Design Patterns

Matthias Mertens Marvin Jansen
Betreuer: David Korzeniewski

20. Mai 2017

1 Vorwort

Mit dieser Ausarbeitung im Rahmen des Proseminars ,,Fortgeschrittene Programmierkonzepte" wollen wir eine kompakte Einführung in das weite Feld der Design Patterns bieten. Dabei zielt diese Arbeit keineswegs auf eine vollständige Abdeckung aller diskussionswürdigen Themen ab, sondern es soll dem Leser eine Zusammenfassung der relevanten Aspekte für einen reibungslosen Einstieg in das Thema geboten werden.
Inhaltlich gliedert sich die Ausarbeitung in einen theoretischen Teil, der das Konzept und die Benutzung von Design Patterns einführen soll (dazu unter 2), gefolgt von einem praktischen Teil, der eine mögliche Klassifizierung aufzeigt und diese mit eingängigen Beispielen beleuchtet (dazu unter 3). Schließlich fassen wir die gesammelten Erkenntnisse zusammen und geben einen Ausblick (dazu unter 4). Wir beziehen uns inhaltlich im Allgemeinen auf ,,Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software" von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, welche in der Literatur oft auch als ,,Gang of Four"(GoF) referenziert werden. Sie gelten durch ihr Werk ,,Design Patterns. Elements of Reusable Object-Oriented Software" als Mitbegründer von Entwurfsmustern.

2 Einführung

2.1 Prinzip von Design Patterns

Für Menschen, die bereits Kontakt mit Programmierung im Allgemeinen hatten, ist das Prinzip von Design Patterns ein äußerst intuitives. Denn im Grunde nutzt man für das eigene aktuelle Problem aus, dass bereits eine große Zahl an anderen Programmierern vor dem selben Problem oder zumindest einer vergleichbaren Problemstellung standen. Es ist sogar denkbar, dass man selbst eine ähnliche Aufgabe bereits in der Vergangenheit gelöst hat. Doch die Erfahrung hat gezeigt, dass sich immer wieder gleichartige Komplikationen ergeben, welche

eine Wiederverwendung von vorhandenen Lösungen verhindern. So zum Beispiel das Fehlen einer ausreichend verständlichen und vollständigen Dokumentation des fremden oder sogar eigenen Codes. Oft ist es der Aufwand und die damit verbundene Zeit nicht wert sich ohne Hilfestellungen in mehrere hundert Zeilen Code einzulesen. Ein weiteres Hindernis liegt nicht am Verständnis, beziehungsweise der Einarbeitung, sondern am Code selbst. Inwiefern wurde dieser in der Vergangenheit sinnvoll designt? Wurden grundlegende Konzepte wie etwa die Datenkapselung berücksichtigt? Wenn dieser nicht gerade von einem erfahrenen objektorientierten Entwickler stammt, wird man sich langfristig auch mit diesem Code verrennen.

2.2 Was sind Design Patterns?

Design Patterns[1] setzen genau dort an und stellen in diesem Kontext ein *Lösungskonzept* dar. Dabei ist besonders der Ausdruck *Konzept* herauszustellen, denn es handelt sich nicht um konkreten Code, den man ohne weitere Arbeit in sein Projekt kopieren kann. Vielmehr handelt sich um Ideen auf denen man erst aufbauen muss. Dies spiegelt sich ebenfalls in der Herkunft dieser Konzepte wieder. Denn diese wurden nicht aktiv entwickelt, sondern meist im Laufe eines konkreten Projekts als Lösung entworfen und dabei als allgemein nützlich befunden. Daraufhin wird aus dem speziellen Kontext heraus ein Konzept abstrahiert, welches in möglichst jeder Domäne Verwendung finden kann (vgl. *Fowler* 1997, Preface xvii).
Darüber hinaus stellen Design Patterns in erster Linie ,,nur" Vorschläge für das geforderte Design dar. Man ist nicht dazu verpflichtet, diese genau so zu nutzen wie die Autoren es vorgesehen haben, sondern man kann nach eigenem Ermessen Komponenten ergänzen und streichen (vgl. *Fowler* 1997, S.12). In diesem Kontext kann man sich Entwurfsmuster als eine Art Schablone vorstellen, die dem Entwickler einen Rahmen und Formen vorgibt, welche man aber für seine Zwecke erweitern und reduzieren kann. Eine eindeutige Definition existiert nicht, aber eine formalere Definition könnte wie folgt lauten:

