circles. I introduced a `Shape` interface that both `Square` and `Circle` implement. This allowed `PostageStamp` to work with any shape without modification.

This design change improved compliance with OCP and DIP by ensuring the system is open to extension and depends on abstractions rather than concrete implementations. Additionally, this made the code more reusable and aligned with clean architecture principles.

Exercise 2: Module Attendee

The original `ModuleAttendee` class managed both the student's identity and module enrollment details, violating the **Single Responsibility Principle (SRP)**. To resolve this, I introduced a `ModuleAttempt` class to encapsulate module-specific details and grades.

This change also makes room for handling the case where a student takes multiple modules. Although the new design supports this, it raises concerns about how to interpret multiple grades for the same module. This could be addressed in the future by introducing a grading strategy class. The design remains consistent with SOLID principles.

Exercise 3: Blog Users

The original `IBlogUser` interface was too broad, violating the **Interface Segregation Principle (ISP)**. Not all users need to edit or block posts, yet they were forced to implement these methods. This resulted in runtime errors or placeholder exceptions.

I split the interface into smaller ones: `ICanView`, `ICanEdit`, and `ICanBlock`. Each role class (e.g., `Admin`, `Writer`, `Reader`) now implements only what it needs. This results in clearer responsibilities, better maintainability, and safer design.

Exercise 4: Shopping

The initial `ShopKeeper` class accessed and manipulated the internal `Wallet` object of a `Customer`, violating the **Law of Demeter**.

I refactored the code so that `Customer` handles the charging logic itself. `ShopKeeper` now only interacts with `Customer`'s public interface, avoiding direct dependency on implementation details. This makes the code easier to modify and reduces tight coupling between classes.

Exercise 5: Hospital System

The hospital management example presented multiple violations:

- **Single Responsibility Principle (SRP)**: `HospitalSystem` was responsible for admitting patients, saving data, sending notifications, and scheduling nurses.
- **Open/Closed Principle (OCP)**: `scheduleNurse` used conditionals rather than polymorphism to distinguish behavior.
- **Liskov Substitution Principle (LSP)**: `Doctor` overrode `clockIn` in a way that breaks expectations.
- **Interface Segregation Principle (ISP)**: `HospitalWorker` was too heavy.
- **Dependency Inversion Principle (DIP)**: `EmergencyRoom` directly depended on a concrete logger.
- **Law of Demeter**: `getPatientDetails` returned entire objects when only a portion of the data was needed.
- **No Concrete Superclasses**: `MedicalStaff` was a concrete base class, which should have been abstract to avoid misuse and improve semantic clarity.

To address these, I introduced service classes for specific concerns (e.g., `AdmissionService`, `NotificationService`), used interfaces and abstractions, and broke large interfaces into smaller, more meaningful ones. The resulting design is modular, testable, and easily extensible.

6.2 Reflections

Completing these tasks helped to demonstrate how crucial it is to follow SOLID principles while creating object-oriented systems. The most important lesson in each case came from identifying the patterns that caused the violations, even if they felt obvious after they were found.

In order to decouple components and lower future maintenance costs, I decided to refactor whenever possible using an interface-based design. I initially believed that a few conditionals were okay in the hospital system, but after some study, I realized that the polymorphic solution was much cleaner and more compatible with the OCP.

The Law of Demeter connection was especially intriguing since it demonstrated how unnecessary dependencies are introduced by relatively unnoticed methods like `getWallet().getValue()`. The advantage of approaching code as a living structure was also brought back to me by this exercise; modularity is key and it becomes essential as complexity rises.

Finally, this assignment resonated with ideas from other topics, such as software engineering and design patterns, and connected nicely with my past system architecture knowledge. I also liked how minor violations (such as a concrete base class or a bloated interface) can lead to more serious maintenance problems.

All things considered, this lab provided me with a toolkit for identifying and resolving design faults early in the development process as well as a practical understanding of the SOLID principles.

Chapter 7: Refactoring - Part 1

This lab focused on improving a small Java vehicle rental application that suffered from multiple code smells, including long methods, feature envy, switch statements, and data classes. The goal was to make the code easier to extend and maintain by applying refactorings such as Extract Method, Move Method, Rename, and Replace Type Code with Polymorphism.

