# Stories

* ~~As a fighter, I start the game with full health~~
* ~~As a fighter, I lose health if I get damaged~~
* ~~As a fighter, I receive health if I heal myself~~
* ~~As a fighter, I start the game alive and die when my health drops to zero~~
* ~~As a fighter, I am on my last breath when my health drops under a certain threshold~~
* ~~As a fighter, I can deal normal damage and heavy damage after charging (which is reset after one attack)~~
* ~~As a fighter, I can charge up to three times and get a bigger boost~~
* ~~As a fighter, I can receive lasting damage boost on my attack values and remove them again~~
* ~~As a fighter, I have a small chance to dodge an incoming attack~~
* ~~As a player, I start the game with no points~~
* ~~As a player, I can gain and lose points~~
* ~~As a player, I can equip a weapon that gives me a constant bonus on attack damage~~
* ~~As a player, I can collect items and store them in an inventory~~
* ~~As a player, I can use items from my inventory for healing e.g.~~
* As a player, if I collect a new weapon, I want to be asked whether I want to equip it directly
* As a player, I have an inventory limit for items
* As an enemy, I drop random loot (weapons/items) when I die
* **As an enemy, I give the player points when he kills me**
* As an enemy, I will choose my actions randomly (charge or attack)
* As a support enemy, I can heal will try to heal another enemy or myself once their health drops below 50%
* The game controller controls the flow of battle, so that player and enemies each get their turn??? ATB???
* The game controller starts the battle as soon as the player approaches an enemy
* The game controller ends the battle when all enemies are defeated or when the player dies
* The game controller groups enemies into small squads that the player must fight
* Items/Weapons can be collected by approaching them

