Dokumentation zum Projekt „Terminal Arena“

Einleitung

Dieses Programm soll die Prinzipien der Objektorientierten Programmierung (OOP) verdeutlichen und anwenden. Es erhebt keinen Anspruch auf Vollständigkeit, auch wurde zugunsten der Übersichtlichkeit und Kompaktheit auf Unit Tests oder ähnliches verzichtet. Die UML-Diagramme und das Programm selbst wurden mit Visual Studio Community 2019 erstellt.

Abstraktion

Der Vorgang der Abstraktion besagt, dass ein ursprünglich komplexes Konzept auf ein weniger komplexes reduziert wird. Durch das Finden von Gemeinsamkeiten kann hiermit eine Basis geschaffen werden, auf der zusätzliche Funktionalität aufsetzen kann. Am Beispiel des Kämpfers wird dies deutlich:

Als Ausgangslage seien hier ein Gegner und ein Spieler genannt. Beide haben grundsätzlich ähnliche Eigenschaften und Fähigkeiten (Methoden), die aber nicht alle relevant für den Anwendungsfall eines Arenakampfes sind. So sind beispielsweise Alter, Haarfarbe, Größe und so weiter im vorliegenden Fall nicht ausschlaggebend für das Ergebnis. Also *abstrahieren* wir die Kämpfer so weit, dass nur noch relevante Daten übrigbleiben (Abb. 1: Das Interface "Entity"). Die interessantesten Eigenschaften und Methoden bleiben erhalten, der Rest wird weggelassen.

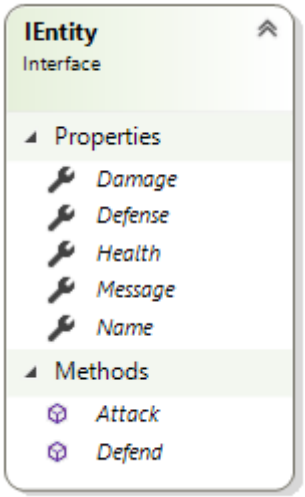


Abb. 1: Das Interface "Entity"

Ein anderes Beispiel hierfür ist ein Gegenstand, der sich in einem Inventar befindet. Hier gibt es Waffen, Rüstung, Tränke und Ähnliches, jedoch wäre es sehr aufwendig, alle diese Gegenstände einzeln zu implementieren. Nach der Abstraktion sind beispielsweise nur noch Wert, Gewicht und Name des Gegenstandes übrig, alle weiteren Informationen werden ignoriert (vgl. Abb. 2: Das Interface "Item").

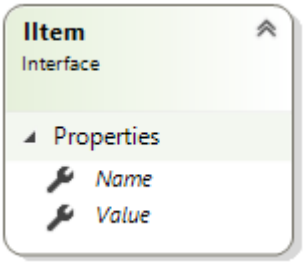


Abb. 2: Das Interface "Item"

Nach und nach werden nun Eigenschaften hinzugefügt, die weiterhin von Interesse sind (siehe Vererbung und Polymorhpie).

Datenkapselung

Die Datenkapselung soll sicherstellen, dass ausschließlich Eigenschaften und Methoden, die für andere Klassen oder externe Methoden von Interesse sind, nutzbar sind. Außerdem bietet sie durch Zugriffsfunktionen („getter“ und „setter“) die Möglichkeit, Daten zu validieren oder zu verändern.

Als Beispiel:

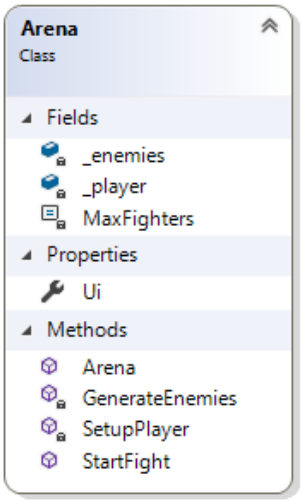


Abb. 3: Die Arena-Klasse

In der Arena-Klasse, die eine Kampfarena darstellt, sind nur der Konstruktor und die Methode `StartFight()` von außen sichtbar. Die Eigenschaften und restlichen Methoden werden nur innerhalb des Objektes benötigt, sind also auch nur dort sichtbar bzw. können nur dort verändert werden.

