Discussion on tactics selected to support architectural use cases.

*“Programming to an interface, not an implementation”:* Creational patterns are one way to ensure that your program is written in terms of interfaces not implementation. The Factory Method for example extracts the process of creating our ‘Enemy’ class, thus giving us different ways to associate an implementation with its interface at instantiation.

“Enemy[] monsters = enemyFactory.createEnemy(“”Monster”,…”)”

Also the use of design patterns such as the ‘Observer’ promotes this tactic. Our ‘Observer’ class acts as an interface from which all Observable classes (‘Enemy’, ’Ui’ and all their subclasses) must comply with, by simply implementing it’s ‘update()’ method. This allows modifications of interest to be supported in the ‘Observer’ design pattern. This is possible due to the loose coupling between Subject and Observer. All the Subject knows is that it has a list of Observers, each conforming to the simple interface defined in our ‘Observer’ class.

*“Favour aggregation/composition over inheritance”:*

*“Find what varies and encapsulate it”:* Our use of the ‘State’ design pattern is an example of how we encapsulate what varies in our code. Depending on the difficulty of the game, our enemies will behave appropriately in terms of their movement strategies. We therefore have an abstract ‘State’ class that encapsulates the enemy’s stateand have three ‘ConcreteState’ classes that provide implementation details for these states at easy, medium and hard difficulty. The behaviour of our enemies must change at runtime and using the ‘State’ design pattern allows us to do this because the changing states are encapsulated through the use of a ‘State’ interface.