C# Design Patterns

The following notes are my own phrasing of various C# design patterns based on the understanding I have gained through independent study. They are intended for the purpose of revision and proof of my understanding.

- JACPro

# SOLID Design Principles

**Single Responsibility Principle** – Any class should only be responsible for a single part of the program’s functionality and thus have only one reason to change. New functionality should be added in new classes. This aids readability and in the event of a problem, the source of the issue will be easy to locate based on the nature of the problem (e.g. if text is not being saved to a .txt file correctly, you know that the issue will be in the class that handles persistent data).

*Separation of Concerns* – Different classes should handle different, independent tasks/problems.

**Open-Closed Principle** – Classes should be open for extension but closed for modification. It should be possible to extend a class when expansion is required without going back and modifying the original code (e.g. rather than making an enemy class and filling it with different functionality for different kinds of enemies, abstract common information to a class or interface that is then inherited or implemented by the different kinds of enemies in their own classes).

**Liskov Substitution Principle** – You should be able to substitute a base type for a subtype. When upcasting to a base type from a subtype, operations should still function as expected. You can easily amend issues like this by making properties and functions virtual in the base type and using the override keyword in the subtype so that when an object of the sub type is stored in a variable of the base type, properties and functions from the sub type will still be used.

**Interface Segregation Principle** – Interfaces should be segregated so that developers implementing your interface is only implementing the functions they need and not unnecessary functions. Don’t put too much into any one interface.

**Dependency Inversion Principle** – High-level parts of the system should not depend on low-level parts of the system directly; they should instead depend on some kind of abstraction. Low-level parts should provide an interface that exposes the necessary properties or functions to higher level functions; this allows for low-level parts to be reworked as necessary without affecting the high-level classes that implement the interface.