Definition 1 *,,Design Patterns [sind] Darstellungen kommunizierender Objekte und Klassen, die auf die Lösung eines allgemeinen Designproblems in einem speziellen Kontext zugeschnitten sind."(*Gamma *u. a. 2015, S.29).*

2.3 Vorteile von Design Patterns

Der grundlegende Vorteil, den Design Patterns bieten, besteht darin, dass sie ordentlich dokumentiert und verständlich erklärt niedergeschrieben sind. Zudem handelt es sich um bewährte Konzepte, die bereits in der Praxis erprobt wurden und auch ihren langfristigen Nutzen unter Beweis stellen konnten. Man kann

[1] Auf Deutsch auch als Entwurfsmuster bekannt.

sich also ihrer Zuverlässigkeit sicher sein. Dazu kommt, dass sie insgesamt objektorientierte Konzepte fordern und fördern. Sei es nun zum Beispiel die strenge Kapselung von Daten (vgl. 3.1) oder die Wiederverwendbarkeit von Softwarekomponenten (vgl. 3.3).
Doch Muster leisten noch mehr als das. Sie etablieren unter Entwicklern eine Fachsprache, durch welche die Kommunikation stark vereinfacht wird. So können Ideen und Entwürfe durch die allgemein bekannten Konzepte der Patterns abstrahiert und so einfacher mit Dritten geteilt werden. Schließlich helfen Entwurfsmuster nicht nur in der Praxis, sondern auch in der Theorie. Denn selbst wenn man keine Muster aktiv implementiert, so beeinflussen sie dennoch das Denken und die Herangehensweise an bestimmte Probleme. Man überdenkt automatisch seine eigenen Modellierungen unter ganz neuen Aspekten, welche sich durch die Auseinandersetzung mit Design Patterns und ihrer zugrunde liegenden Idee eröffnet haben (vgl. *Fowler* 1997, Preface xvi).

3 Klassifizierung von Design Patterns

Der erste Kontakt mit Entwurfsmustern stellt sich für viele Programmierer, die eine schnelle Nutzung beabsichtigen, als ernüchternde Erfahrung heraus. Es gibt unzählige Entwurfsmuster von verschiedenen Autoren mit unterschiedlichem Aufbau und Ansätzen, sodass ein unerfahrener Entwickler mit der komplexen Sachlage überfordert ist. Um Herr dieses Werkzeugs werden zu können, benötigt es eine Art Klassifizierung. Dafür gibt es hinreichend viele Merkmale, welche Design Patterns auszeichnen und eine vernünftige Art und Weise bieten, diese einordnen zu können. So könnte man sich etwa eine Zuordnung vorstellen, welche diejenigen Muster zusammen aufführt, die in der Praxis häufig ergänzend zueinander Verwendung finden oder die alternativ eingesetzt werden können.

Wir haben uns an dieser Stelle für eine zweckorientierte Klassifizierung entschieden, welche eine für unsere Zwecke vereinfachte Form von *Gamma* u. a. (2015) darstellt und konzentrieren uns innerhalb dieser auf objektbasierte Patterns. Das bedeutet, dass sich jene Muster, im Gegensatz zu klassenbasierten Patterns, auf die Beziehung zwischen Objekten fokussieren. Diese Klassifizierung sortiert Muster nach ihrem Wirken innerhalb eines Programmes ein und eröffnet eine schnelle Suche, vorausgesetzt man weiß, was man von einem Muster erwartet. Es ergeben sich folgende Kategorien, welche wir im Folgenden genauer betrachten werden:

- Erzeugungsmuster
- Strukturmuster
- Verhaltensmuster

Um einen praktischen Bezug herstellen zu können und die Anwendung der theoretischen Basis in der Praxis zu erleben, wollen wir in diesem Abschnitt Teilaspekte