7.1 Work Done

- Analyzed the original code for code smells and design flaws (God Method, Feature Envy, Primitive Obsession, etc.)
- Extracted the rental charge calculation logic into a new method
- Renamed unclear variables like `each` to `rental` for clarity
- Moved the new charge calculation method from `Customer` to `Rental` to eliminate feature envy
- Used automated refactorings (Extract Method, Move Method, Rename) and compared results with manual versions
- Discussed when and why test cases should or shouldn't be added after refactoring
- Reflected on method naming and removed redundant comments

7.2 Reflections

- Refactoring often means moving logic to where it belongs, not just cleaning code superficially
- When moving methods existing tests already gave good coverage
- IDE refactorings are fast and reliable, but it's good to understand what they do manually
- Naming is just as important as logic, bad names make good code unreadable
- Avoiding switch statements with polymorphism is cleaner and easier to extend later
- Separating logic from presentation (text/HTML output) would benefit from the Strategy pattern [Ref]
- Recognizing and fixing "Feature Envy" helped reinforce what OOP design is really about

Chapter 8: Refactoring - Part 2

This practical was a continuation of the previous lab focused on refactoring a vehicle rental system. However, the improvements suggested for this part had already been implemented during the earlier refactoring (Part 1). The original code had clear signs of long methods, feature envy, and others, and was previously cleaned up using several design principles and patterns.

8.1 Work Already Completed (from Part 1)

The tasks listed in this part of the assignment were addressed during the first refactoring session:

- **Extracted and moved methods:** The logic for calculating charges and frequent renter points was extracted into cohesive methods and moved to the appropriate classes (e.g.: `Rental` and pricing strategies), improving cohesion and reducing code duplication.
- **Replaced temp with query:** Local variables like `thisAmount` were removed in favour of direct method calls, clarifying the purpose of each computation.
- **Used polymorphism:** The switch statement on car rented pricing was replaced with polymorphic subclasses using different types of cars and therefore different methods for pricing, each handling its own logic.
- **Applied the Strategy Pattern:** Converted generating the statement for every rental into a Strategy Pattern [Ref], allowing customers to get their statements in different types of formats.
- **Added HTML Output:** An `htmlStatement()` method was implemented with minimal code duplication due to previous cleanups and separation of logic from presentation.

8.2 Reflections

- Much of this practical reinforced the benefits of doing refactorings early, many of the tasks suggested here became trivial or were already resolved.
- Strong encapsulation and the Strategy pattern helped make the code easy to extend (e.g.: adding HTML output).
- The use of polymorphism decoupled pricing logic from customer logic, making changes much simpler.
- This practical served as validation that the refactorings in (Part 1) were correctly targeted at major design issues like God methods, Feature Envy, switch statements and others.

Chapter 9: Observer Pattern

9.1 Work Done

- Identified tight coupling between `AlarmClock` and `Person` in `AlarmApplication`.
- Created a custom `Observer` interface and a `Subject` abstract class (without using Java's built-in library).
- Refactored `AlarmClock` to extend `Subject` and notify observers when the alarm is triggered.
- Made `Person` implement `Observer`, allowing it to wake up when notified.
- Extended the application to support multiple `Person` objects observing the same alarm.
- Implemented alarm time synchronization using the push model, then refactored to use the pull model.
- Introduced a new observer class, `Cron`, that performs unrelated tasks at a specific time.
- Implemented safe observer self-detachment after reacting to notifications using a copy of the observer list.

9.2 Reflections

- Initially used the push model to directly pass data to observers, simple, but less flexible.
- Switched to the pull model for better decoupling and cleaner design, especially with diverse observers.
- Pull model felt more scalable and aligned with real-world event-driven systems.
- Encountered a `ConcurrentModificationException` when modifying the observer list during iteration, resolved it by iterating over a copy of the list.
- Adding the `Cron` observer showcased how extensible the pattern becomes once properly set up.
- Considered extending the model further with personalized alarm times per `Person`.
- Reinforced understanding of OOP principles like SOLID (especially Open/Closed and Single Responsibility).
- Found the dynamic attach/detach concept compelling, with potential for use in GUIs or networked apps.