Die Felder `_enemies` und `_player` werden erst in der Arena befüllt, während die Eigenschaft `Ui` aus der aufrufenden Methode übergeben wird.

Dieses Verhalten kann in einer Klasse mit den Schlüsselwörtern `private` und `public` erreicht werden. Handelt es sich um eine *abstrakte* Klasse, so nutzt man statt private das Schlüsselwort `protected`.

Am Beispiel des Kämpfers wird dieses Prinzip auch noch einmal deutlich (Abb. 4: Die Kämpfer-Klasse).

Außer dem Kämpfer selbst kann niemand denselben dazu anweisen, einen Trank aus dem Inventar zu nehmen. Lediglich die Methode `ActOn(string what)` kann den Kämpfer dazu veranlassen, eine Aktion auszuführen, von welcher eine ist, einen Trank zu benutzen. Ebenfalls hat ausschließlich die jeweilige Instanz des Kämpfers Informationen über den Inhalt des Inventars.

Die Eigenschaften `MaxHealth` und `Dice` und das Feld `_inventory` sind ebenfalls vor einem Zugriff von außen geschützt, da diese lediglich für die jeweilige Instanz des Kämpfers von Bedeutung sind.

Durch diese Kapselung ist der Umgang mit dem Kämpfer relativ leicht. Man gibt beispielsweise dem Kämpfer die Anweisung, einen Trank zu nehmen, ohne wissen zu müssen, ob sich überhaupt ein Trank im Inventar befindet. Die Klasse „Inventar“ bietet dafür die Methode `Contains()`, mit der dies geprüft werden kann.

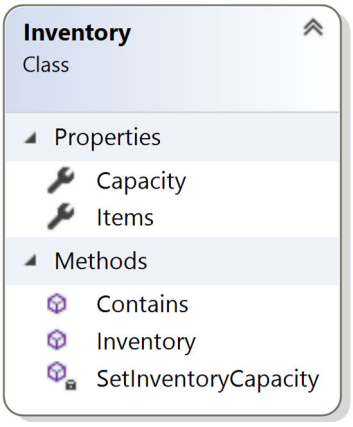


Abb. 5: Die Inventar-Klasse

Auch hier ist die Methode `SetInventoryCapacity()` privat, da nur über den Konstruktor die Größe des Inventars bestimmt werden kann.

Abb. 4: Die Kämpfer-Klasse

Vererbung

Am Beispiel des Heiltranks kann das Prinzip der Vererbung am besten beschrieben werden. Eine untergeordnete Klasse, oder Kindklasse, erbt die Eigenschaften und Methoden ihrer Basisklasse. Dies ist insbesondere dann von Vorteil, wenn eine Basisklasse viele Eigenschaften oder Methoden hat, welche für mehrere abgeleitete Klassen identisch sind.

Jede Klasse in C# leitet sich von der Basisklasse `System.Object` ab, erbt also schon einmal die Methoden dieser Klasse. So ist beispielsweise die Methode `Equals()` und `GetType()` für jede Klasse verfügbar, die von der Basisklasse `System.Object` abgeleitet ist.

Die jeweiligen Klassen `LargeHealthPotion`, `MediumHealthPotion` und `SmallHealthPotion` sind von der abstrakten Klasse `HealthPotion` abgeleitet, besitzen also alle deren Eigenschaften `HpBonus`, `Name` und `Value`.