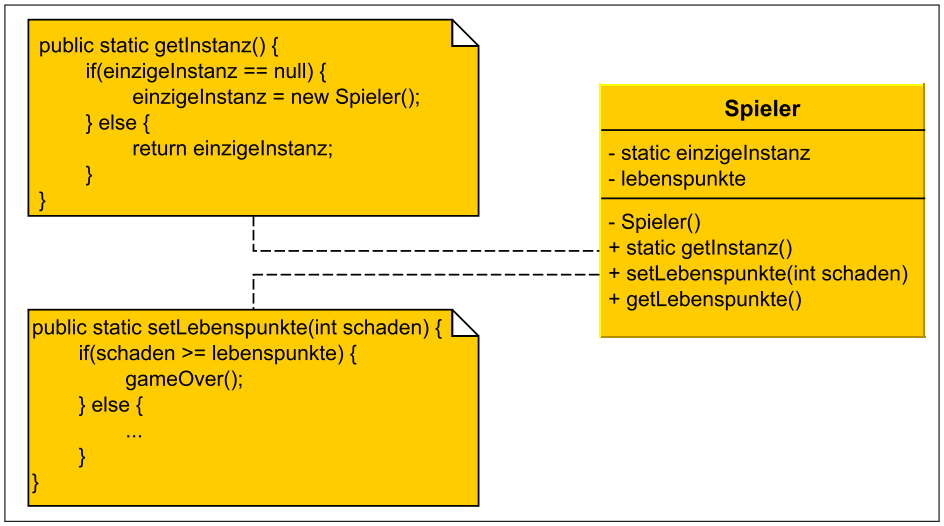

Abbildung 1: Entwurf für Spieler nach Singleton

eines fiktiven Videospiels namens ,,The Legend of Patterns" auf einer stark vereinfachten Ebene modellieren.

3.1 Erzeugungsmuster

Diese Art von Patterns dienen grundsätzlich der Erstellung von Objekten. Sie sollen das System möglichst unabhängig von dem Wissen und der Erzeugung von konkreten Instanzen gestalten. Dazu verbergen sie genau diese Informationen und stellen der Anwendung bloß Schnittstellen bereit. Dabei orientiert man sich stark an dem Konzept der Kapselung. Durch die Auslagerung der Erzeugung in eigene Klassen, bleibt das System möglichst isoliert von einer konkreten Implementierung, sodass ihm im Idealfall bloß jene Schnittstellen der abstrakten Klassen bekannt sind. Durch diese neu geschaffene Unabhängigkeit des Kontextes bleibt viel Spielraum was Details, wie zum Beispiel Zeitpunkt und Beschaffenheit der Erzeugung, angeht.

Für unser Spiel brauchen wir für den Anfang erst einmal einen Spieler. Wir haben entschieden, dass es sich bei ,,The Legend of Patterns" um ein reines Einzelspieler-Spiel handeln soll. Somit haben wir folgende Anforderungen festgelegt:

1. Es darf zu jedem Zeitpunkt nur einen einzigen Spieler geben.
2. Der Spieler hat Lebenspunkte welche niemals unter Null fallen können.

Diese Anforderungen modellieren wir in Abbildung 1 mit einem Erzeugungsmuster namens ,,Singleton". Wie man sieht, hat dieses Muster eine vergleichsweise simple Anwendung. Durch das Privatisieren des Konstruktors ist sichergestellt,