Chapter 10: Final Reflection on Refactoring and Design Patterns

10.1 Introduction

This report reflects on the software quality practicals focused on code refactoring and design patterns, specifically: Refactoring Part 1, Refactoring Part 2, and the Observer Pattern practical. The primary goal of each of these laboratories was to analyze poorly organized Java code, identify design errors and code smells, and use the right refactoring strategies and design patterns to improve flexibility, modularity, and maintainability.

The practicals emphasised the importance of software design principles, particularly the SOLID principles, the Law of Demeter, and no concrete superclasses, and their role in producing clean, adaptable code. Each task provided an opportunity to apply object-oriented thinking and evaluate the advantages and disadvantages between simplicity and flexibility.

In addition to implementing modifications into practice, the task included evaluating the design choices critically, identifying the effects of coupling and cohesion, as well as discovering links between code quality and long-term software maintainability. I explored a range of code restructuring techniques during these labs, from redesigning object roles and interface design to implementing polymorphism and strategy/observer patterns.

In the following sections, I will briefly summarize the technical work done and then explore deeper reflections across all three practicals, highlighting the key lessons learned, the reasoning behind certain decisions, and how they connect to broader concepts in software engineering.

10.2 Summary of Work Done

Each of the following presented tasks focused on identifying structural issues in object-oriented code and applying appropriate design improvements using refactoring techniques, design patterns, and key software principles (SOLID principles).

10.2.1 Refactoring Part 1

The first refactoring lab focused on a small Java vehicle rental application suffering from various code smells. Major issues included a God Method in the `Customer.statement()` function, violation of the Single Responsibility Principle (SRP), a switch statement that broke the Open/-Closed Principle (OCP), and Law of Demeter/Feature Envy where `Customer` knew too much about `Rental` and `Vehicle`.

The code was incrementally refactored using techniques such as Extract Method, Move Method, Rename, and Replace Type Code with Polymorphism. Responsibilities were distributed more appropriately across classes. A polymorphic architecture for `Vehicle` types (`Car`, `Motorbike`, `AllTerrainVehicle`) was introduced to eliminate switch statements. Business logic and presentation logic were separated using a Strategy pattern to support different output formats like text and HTML.

10.2.2 Refactoring Part 2

The second lab was designed to extend the refactoring done in Part 1. However, most of the improvements suggested had already been implemented earlier. Tasks such as moving charge logic into `Rental`, replacing temporary variables with queries, and applying the Strategy pattern were already complete. The work done in Part 1 aligned so well with the suggestions that no additional changes were needed.

This second part reinforced the value of performing targeted refactorings early. It showed how clean design choices made in Part 1 significantly reduced future maintenance and enhancement effort.

10.2.3 Observer Pattern

In the Observer Pattern lab, a small Java application involving an `AlarmClock` and a `Person` class was restructured using the Observer design pattern. The original code had tight coupling and direct control logic in the main application. These responsibilities were refactored by introducing a custom `Observer` interface and `Subject` abstract class.

`AlarmClock` was modified to become a subject that notifies observers when the alarm time is reached. `Person` was refactored to implement the `Observer` interface and react to notifications by waking up. Initially, the push model was implemented, where the alarm time was passed directly to observers. Later, this was replaced with a pull model, where observers query the `AlarmClock` for data.

The system was then extended to include multiple `Person` observers and a new `Cron` class, which performs separate tasks at a specific time. After that, observers were reimplemented to detach themselves safely during notification, avoiding a `ConcurrentModificationException` when modifying the observer list during iteration, by iterating over a copy of the list. These additions showcased the extensibility and flexibility provided by the Observer pattern when applied correctly.

10.3 Deep Reflections

The three practicals (Refactoring Part 1, Refactoring Part 2, and the Observer Pattern) offered a comprehensive journey through code quality, software architecture, and object-oriented design. Each exercise emphasised how even small improvements in structure and the distribution of responsibilities can greatly influence maintainability, flexibility, and developer understanding. Below, I reflect on the key lessons and insights gained, grouped around the core ideas of software design.

10.3.1 Applying Software Design Principles in Practice

Throughout all three labs, the SOLID principles and related software design guidelines were constantly there. For instance, in the refactoring labs, I encountered the classic "God Method" in the `Customer` class. Its `statement()` method handled everything from rental logic to pricing and formatting. This directly violated SRP, and the fix involved breaking that method into smaller, cohesive functions and relocating them to the appropriate classes. The method was essentially doing the job of three different objects (not good in practice).

