Reflection on Sea++ Development for CSE5008 Programming Assignment 2 May 4, 2025

1 Introduction

This reflection discusses the development of Sea++, a C++ application for managing seafood catches, as part of CSE5008 Programming Assignment 2. It covers the extensions made in Part III, the experience of designing and implementing the initial system in Parts I and II, the challenges and rewards encountered, and the key lessons learned.

2 Changes Made in Part III

In Part III, I extended the Sea++ application with two significant additions to enhance its functionality and maintainability.

2.1 Bag and BagChecker

The first extension introduced the `Bag` and `BagChecker` classes. The `Bag` class allows anglers to store multiple `SeaCreature` objects, simulating a real-world fishing bag. It uses a `std::vector<SeaCreature*>` to manage creatures and handles their lifecycle through its destructor. The `BagChecker` class validates the bag against regulations, such as bag limits (e.g., maximum 5 Snapper per angler, as per NSW DPI guidelines). This extension required updates to the `App` class to manage the `Bag` and to the `SeaPlusPlusEngine` class to delegate bag validation via a new `processBag` method.

This addition made Sea++ more realistic by enabling anglers to track multiple catches and ensure compliance with bag limits, a critical aspect of sustainable fishing. Integrating `BagChecker` into the Mediator pattern (via `SeaPlusPlusEngine`) maintained the system's modularity.

2.2 SeaPlusPlusInfoSupplier and JsonInfoSupplier

The second extension introduced the `SeaPlusPlusInfoSupplier` abstract class and its concrete subclass `JsonInfoSupplier`. These classes enable loading regulation data (e.g., size limits) from a JSON file (`regulations.json`) using the `nlohmann/json` library. The `SeaChecker` class was modified to hold a `SeaPlusPlusInfoSup` pointer, allowing `VertebrateChecker` and `InvertebrateChecker` to access dynamic regulations instead of hardcoded limits.

This extension improved maintainability by decoupling regulation data from the code, allowing updates without recompilation. It also made Sea++ more extensible, as new data sources (e.g., CSV or database) could be supported by adding new `SeaPlusPlusInfoSupplier` subclasses.

3 Initial Design and Implementation Experience

3.1 Part I: UML Design

Creating the UML diagram in Part I was straightforward due to the clear application of design patterns. The Façade pattern was implemented via the App class, simplifying user interaction. The Mediator pattern was realized through the SeaPlusPlusEngine class, centralizing coordination between checkers. The Factory Method pattern was used in the SeaCreatureFactory hierarchy to enable flexible creation of SeaCreature objects. Designing the class relationships (e.g., inheritance for SeaCreature and SeaChecker, association between App and SeaPlusPlusEngine) was intuitive, as the patterns provided a clear structure. Using PlantUML to generate the diagram ensured clarity and professionalism.

3.2 Part II: C++ Implementation

Implementing the design in C++ for Part II was more time-consuming but rewarding. Separating each class into header (.h) and source (.cpp) files enforced modularity and maintainability. Implementing the Factory Method pattern required careful design of the SeaCreatureFactory hierarchy, ensuring polymorphic creation of VertebrateCreature and InvertebrateCreature. The Mediator pattern in SeaPlusPlusEngine was effective in coordinating SeaChecker subclasses, using dynamic$_{c}asttoselecttheappropriatechecker.$

Memory management with raw pointers was a significant consideration, requiring explicit delete calls in destructors (e.g., App:: App). While smart pointers would have been safer, raw pointers were used for simplicity, as permitted by the assignment. Hardcoding regulations in VertebrateChecker and InvertebrateChecker was straightforward but less flexible, motivating the SeaPlusPlusInfoSupplier extension in Part III.

4 Difficult Aspects

The most challenging aspects of the project included:

- **Memory Management**: Ensuring proper deletion of SeaCreature objects and checker pointers was tedious, especially in App and SeaPlusPlusEn A single oversight could cause memory leaks, highlighting the need for careful destructor implementation.
- **JSON Integration in Part III**: Setting up the nlohmann/json library for JsonInfoSupplier required configuring the build system (e.g., CMake), which was complex for a C++ beginner. Parsing the JSON file correctly to extract size limits also required learning the library's API.
- **Extending the Design**: Integrating BagChecker into SeaPlusPlusEngine required modifying the Mediator pattern without breaking existing functionality, which demanded careful planning to maintain encapsulat

5 Rewarding Aspects

The most rewarding aspects included:

- **Seeing Design Patterns in Action**: Applying the Façade, Mediator, and Factory Method patterns made the code feel professional and scalable, mirroring real-world software engineering practices. The UML diagram provided a clear roadmap, making implementation enjoyable.
- **Functional Application**: Running Sea++ and seeing it correctly validate creatures (e.g., “You can keep the Snapper!” for a 35 cm Snapper with no eggs) was satisfying, especially with the Part III extensions enabling bag validation.
- **Extensibility**: Adding Bag and SeaPlusPlusInfoSupplier in Part III was seamless due to the modular design, reinforcing the value of design patterns. Updating regulations via regulations.json felt like a practical solution for real-world use.

6 Lessons Learned

This project provided valuable insights:

- **Design Patterns Enhance Scalability**: The Façade, Mediator, and Factory Method patterns simplified extensions and maintenance, demonstrating their power in software design.
- **Planning is Critical**: The UML diagram in Part I reduced implementatio errors and clarified relationships, emphasizing the importance of upfront design.
- **C++ Requires Vigilance**: Manual memory management underscored the need for careful resource handling. Exploring smart pointers in future projects could improve safety.
- **External Data Improves Flexibility**: The SeaPlusPlusInfoSupplier extension highlighted the benefits of decoupling data from code, a principle applicable to many applications.

7 Conclusion

Developing Sea++ was a challenging yet enriching experience. The design patterns provided a robust framework, making the initial design and extensions manageable. While memory management and JSON integration posed difficulties, the functional application and extensible design were highly rewarding. The lessons learned about design patterns, planning, and C++ programming will inform future software development projects, ensuring more efficient and maintainable solutions.