dass von Benutzerseite keine Instanzen kreiert werden können. Der einzige Zugriff von Außen, sowie die indirekte Erschaffung beim ersten Aufruf, geschieht nun reguliert durch die öffentliche Methode `getInstanz()`, welche bei der ersten Benutzung die einzige Instanz von `Spieler` erstellt und diese im Regelfall zurückgibt. Somit ist bereits im Entwurf sichergestellt, dass die Klasse `Spieler` zur Laufzeit nur eine einzige Instanz und zusätzlich einen globalen Zugangspunkt besitzen kann. Da ebenso alle Felder der Klasse `Spieler` privat sind, müssen auch diese per get- und set-Methode erreicht werden, welche mit weiteren Subroutinen ergänzt werden können und somit unsere zweite Anforderung erfüllen.
Jedoch ist dieses Muster behutsam zu benutzen, denn spätere Änderungen, wie die Ergänzung eines Mehrspielermodus in unserem Beispiel, sind in diesem Design nicht mehr möglich und wären mit einer Umstrukturierung der gesamten Modellierung verbunden. Diesem Aufwand wirkt zwar gutes, objektorientiertes Design in Form von loser Kopplung und größtmöglicher Modularität entgegen, widerspricht aber dennoch dem objektorientierten Ansatz der Objekterstellung.

3.2 Strukturmuster

Strukturmuster im Allgemeinen sollen die Handhabung und das Erstellen von umfassenderen Datenstrukturen erleichtern. Dabei setzen sie es sich zum Ziel, durch Kombinieren von Objekten zur Laufzeit ein flexibleres System zu gestalten, als es durch starre und vorbestimmte Klassenhierarchien möglich ist.

In diesem Zusammenhang wollen wir einen weiteren Aspekt von ,,The Legend of Patterns" genauer untersuchen. Um die Welt interessanter zu gestalten, wollen wir neben dem Spieler natürlich auch Gegenstände einführen und modellieren. Dafür haben wir bestimmte Voraussetzungen an einen Gegenstand:

1. Jeder Gegenstand soll eine bestimmte Form besitzen, das bedeutet er muss im Spiel darstellbar sein.

2. Es soll verschieden komplexe Gegenstände geben, die aufeinander aufbauen können.

3. Der Spieler soll nicht den Unterschied zwischen komplexen und simplen Gegenständen bemerken.

Betrachten wir das Design in Abbildung 2 nach dem Erzeugungsmuster ,,Composite". Es wird sich zeigen müssen, ob dieser Entwurf alle gestellten Anforderungen erfüllt. Beginnen wir bei der abstrakten Klasse `Gegenstand`. Diese gibt vor, welche Eigenschaften und Funktionen alle Gegenstände haben müssen, die wir instanziieren möchten. So können wir durch die Methode `getDimension()` auf die (privaten) Informationen zugreifen, die wir beispielsweise zum Rendern des Gegenstandes benötigen und haben somit sichergestellt, dass alle Gegenstände zur Laufzeit darstellbar sind.

