Design Patterns in ConsoleDungeonCrawler

**Proxy Pattern:**

Wird für verschiedene Systeme des Games verwendet.

Das Game implementiert die folgenden Interfaces: ITraitBehaviour.cs, IBaseController.cs, IBaseView.cs, IImpactBehaviour.cs, ILevelBuilder.cs.

Die Interfaces haben dabei jeweils die Aufgabe, ein System zu ermöglichen in welchem durch Klassen interiert werden kann bzw. Klassen untereinander ausgetauscht werden können, welche die selbe Verwendung innerhalb der Programstruktur, jedoch unterschiedliche Einzelaufgaben/Funktionalitäten besitzen.

Als einfaches Beispiel ist hierbei IBaseController zu nennen. Die beiden Klassen ConsolePlayerController.cs und ConsoleInventoryController.cs implementieren IBaseController und sind dadurch innerhalb von MasterControlProgram.cs austauschbar. Dieses führt lediglich die Execute() Funktion des Controllers aus, welche beide genannten Controller besitzen. Execute() selbst sorgt jedoch dafür, dass beim ConsolePlayerController die Inputs den Charakter innerhalb des Spiels steuert, beim Wechsel in das Inventar jedoch mit dem ConsoleInventoryController ausgewechselt wird, welcher nun für die gleichen Inputs andere Funktionalität, nämlich das Navigieren des Inventars erlaubt.

Durch das problemlose Austauschen von Klassen, welche IBaseController implentieren, kann zur Laufzeit jederzeit bestimmt warden, welche Inputs dem Spieler zur Verfügung stehen, ohne einzigartige Inputs oder komplexe Kontrollerstrukturen zum Management zu benötigen.

**Singleton Pattern:**

Verwendung als Pseudo Prefab-System in Form der ItemLibrary.cs und EnemyLibrary.cs.

Ddadurch, dass das Spiel in seiner Natur zufällig ist und nicht immer den selben Content bietet, haben wir uns entschieden ein System zu verwenden, welches uns erlaubt die Objekte die das Spiel zur Verfügung hat im Voraus zu erstellen und dann das Spiel aus diesen Objekten zufällig auswählen zu lassen um immer andere Kombinationen zu ermöglichen. Dadurch ist es Möglich, von 30 verschiedenen Waffen jedesmal eine andere Kombination auf 15 dieser Waffen ins Level zu werfen. Oder aus im Voraus gebauten Feinden zufällige auszuwählen, welche der Spieler im jetzigen Level zu besiegen hat.

Verwendet wurden dafür die beiden oben genannten Klassen, welche das Singleton-Pattern verwenden. Dies erlaubt uns globalen Zugriff auf die Listen von Items und Feinden und gleichzeitig eine Klasse in welcher wir diese erstellen und ordnen können, wie etwa das Aufspalten der Einheiten in verschiedene Gruppen um sicherzustellen das keine „Art“ oder „Gruppe“ von Feind im Level später dominiert, sondern eine ausgewogene aber zufällige Auswahl an Feinden existiert.

**Facade Pattern:**

Verwendung innerhalb der ItemLibrary.cs zur Erstellung von Objekten des Usable.cs-Typs die als „AmmoMods“ gedacht sind. „AmmoMod“ ist hierbei ein Tag welchen Usables bekommen, wenn sie Traits.cs zum Player hinzufügen, welche nach dem Nachladen der Waffe wieder entfernt werden sollen.

Da das Erstellen solcher Usables die Traits und ITraitBehaviours.cs sehr stark implementiert, ist beim Erstellen eines neuen Items welches im Spiel die Funktion eines AmmoMod-Items haben soll über den Usable.cs Konstruktor sehr komplex, da bestimme Variablen konstant bestimmte Werte besitzen müssen, damit die gewünschte AmmoMod-Funktion auftritt.

Um die Implementierung neuer AmmoMods zu erleichtern wurde deshalb eine Facade-Klasse geschrieben, welche Usables erstellt die immer die AmmoMod Funktionalität besitzen und optional die Werte anpassbar macht, welche verändert werden dürfen ohne die Funktionalität zu beeinflussen. Dies beinhaltet das Hinzufügen von eigenen ITraitBehaviour sollten die built-in Behaviour nicht ausreichen.