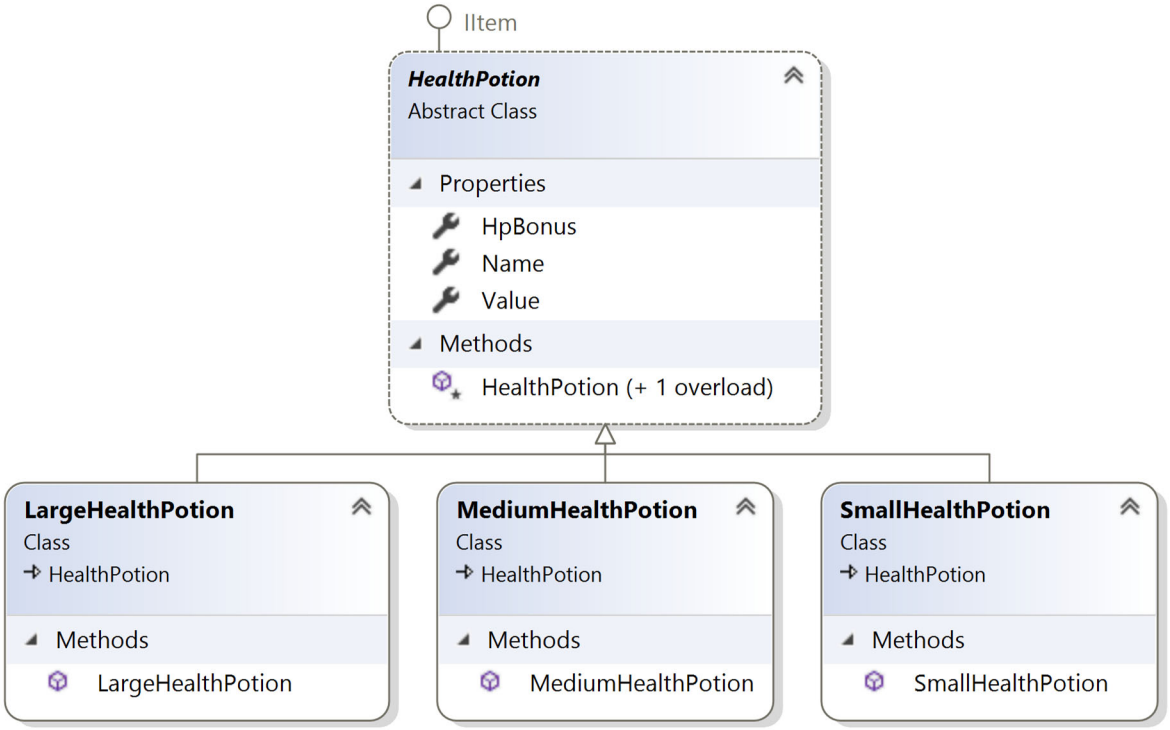


Abb. 6: Vererbung am Beispiel des Heiltranks

```
1. /// <summary>
2. /// Ein kleiner Heiltrank, der 5 LP wiederherstellen kann.
3. /// Abgeleitet von der Heiltrank-Klasse.
4. /// </summary>
5. /// <seealso cref="HealthPotion"/>
6. public class SmallHealthPotion : HealthPotion
7. {
8.   /// <summary>
9.   /// Erstellt einen Heiltrank mit der Stärke 5
10.          /// </summary>
11.          public SmallHealthPotion() : base(5)
12.          {
13.                  Name = "Kleiner Heiltrank";
14.                  Value = 5;
15.          }
16.  }
```

Polymorhpie

Polymorphie (oder „Vielgestaltigkeit“) bezeichnet den Umstand, dass eine abgeleitete Klasse die Methode einer Basisklasse überschreibt und somit „eine andere Form“ erhält. Am Beispiel der `ToString()` – Methode aus der `System.Object`-Klasse kann dies beobachtet werden:

```
1. /// <summary>
2. /// Liefert die Repräsentation des Fighter-Objekts als String
3. /// </summary>
4. /// <returns>(string) Fighter-Objekt</returns>
5. public override string ToString()
6. {
7.     return Name;
8. }
```

Abb. 7: Überschreiben der Methode System.Object.ToString()

Das Schlüsselwort `override` besagt, dass hier eine Methode der Basisklasse überschrieben wird. Statt der Rückgabe aus dieser wird nun eine Eigenschaft `Name` zurück geliefert.