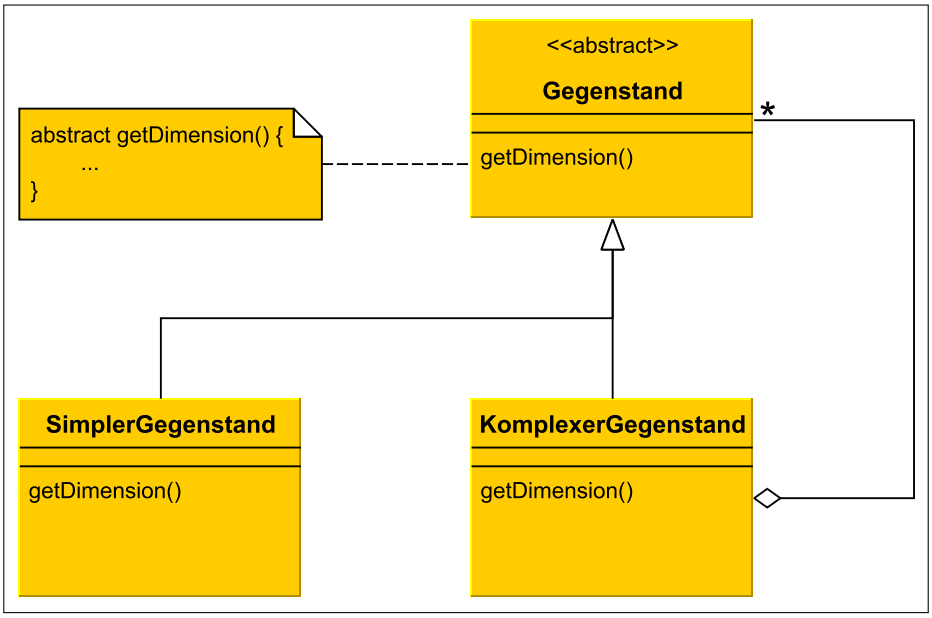

Abbildung 2: Entwurf für Gegenstände nach Composite

Diese Klasse wird von zwei Unterklassen implementiert. `SimplerGegenstand` präsentiert atomare Gegenstände in unserer Welt. Diese stehen für elementare Dinge und lassen sich nicht in weitere Komponenten auseinandernehmen. Sie müssen den in `Gegenstand` gestellten Anforderungen entsprechen, können aber auch eigene logische Funktionen und Attribute integrieren. Beispiele für diese Gegenstände in unserem Kontext sind `Metall`, `Batterie`, `Glühbirne` und `Magnet`. So kann `Metall` nur dargestellt werden (Anforderung), während `Batterie` zusätzlich angibt, ob sie leer ist (logische Funktion).
Zu guter Letzt kommen wir zum Kern dieses Musters. Die Klasse `KomplexerGegenstand` modelliert aggregierte Gegenstände. Anschaulich kann man es so deuten, dass man für komplexere Gegenstände viele atomare Einheiten braucht, um ein Ganzes bilden zu können. So braucht man für eine Taschenlampe mindestens eine funktionierende Batterie, sowie eine intakte Glühbirne. Somit existieren diese Objekte als Kompositionen sowohl aus beliebig vielen atomaren Gegenständen, aber auch aus komplexen Gegenständen. Funktionen von aggregierten Objekten werden anders als bei simplen Gegenständen allgemein gehalten, da sie meist auf den bereits vorhandenen Implementationen von ihren Aggregaten basieren.

Ein Beispiel für eine mögliche Objektkonstellation zeigt Abbildung 3. Leicht zu erkennen ist, dass es sich bei diesen Kompositionen immer um eine Baumstruktur handelt, die nach unten beliebig komplex werden kann. Durch Rechtecke hervorgehoben sind die Blätter des Baumes. Diese stellen die simplen Gegenstände