The use of polymorphism addressed both the Open/Closed Principle (OCP) and the switch statement problem. By introducing a polymorphic vehicle architecture with `getCharge()` methods, new vehicle types could be added without touching existing logic. This was a clear win in terms of extensibility and reduced long-term risk.

In the Observer Pattern lab, I found the Law of Demeter and SRP coming back. Initially, `Person` and `AlarmClock` were tightly coupled, and the `AlarmApplication` class was doing everything. Moving to an Observer model allowed the logic to live where it made sense: `Person` responded to notifications, and `AlarmClock` only needed to maintain a list of observers. This improved modularity and respected SRP by distributing responsibilities clearly.

10.3.2 Refactoring for Software Quality

One major realization was that refactoring is not just about making code look cleaner, it's about preparing it for the future. Moving logic into the right places (e.g.: shifting charge calculations from `Customer` to `Rental` and `Vehicle`) created a more natural, intuitive structure that could be easily extended. The Strategy pattern, used to separate business logic from output formatting, proved especially useful and much more cleaner. Adding HTML output became trivial because the logic was already isolated from presentation.

Another aspect was dealing with Feature Envy. The original `Customer` class depended too much on details of `Rental` and `Vehicle`, which violated Law of Demeter. I gave each class the ability to "own" its logic by giving responsibility to the objects that contained the data. It became simpler to update or test individual components individually as a result of the decreased coupling.

I got a real understanding of how automation supports safe transformations by comparing manual refactoring with IDE tools. Although IDE tools are solid, especially when it comes to renaming or extracting methods, knowing what was underneath the code helped me make more appropriate design decisions when more important structural changes were needed.

10.3.3 The Magic of Design Patterns

The Observer Pattern lab brought a different flavor to the exercise. It involved reconsidering how things interacted with one another rather than simply reorganizing code. Implementing a custom Observer/Subject system gave me full control over how objects communicate.

Initially, I used the push model, where the `AlarmClock` directly passed the alarm time to observers. This worked, but felt awkward, especially once I added the `Cron` observer. Switching to the pull model, where the observers pull the needed data from the subject, made the design more scalable. Observers no longer needed to know what data would be pushed; they simply knew when to act.

This change highlighted the true power of the pattern: flexibility and decoupling. I could add new observer types without modifying AlarmClock at all, and each observer could behave differently. For example, Person used the alarm time to wake up, while Cron executed a task when a particular time was reached. The same notification mechanism supported both.

The Cron observer also reinforced the idea that well-designed patterns are open for extension. Without changing any existing code, I added new functionality that looked completely unrelated on the surface. This was a strong demonstration of the Open/Closed Principle in practice.

10.3.4 Dynamic Behaviour and Runtime Flexibility

One of the more advanced features I experimented with was observer self-detachment. Initially, I was trying to remove an observer from within the update() method, which caused a ConcurrentModificationException, because I was modifying the observer list while iterating over it. This forced me to think more carefully about how systems behave at runtime.

The solution (iterating over a copy of the observer list) may seem simple, but it opened the door to more dynamic behaviour. Now, observers can logically attach and detach themselves, allowing interactions limited by time or context. For example, once a Person wakes up, it no longer needs to observe the alarm clock. Similarly, Cron tasks could be implemented to run at recurring intervals or just once, depending on configuration.

This kind of flexibility is valuable in real-world applications such as GUIs, systems that have scheduled events, or IoT [Ora] devices, where components must respond to changes independently. It also highlighted the importance of designing with runtime behaviour in mind, not just compile-time structure.

10.3.5 Design Options and Reasoning

While the Observer pattern was effective, I did consider alternative implementations, especially when switching from the push to the pull model. Initially, passing all the data to observers seemed simpler, but it tightly coupled the subject to the format and semantics of that data. The pull model introduced a cleaner separation: the subject triggers an update, and observers decide what, if anything, to do with it.

Similarly, in the refactoring labs, I went over whether to keep some logic within Customer for convenience. But eventually, distributing responsibilities to Rental and polymorphic Vehicle classes made the system much easier to understand and modify. Refactoring requires balancing short-term clarity with long-term maintainability, and in most cases, the design which included principles proved to be more robust.