Ist eine Methode mit dem Schlüsselwort `virtual` versehen, so kann sie von einer abgeleiteten Klasse überschrieben werden:

```
1. /// <summary>
2. /// Greift den spezifizierten Gegner an
3. /// </summary>
4. /// <param name="enemy">Der anzugreifende Gegner</param>
5. public virtual void Attack(IEntity enemy)
6. {
7.   int calculatedDamage = Damage + Dice.Roll();
8.   Message = ($"{Name} verursacht {calculatedDamage} Schaden.");
9.   enemy.Defend(calculatedDamage);
10.  }
```

Abb. 8: Die Methode Attack() kann in einer abgeleiteten Klasse mit override überschrieben werden

Hier wäre es beispielsweise möglich, eine Klasse `Mage` abzuleiten, die die Methode `Attack()` mit Hilfe des `override`-Schlüsselwortes anders implementiert.

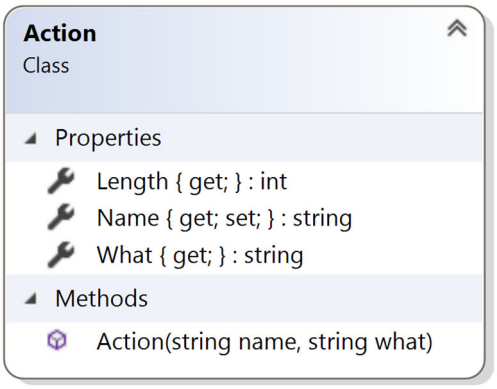
Private Methoden und Eigenschaften sind weiterhin nur in der Basisklasse sichtbar. Sollen diese der abgeleiteten Klasse zur Verfügung stehen, so benötigt man das Schlüsselwort `protected`.

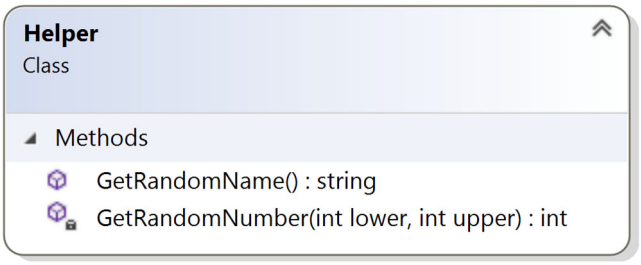
```
1. protected string BarGraphic(int current, int maximum)
2. {
3.   string bar = "";
4.   int total = 20;
5.   double count = Math.Round(((double) current / maximum) * total);
6.   if (count == 0 && IsAlive())
7.   {
```

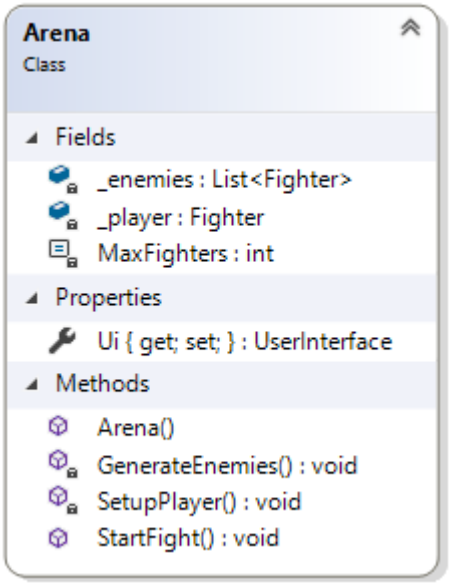
Abb. 9: Das Schlüsselwort "protected" stellt die Methode oder Eigenschaft abgeleiteten Klassen zur Verfügung

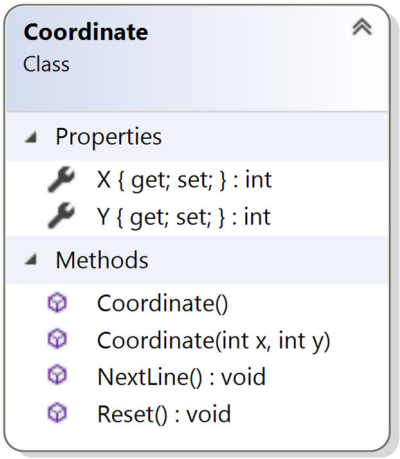
Diese Methode würde der abgeleiteten Klasse Mage zur Verfügung stehen, um beispielsweise eine Manaleiste darzustellen.