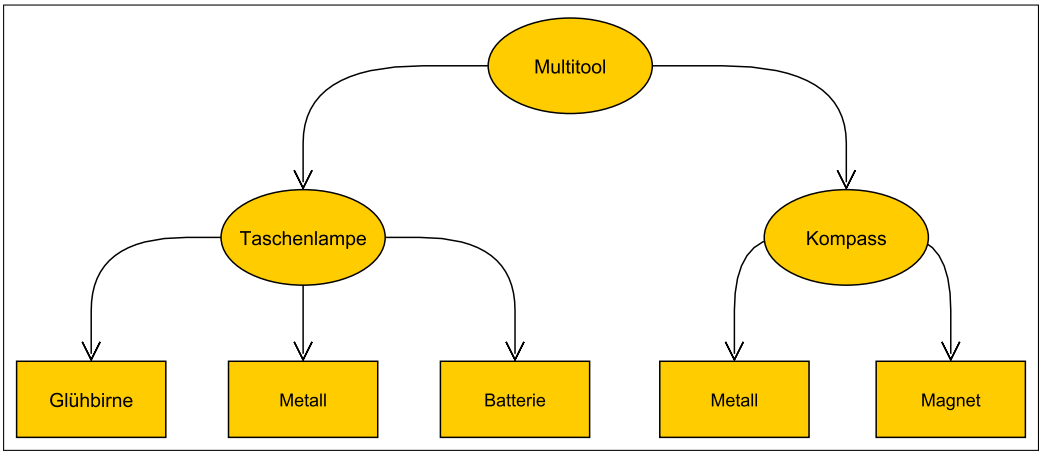

Abbildung 3: Ein mögliches Aggregat von Gegenständen

dar und bilden immer das Ende eines Zweigs. Die komplexen Gegenstände werden durch Ellipsen dargestellt. Nehmen wir als Beispiel die `Taschenlampe`. Da es sich immer noch um einen Gegenstand handelt, muss sie `getDimension()` implementieren. Dies geschieht durch den Aufruf jener Funktion von allen Objekten, welche die `Taschenlampe` ausmachen. Also den atomaren Gegenständen `Batterie`, `Glühbirne` und `Metall`. Daraus setzt sich eine Form zusammen, welche ohne Probleme gerendert werden könnte.
Zusätzlich soll die `Taschenlampe` noch die Funktion `leuchten()` besitzen. Nun verwenden wir die logischen Methoden der Aggregate. So muss zuerst überprüft werden, ob die Instanz von `Batterie` nicht leer ist. Falls dies erfolgreich war, kann mit Hilfe der `Glühbirne` berechnet werden, wie stark oder groß der Lichtstrahl sein kann.

Ein großer Vorteil und unser Motiv für diesen Entwurf ist die Tatsache, dass der Spieler nichts von dieser Mechanik merkt. Er kann zwar unterschiedliche Gegenstände erkennen, doch er kann keine individuellen Klassen von Gegenständen ausmachen, sodass aus der Sicht des Spielers eine einheitliche Behandlung entsteht. Doch darüber hinaus profitieren vor allem Programmierer von dem Muster. Sie vermeiden redundanten Code und damit unnötige Arbeit und erhalten gleichzeitig, durch die strenge interne Modularität, ausnahmslos wiederverwendbare Komponenten.
Ein aktuelles und relevantes Beispiel für dieses Muster kann man in der heutigen Spielbranche wiederfinden. In der *Unreal Engine 4*, welche eine der verbreitetsten und erfolgreichsten Spiel-Engines darstellt (vgl.*Pluralsight* 2015), findet man ein Composite Muster in der Beziehung zwischen einem `Actor` und seinen `Component`'s (vgl. *EpicGames* 2017).

3.3 Verhaltensmuster

Als letzte Klasse von Mustern sollen an dieser Stelle die Verhaltensmuster betrachtet werden. Hier steht die Auslagerung von Zuständigkeiten in eigene, separate Objekte im Vordergrund. Sie kümmern sich um die Zusammenarbeit und die Kommunikation zwischen Objekten zur Laufzeit und versuchen so auch ,,komplexe Programmabläufe erfassbar zu machen" (*Gamma* u. a. 2015, S.279). Besonders wichtig wird auch hier wieder das Konzept der Kapselung. Abschließend wollen wir in ,,The Legend of Patterns" Nicht-Spieler Charaktere (NPC) einführen und insbesondere auf ihr Kampfverhalten eingehen und sinnvoll umsetzten:

1. Es soll verschiedenen Arten von NPCs jederzeit möglich sein, ihr Kampfverhalten zu ändern, um einen möglichst dynamischen Kampf zu ermöglichen.

2. Zudem haben wir bisher nur einige Archetypen von Kampfverhalten entworfen, welche wir umsetzen wollen, sodass beliebig viele Erweiterungen im Nachhinein ohne viel Aufwand möglich sein sollen.

Ein erster Entwurf könnte wie in Abbildung 4 aussehen. Dieser erste Entwurf besitzt einige Besonderheiten, auf die wir nun eingehen möchten. Jede Klasse von NPCs kann zumindest auf alle Fälle angreifen, muss aber ihr eigenes Kampfverhalten selbst implementieren. Das hat zur Folge, dass wenn beispielsweise `Tier` und `Monster` das selbe Angriffsmuster hätten, man den gleichen Code zweimal verwenden müsste, anstatt eine wiederverwendbare Komponente zu schaffen. Man benötigt also redundanten Code, was oft ein Indiz von schlechtem Design ist. Ein beliebter Ansatz um diesem Problem entgegenzuwirken stellt das Prinzip der Vererbung dar. Doch dieses würde, genau wie die Modellierung in Abbildung 4, eine unserer wichtigsten Anforderungen unerfüllt lassen: Die Methode `angreifen` eines NPCs steht zum Zeitpunkt des Kompilierens fest und kann zur Laufzeit nicht mehr geändert und insbesondere nicht mehr erweitert werden. Insgesamt ist dieser Entwurf also für unsere Zwecke ungeeignet.