10.3.6 Connections to Other Modules and Personal Experience

These practicals reinforced concepts I encountered in other modules like Software Engineering, System Architecture and Design and Object Oriented Programming. The SOLID principles, design patterns, and refactoring techniques were not just theoretical, they became concrete tools for improving real code.

For example, in system architecture and design discussions, the importance of low coupling and high cohesion is often emphasised. I learned how to design flexible and testable systems by separating interaction logic from the application using the Observer pattern. Similarly, in previous work with GUI frameworks like Qt/PyQt, JavaFX and Android, I've seen how event listeners function as real-world observers. That connection helped me understand the relevance of this pattern beyond textbook examples.

I also recognized how Feature Envy and Primitive Obsession (like raw type codes) show up even in small projects. Addressing these early prevents long-term complexity and pain. Having built side projects and participated in team-based university projects, I now better appreciate how structural code issues can slow down collaboration and onboarding. These labs showed me how to use smart design to stay away from these traps.

10.3.7 Patterns and Principles Working Together

What stood out most across the labs was how design patterns and software principles reinforce each other. The Observer pattern respects SRP and OCP by moving logic into reusable components. The Strategy pattern used in the rental refactoring encapsulates behaviour and enables formatting flexibility, again respecting OCP and SRP.

Even the Law of Demeter, which can feel abstract, became meaningful when I removed chained access like `getWallet().getValue()` and instead allowed objects to handle their own responsibilities. Each small improvement added up to a cleaner, more understandable system.

The advantages were obvious since the concepts were applied consistently throughout the three labs. When I applied the Strategy pattern in the refactored rental system, adding HTML output was easy and did not require rewriting core logic. When I introduced Cron as a new observer, no changes were required in the alarm logic. These achievements demonstrated how beneficial it is to put effort into architecture.

10.3.8 Final Thoughts on Design Improvement

These practicals changed my perspective on code quality. Initially, I approached them with a focus on getting things to "work" or cleaning up messy methods. In the end, however, I was considering the flexibility, clarity, and long-term sustainability of the project from a broader perspective.

Good design is invisible when done right, but painfully obvious when missing. These exercises helped me practice making that design explicit, aligning my code with principles that promote simplicity and control, simultaneously. They also showed me that patterns and refactoring are not distinct techniques but rather matching elements of a larger strategy for developing software that persists.

10.4 Conclusion

The refactoring and design pattern practicals provided a hands-on exploration of software quality, maintainability, and extensibility. Through the process of analyzing code smells, applying targeted refactorings, and introducing design patterns, I developed a deeper appreciation for the structure and purpose behind well-designed software systems. Concepts such as the Single Responsibility Principle, the Open/Closed Principle, and the Law of Demeter became not just theoretical ideals, but practical tools that guided my decision-making.

Each lab reinforced the importance of clean, modular design, not as a form of perfectionism, but as a way to make code easier to extend, test, and collaborate on. The Observer Pattern exercise, in particular, highlighted how design choices can open up systems to new features without needing aggressive changes, reflecting the Open/Closed Principle in its purest form.

In the end, these exercises helped me change my perspective from just addressing problems to doing so in a way that respects the long-term health of the application. They helped me understand that code is an ecosystem that should be understandable, adaptable, and resistant to change.

Bibliography

[All23] Allen Higgins and Déaglán Connolly Bree, joined by Karl McCabe. Software Quality in Large Scale-Software Development. https://corporatefinanceinstitute.com/resources/fpa/types-of-budgets-budgeting-methods/, February 2023.

[All25] Allen Higgins and University College Dublin (UCD) students, joined by Roland Tritsch. Programming as Craft. https://shows.acast.com/design-talk/episodes/programming-as-craft, February 2025.

[Hen] Henry Coles. Pitest. https://pitest.org/. Source code: https://github.com/hcoles/pitest.

[Ora] Oracle. What is IoT? https://www.oracle.com/ie/internet-of-things/.

[Ref] Refactoring Guru. Design pattern: Strategy. https://refactoring.guru/design-patterns/strategy. Java example: https://refactoring.guru/design-patterns/strategy/java/example.

[Wika] Wikipedia contributors. Conway's game of life — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=1275156291".

[Wikb] Wikipedia contributors. Ensemble learning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ensemble_learning&oldid=1276597003.

[Wikc] Wikipedia contributors. Moore neighborhood — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Moore_neighborhood&oldid=1262279738.