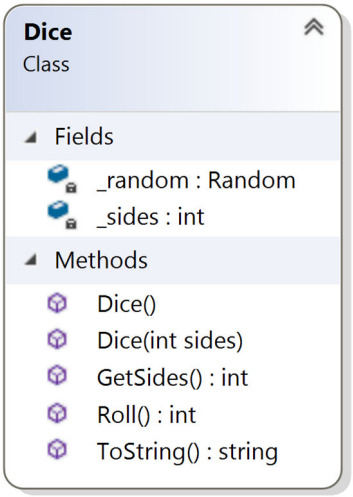
Klassendiagramme

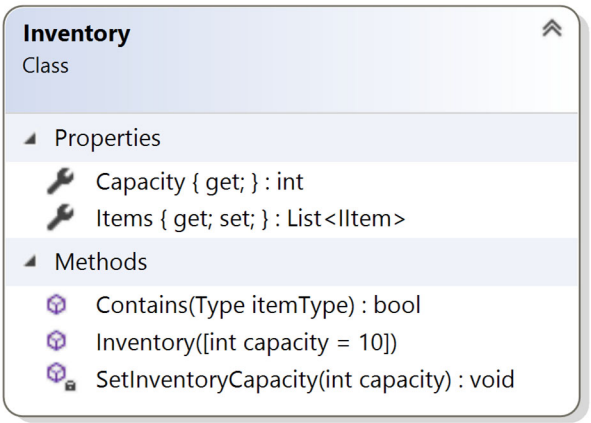

Action
Class
Properties
Length { get; } : int
Name { get; set; } : string
What { get; } : string
Methods
Action(string name, string what)

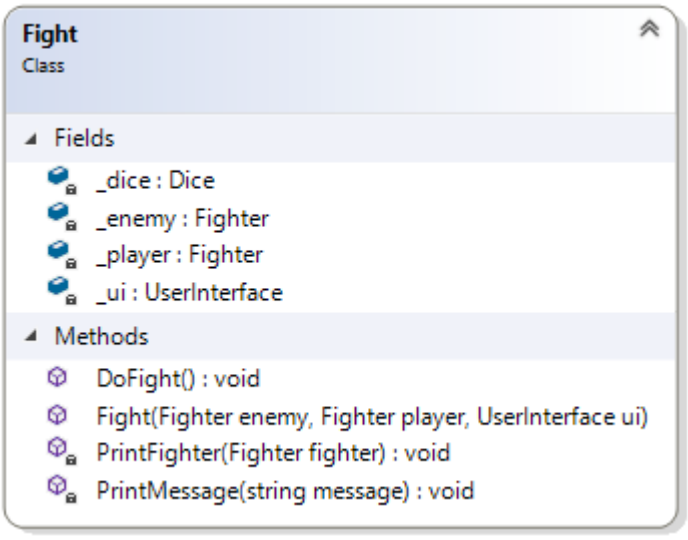

Helper
Class
Methods
GetRandomName() : string
GetRandomNumber(int lower, int upper) : int


Arena
Class
Fields
_enemies : List<Fighter>
_player : Fighter
MaxFighters : int
Properties
Ui { get; set; } : UserInterface
Methods
Arena()
GenerateEnemies() : void
SetupPlayer() : void
StartFight() : void


Coordinate
Class
Properties
X { get; set; } : int
Y { get; set; } : int
Methods
Coordinate()
Coordinate(int x, int y)
NextLine() : void
Reset() : void


Dice
Class
Fields
_random : Random
_sides : int
Methods
Dice()
Dice(int sides)
GetSides() : int
Roll() : int
ToString() : string


Inventory
Class
Properties
Capacity { get; } : int
Items { get; set; } : List<IItem>
Methods
Contains(Type itemType) : bool
Inventory([int capacity = 10])
SetInventoryCapacity(int capacity) : void


Fight
Class
Fields
_dice : Dice
_enemy : Fighter
_player : Fighter
_ui : UserInterface
Methods
DoFight() : void
Fight(Fighter enemy, Fighter player, UserInterface ui)
PrintFighter(Fighter fighter) : void
PrintMessage(string message) : void