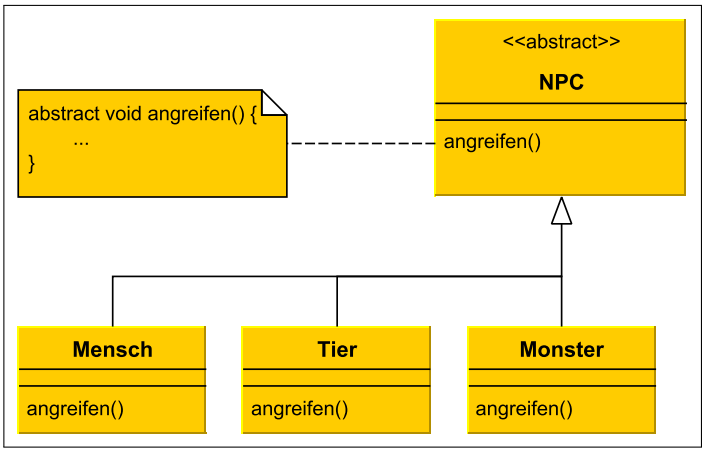

Abbildung 4: Ein erster Entwurf für NPC

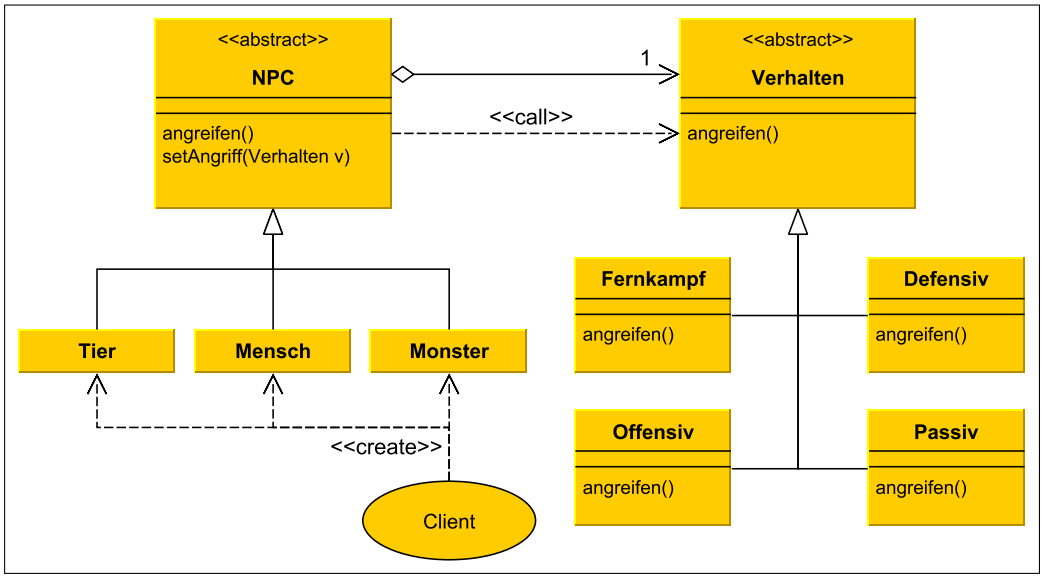

Abbildung 5: Kampfverhalten nach Strategy

Nun betrachten wir in Abbildung 5 eine alternative Modellierung mittels des Verhaltensmusters „Strategy". Offensichtlich hat sich der Entwurf um einige Komponenten erweitert. Beginnen wir bei der bereits bekannten Klasse `NPC`. Auch hier enthält diese die Funktion `angreifen()` und darüber hinaus noch eine weitere namens `setAngriff()`. Ein entscheidender Unterschied ist, dass die genannten Methoden nicht abstrakt, sondern bereits implementiert sind, sodass dies nun nicht mehr in den konkreten Klassen geschieht. Wie das genau funktioniert, werden wir gleich genauer betrachten. Ebenfalls identisch ist die Vererbungshierarchie im linken Teil des Diagramms.
Der entscheidende Unterschied besteht in der Auslagerung der verschiedenen konkreten Kampfverhalten in eigene Klassen und deren abstrakte Schnittstelle `Verhalten`. Die Relationen sind eindeutig festgelegt. Jeder `NPC` (sei es nun Monster, Mensch oder Tier) hat genau ein `Verhalten`. Ein konkretes `Verhalten` besitzt die Schnittstelle `angreifen()`, welche nun genau den Code enthält, welcher in unserem ersten Beispiel noch in den einzelnen `NPC`-Implementierungen zu finden war und der nun in `NPC` aufgerufen werden kann. Welches Kampfverhalten dieses nun konkret darstellt, steht nicht fest und kann insbesondere zur Laufzeit ständig wechseln. Genau zu diesem Zweck gibt es die Methode `setAngriff(Verhalten v)`. Dieser kann man ein beliebiges Verhalten übergeben, sodass die Referenz in `NPC` auf das neue Verhalten aktualisiert wird. Nun ist es meistens so, dass in der `NPC` Klasse ein Default-Verhalten vorliegt, welches angewandt wird, wenn noch nichts manuell überschrieben wurde.
Unsere Anforderungen sind erfüllt, denn man kann wie oben beschrieben jederzeit das Verhalten ändern. Es stellt auch kein Problem mehr dar, das vorhande-

