**GRASP and Design Patterns:**

This section is dedicated to any programming problems are inefficacies that were overcome using the object oriented GRASP principles or design patterns learned in class. Throughout the development of the Tower Defense game our team was able to identify and resolve general coding problems using the various GRASP principles. Some of the most prevalent principles used include Indirection, Information Expert, High Cohesion, and Polymorphism. These principles allowed the code of the Tower Defense game to be considerably easier to understand and edit. To complement and implement the numerous GRASP axioms, an assortment of design patterns were used. Among them, some of the more prominent patterns are Observer, Strategy, and Singleton. Without these fundamental principles and patterns, the process of interpreting or editing existing code would be an arduous and unpleasant experience.

**Principles**

**1) Principle Name:** Indirection Principle

**Problem:** In the tower defense game, who should be responsible for adding removing or notifying observers?

**Solution:** Instead of directly implementing the add remove and notify observer methods in critter, a critter extends a separate subject class that includes the list of IObservers and all associated methods. This lowers the coupling between observers and critters and allows the potential for other classes to be observed by having them extend the Subject class.

(Diagram HIGHLY recommended)

**2) Principle Name:** Information Expert

**Notes:** This principle is one that is used on multiple occasions throughout the coding process ho

**Problem:** In the Tower Defense Game which class should be responsible for knowing the activity information of a tower or critter instance?

**Solution:** Towers and critters are the classes that are responsible for knowing their respective list of activities. The tower and critter instances are their own respective information experts. This results in low coupling creating potential for modularity.

(Optional Diagram)

**3) Principle Name:** High Cohesion

**Problem:**

**Solution:**

(Optional Diagram)

**4) Principle Name:** Polymorphism

**Problem:** In the Tower Defense game how are critters of identical attributes yet varying attribute values handled?

**Solution:** The Tower Defense game will require the creation/use of multiple critter “types” and as such, it would impractical to reference critter “types” individually when the demand to reference every critter is required. In this situation the Polymorphism principle is extremely applicable and recommended. With the implementation of a critter “superclass”, every critter type does not need to be referenced individually and furthermore the creation of additional critter types becomes significantly simplified has critter all critter types inherit attributes from the critter superclass. A beneficial side effect of applying this principle is the reduced coupling it produces within the critter hierarchy as the individual critter types are very rarely if ever referenced instead of the critter superclass.

(Optional Diagram)

**5) Principle Name:** Creator

**Problem:** In the tower defense game, who should be responsible for creating instances of the tower and critter class?

**Solution:** A creator class, in the form of gameController, is responsible for the instantiation of all towers and critters. The gameController contains towers and critters and as such has the authority to instantiate them demonstrating an upwards dependency.

(Optional Diagram)

**6) Principle Name:** Controller

**Problem:** In the tower defense game, who should be responsible for mouse and keyboard input events?

**Solution:** There is a dedicated class for handling all mouse related input events named MouseAndKeyboardHandler. This class is referenced and controlled by other classes like gameController, in the business logic layer, and consequently by GameApplicationFrame, in the presentation layer. Allowing all input related events to be handled by a controller class has allowed the implementation of input related events in the presentation layer without any downward dependencies.

(Optional Diagram)

**Patterns**

**1) Pattern name:** Singleton

**Problem:** In the tower defense game who is responsible for creating one and only one instance of the clock class while communicating and interacting with various other classes?

**Solution:** The clock class is the class responsible for instantiating itself and ensuring that there is only ever one instance of the class at all times.

(Optional Diagram)

**2) Pattern name:** Observer

**Problem:** In the tower defense game who is responsible for allowing the gameController to be updated every time a change occurs in the critter class?

**Solution:** The implementation of the IObserver interface and the subject class allow the gameController to be informed every time a critter moves or becomes damaged. The critter class extends the subject class and the gameController implements IObserever thus acting as an observer for critters.

(Optional Diagram)

**3) Pattern name:** Strategy

**Problem:** In the tower defense game who is responsible for the various strategies that can be applied to the tower class resulting in a change in targeting patterns?

**Solution:** The interface IStrategy is responsible for all strategies used by the tower. The strategy classes themselves, each with different targeting patterns, implement the strategy interface.

(Optional Diagram)