**Reflection on Sea++ Development for CSE3PSD Programming Assignment 2**

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.

1. **Changes Made in Part III**

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

* 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.

1. **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\_cast to select the appropriate checker. 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.

1. **Difficult Aspects**

The most challenging aspects of the project included:

* Applying the Façade Pattern: Implementing the Façade pattern in the App class required balancing simplicity for the user with the need to expose enough functionality to integrate with the SeaPlusPlusEngine. Ensuring the Façade did not become a bottleneck or overly restrictive demanded careful design, especially when adding the Bag management in Part III.
* Extending the Mediator Pattern: Integrating the BagChecker into the SeaPlusPlusEngine as part of the Mediator pattern was complex. Modifying the existing mediation logic to accommodate bag validation without disrupting the coordination of SeaChecker subclasses required precise adjustments to maintain loose coupling and avoid unintended side effects. The use of dynamic\_cast added further complexity, as incorrect type handling could lead to runtime errors.
* Adapting the Factory Method Pattern: Extending the Factory Method pattern in the SeaCreatureFactory hierarchy to support the static createCreature method used in App was challenging. Ensuring the factory correctly handled all creature types (e.g., VertebrateCreature, InvertebrateCreature) and integrated with the Bag extension required thorough testing and refinement to avoid creating invalid objects.

1. **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 Façade pattern’s simplicity in App streamlined user interaction, while the Mediator pattern’s coordination in SeaPlusPlusEngine allowed seamless integration of new components like BagChecker. The Factory Method pattern’s flexibility in SeaCreatureFactory reduced code duplication and enhanced extensibility. 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.

1. **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. The Façade pattern reduced complexity for users, the Mediator pattern facilitated component interaction, and the Factory Method pattern supported object creation flexibility, all of which were critical for the Part III extensions.
* Planning is Critical: The UML diagram in Part I reduced implementation 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.

1. **Conclusion**

Developing Sea++ was a challenging yet enriching experience. The design patterns provided a robust framework, making the initial design and extensions manageable. The Façade pattern simplified the interface, the Mediator pattern enabled coordinated extensions, and the Factory Method pattern supported flexible object creation, all of which were pivotal to the project’s success. While memory management and design pattern adjustments 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.