ne Sortiment zu erweitern, da wir die umgesetzte strikte Kapselung zu unserem Vorteil nutzen können. Es genügt, eine neue Implementierung von `Verhalten` zu schreiben. Aber dieses Design birgt noch viele weitere Vorteile, die sich langfristig bezahlt machen. So haben wir die Klassen, die von `NPC` erben, deutlich vereinfacht, ohne eine andere Komponente zu belasten. Das macht den Code nicht nur übersichtlicherer, sondern auch für die Zukunft besser wartbar. Des Weiteren wurde ein Ansatz verfolgt, der gute objektorientierte Programme auszeichnet. Durch die strenge Kapselung wurden maximal wiederverwendbare Komponenten geschaffen. Diese können ohne Mehraufwand in Zukunft übernommen werden, da sie in erster Linie von keiner anderen Komponente abhängen.

4 Fazit

Design Patterns werden in der modernen Softwarearchitektur immer wichtiger. Besonders in Softwarebereichen wie Datenbanken und grafischen Benutzeroberflächen macht man sich heutzutage regelmäßig die unbestreitbaren Vorteile von Entwurfsmustern zu Nutzen (vgl. *Fowler* 1997, S.11). Auf der anderen Seite sei aber ebenso auf bestimmte negative Aspekte hingewiesen, wie beispielsweise die nicht triviale Anwendung von Design Patterns. Angefangen bei der Suche nach dem passenden Entwurfsmuster, über die Bestimmung der Objektgranularität, bis hin zur Implementierung, stellt die Anwendung einen langwierigen Prozess dar, welcher einen großen Teil eines Architekturprozesses ausmachen kann[2]. Doch auch wenn es anfangs eine gewisse Einarbeitungszeit benötigt, so ist der langfristige Gewinn von der Beschäftigung mit Design Patterns unbestreitbar. Sei es nun durch die flexible Nutzung von effizienten Lösungen oder den Erwerb einer neuen Herangehensweise an Problemstellungen.

5 Literaturverzeichnis

EpicGames (2017). *Unreal Engine 4 Documentation*. URL: `https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/index.html` (besucht am 18. 05. 2017).

Fowler, Martin (1997). *Analysis Patterns - Reusable Object Models*. 1. Aufl. Boston: Addison-Wesley Professional. ISBN: 978-0-201-89542-1.

Gamma, Erich u. a. (2015). *Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. Heidelberg: MITP-Verlags GmbH & Co. KG. ISBN: 978-3-826-69904-7.

Pluralsight (2015). *Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose?* URL: `https://www.pluralsight.com/blog/film-games/unity-udk-cryengine-game-engine-choose` (besucht am 20. 05. 2017).

[2] Ausführlich in *Gamma* u. a. (2015, Seite 41-66)