# Notes

* Einstieg sehr einfach, Test Runner führt einen durch nötigen Aufbau
* Test vs. UnityTest
* MonoBehavior nicht zu testen? Logik in Klassen auslagern?
* Lösung für Namespace Problem? Alles in einem Ordner ist keine Lösung
* Assert.Equals() nur für Objekte? 🡺 AreEqual für Werte
* PlayerStats Test sehr einfach zu erstellen.
* Refactor Idee: maxHealth in PlayerStats variabel machen 🡪 so ist es einstellbar durch Spieler oder im Spiel durch Power-Ups, Refactor der Tests nötig
* Refactor Idee: ReceiveDamage() und GetHealedBy() zusammenführen 🡪 Refactor! Doch verworfen, damit negatives Heilen usw getestet werden kann
* [Test] vergisst man gerne
* Fehler sofort gefunden: Hatte recoveren von Last Breath in Damage Funktion statt Heal Funktion geschrieben
* Tests so unabhängig wie möglich machen: Damage Boost after charge Tests überprüfen, ob der Damage Boost richtig angewendet wurde, indem sie einen Setter anwenden. Ob es nur einen Boost gleichzeitig oder eine Liste gibt, ist den Tests egal und kann unabhängig von ihnen angepasst werden. Sie überprüfen nur die resultierenden Werte und melden, wenn Dinge nicht passen
* Enemy Klasse angefangen: Auffällig, dass viel Code dupliziert wird. Refactor: Eine allgemeine Fighter Klasse, davon erben Player und Enemy
* PlayerInventory für Equipment und Items
* Refactor der Assert Funktionen, Less, NotEqual, Zero, Positive etc sind interessant + custom Fehlermeldungen
* Problem mit Unity 2018: Test Scripts im Test Folder finden normale Scripte in anderen Ordnern nicht. Von Hand Assembly erstellen und in der Test Assembly referenzieren (<https://forum.unity.com/threads/unit-tests-cannot-find-my-namespaces-classes.515742/>)
* Humble Object Pattern für MonoBehaviors? MonoBehavior enthält Referenzen auf ausgelagerte C# Scripts, die die Logik enthalten, ausführen und getestet werden
* [Serializable] Attribut über den C# Klassen must have, damit sie im Inspector angezeigt werden
* Active Time Battle? Dann kann Static Time Klasse von Unity umgangen werden
* Weapons/Items als ScriptableObjects, dann sind die auch verwendet
* Fighter nicht tot, sondern lastBreath wenn Health = 0 nicht gefangen durch Test, weil ich Leben nicht auf 0 sondern negativ gemacht habe. Tests sind nur so schlau wie ihr Autor
* Humble Object Pattern Grund: MonoBehaviors lassen sich nicht per Script instanziieren und verwenden wie normale Klassen
* Problem Inventar und Stats: Stats liefern den Damage Wert des Spielers. Inventar liefert Damage Wert der ausgerüsteten Waffe. Um das zu kombinieren, müssten entweder beide Dinge zu einer Klasse werden (widerspricht den Prinzipien und würde dem Ganzen seine Übersicht nehmen) oder sich gegenseitig referenzieren (dann wären sie sehr stark gekoppelt, was von der Logik komisch wäre und das Ganze verkompliziert). Wir bleiben daher dabei, dass beide getrennt voneinander im MonoBehavior existieren und verwendet werden. Das MonoBehavior muss nun also in einer eigenen Funktion die Damage Werte aus beiden Teilen nehmen und selber mit der korrekten Rechnung kombinieren. Das hilft unserem Problem nicht, dass wir MonoBehaviors nicht mocken können, um sie zu testen. Daher Lösung: Wir lassen unseren Player ein Interface implementieren, welches die Funktion mit der Rechnung vorgibt.
* Pure Functions? Nein, impure functions, zu testen mit Test Doubles, eben Mock etc
* NSubstitute runterladen: war mal in den Testing Tools, aber entfernt bei Merge mit Unity?
* <https://www.nuget.org/packages/NSubstitute/2.0.1-rc> --> Package Manager
* in Package Manager Console ausführen <https://docs.microsoft.com/de-de/nuget/tools/package-manager-console>
* dll in Unity einfügen, sonst compiled zwar vs aber unity nicht <https://answers.unity.com/questions/1461487/nsubstitute-doesnt-register-in-unity.html>
* Selbes mit System.Threading.Tasks.Extensions.dll (sonst ging Substitute.For<> nicht, weil ValueTask<> nicht bekannt war) und CastleCore.dll (weil ohne diese dll NSubstitute Exceptions im Constructor warf, wenn man den Test laufen ließ)
* LÖSUNG: stats gibt den CurrentAttackValue aus, ohne die Waffe zu beachten. Das inventory liefert einen Verweis aus der Waffe, von welcher man den Bonus auslesen kann. Die Kombination beider Werte müsste eig im MonoBehavior passieren (die beiden Klassen sollen in keinem Zusammenhang stehen), aber dort könnte man es nicht testen. Daher implementiert das MonoBehavior nun ein Interface, welches die Funktion GetAllAttackBonus() vorgibt. Das MonoBehavior implementiert diese nun so, dass sie die Waffe überprüft und deren AttackBonus ausliest (ggf. noch weitere Boni von anderen Dingen, später). Später wird dann einfach CurrentAttackValue() + Bonus gerechnet. Da wir mit NSubtitute das Interface mocken können, sind wir in der Lage, den Bonus der Waffe zu testen: wir können der GetAllAttackBonus() Funktion des mock Objekts befehlen, für unseren Test den Bonus der Waffe zurückzugeben – wir können hier also eingreifen und unser benötigtes Verhalten erzeugen, ohne das MonoBehavior zu benötigen. Daraufhin können im Test den Schadenswert mit Waffe vergleichen und schauen, ob er wirklich höher ist als ohne Waffe. Dies läuft über die stats Klasse, welche einen Verweis auf das Interface hat (im Spiel selber wird dies ein Verweis auf das MonoBehavior sein) und darüber GetAllAttackBonus() direkt in ihrer GetCurrentAttackValue() Funktion verwenden kann
* Player Inventar Item Functionality: Sehr vergleichbar mit Conditions etc für mein Quest System
* Random.Range() funktioniert in Tests, aber nicht in der Szene? Random.value funktioniert aber
* Fighter können jetzt dodgen + un-dodgeable Attacks: so testet man random Sachen: Man will verhindern, dass Tests ein unvorhersehbares Ergebnis haben, sie sollen IMMER zuverlässig aussagen, ob ein Feature noch so funktioniert wie gedacht. Daher muss man den Random Faktor irgendwie umgehen, ihn als Input bestimmbar machen. Hier haben die Stats jeweils eine Wahrscheinlichkeit, mit welcher sie dodgen können. Diese kann für Tests selber gesetzt werden, um z.B. garantiert zu dodgen. Außerdem gibt es die Möglichkeit, nicht zu dodgende Angriffe durchzuführen (maybe nach Chargen?)
* Test zum Überprüfen, dass Subklassen eine eigene Implementierung für eine Funktion der Parent Klasse haben: Parent Klasse wirft Warning, wenn ihre Funktion aufgerufen wird. Test der Parent Klasse überprüft, ob diese Warnung kommt. Subklassen rufen Funktion jeweils auf sich selber auf und überprüfen mit LogAssert.NoUnexpectedReceived(); dass kein nicht erwarteter Log geschrieben wird, in diesem Fall der aus der Parent Klasse