Code Patterns

Materialien zur Dozentur an der school4games Berlin

1

Inhalt

Was sind Code Patterns?

Als Code Pattern oder auch Software Design Pattern bezeichnet man Standardlösungen für immer wieder auftretende Probleme.

Wenn man Code Patterns kennt, kann man vermeiden, in Schwierigkeiten zu geraten, für die andere bereits eine Lösung gefunden haben.
Außerdem lassen sich Probleme im Team vermeiden, denn wenn allen ein verwendetes Pattern bekannt ist, muss man sich in weniger Detailfragen einigen. Ein Kollege, der Code liest, kann sich in der Regel schneller orientieren, wenn er bekannte Patterns wiedererkennt.
Außerdem ist es manchmal möglich, das Verhalten eines von außen nicht einsehbaren Systems nachzuvollziehen, wenn man eine Ahnung hat, welche Patterns wohl verwendet worden sind.

Code Patterns können falsch angewendet aber auch Nachteile mit sich bringen.

Die größte Gefahr ist sicher, dass manche Programmierer dazu neigen, Probleme mit Patterns erschlagen zu wollen und dabei unter Umständen vorschnell zu einer Lösung greifen, die nicht wirklich zum Problem passt.

Außerdem führen viele Patterns dazu, dass enge Beziehungen aufgelöst werden. Beispielsweise sorgt ein Singleton dafür, dass ein Objekt von überall aus erreichbar ist. Dieser Effekt ist einerseits auch gewünscht, er kann andererseits aber den Code schwer nachvollziehbar und somit schwer zu debuggen machen.

In diesem Kurs geht es nicht so sehr um die bekannten Patterns wie sie in der Industrie verwendet werden, sondern eher um übliche Codestrukturen, wie sie beispielsweise in den meisten heutigen Game Engines auftreten.

Der Game Loop

Die **Kernfunktion jedes Spiels**, das Informationen in Echtzeit darstellt, ist der Game Loop. In der Programmierung meint man damit eine Schleife, die so lange durchlaufen wird, bis das Spiel beendet wird.

Diese Schleife führt in jedem Durchlauf mindestens diese Schritte durch:
1. Sie ermittelt, wie viel Zeit seit dem letzten Schleifendurchlauf vergangen ist. Diese sogenannte „delta time“ wird im Code verwendet um zu vermeiden, dass schwankende Frameraten das Spiel langsamer oder schneller werden lassen.
2. Sie updatet den Spielzustand.
3. Sie stellt das nächste Bild dar.

Daraus ergibt sich jedoch eine Schwierigkeit für den Programmierer. Es ist nicht mehr ohne weiteres möglich, einer Spielfigur einen Befehl zu geben wie „bewege dich in drei Sekunden nach da vorne“, weil **jeder Vorgang in Einzelschritte zerlegt werden muss**, die jeweils einen Frame (= einen Schleifendurchlauf) lang sind. Alle Spielobjekte müssen sich also ihren Zustand zwischenspeichern und basierend darauf errechnen, wie er sich während der Zeit, die seit dem letzten Frame vergangen ist, verändert hat.

Der Game Loop ist ein so fester Bestandteil heutiger Spiele, dass die **meisten Engines** diesen ganz beinhalten und dem Programmierer **nicht mehr ermöglichen, ihn selbst zu schreiben**. Diese Engines übernehmen die Kontrolle über den Ablauf des Programms, und der Developer muss nur noch Funktionen schreiben, die dann von der Engine zu vordefinierten Zeitpunkten aufgerufen werden. Da dem Entwickler des Spiels die Kontrolle über einen zentralen Teil des Programms entzogen wird, spricht man hierbei von „**Inversion of Control**“.

Double Buffering

Bei der Berechnung der einzelnen Bilder hat sich ein Pattern etabliert, das als „Double Buffering“ bezeichnet wird.

Ganz allgemein ist damit gemeint, dass ein Vorgang, der aus vielen Einzelschritten besteht, erst auf einen Zwischenspeicher angewendet wird, der dann anschließend als Ergebnis verwendet wird.

In diesem Fall bedeutet das, dass jedes Frame erst auf eine nicht sichtbare Textur gerendert wird, und erst wenn ein Frame fertig berechnet wurde, wird diese Textur mit dem sichtbaren Bildschirminhalt ausgetauscht.

Das ist nötig, da die Grafikkarte den Bildschirminhalt regelmäßig auf dem Monitor darstellt. Dabei nimmt sie keine Rücksicht darauf, ob ein Bild fertig ist oder nicht. Würden alle Bestandteile des Bilds (alle Sprites, alle 3D-Objekte etc.) sofort in den sichtbaren Speicher gerendert, würden immer wieder Bilder auf dem Monitor landen, bei denen die letzten Zeichenbefehle noch nicht ausgeführt worden sind, so dass die letzten Objekte am stärksten flackern würden, da sie in manchen Frames zu sehen sind und in anderen nicht.

Von prozeduralem zu objektorientiertem Code

Am ersten Tag des Kurses haben wir die Befehle, die nötig sind, um eine Spielsituation darzustellen, in einer Liste aufgeschrieben. In jedem Frame muss das Grafik-Device den Bildschirm löschen, alle Objekte darstellen und dann das Bild auf den Bildschirm kopieren.

Dabei haben wir den Vorgang **prozedural** beschrieben – als eine Prozedur von einzelnen Befehlen. Dieser Ansatz **entspricht gut dem, was auf der Hardware passiert**, denn die Grafikkarte kennt keine Objekte, die ihre Position ändern könnten. Sie löscht zu Beginn jedes Frames alles und zeichnet dann das, was von ihr verlangt wird.

Wie im vorhergehenden Kapitel geschrieben muss aber der Zustand des Spiels gespeichert und geupdatet werden, damit der Eindruck entsteht, eine Spielfigur würde sich über den Bildschirm bewegen.

Diese Vorstellung lässt sich **mit Objekten** besser darstellen: **aus Sicht eines Spielers oder Game Designers** malt nicht die Grafikkarte Punkte, sondern eine Spielfigur ändert ihre Position.

Der Code, der eine Spielfigur darstellt, war bereits in der ersten, prozeduralen Version vorhanden. Damit wir aber einen Befehl wie „stelle die Spielfigur dar“ benutzen können statt „male einen Punkt an diese Stelle“ müssen wir den vorhandenen Code in eine neue Klasse verschieben. Wir versehen ihn gewissermaßen mit einem neuen Namen. Diesen Vorgang nennt man „**Kapselung**“. Der Code, der direkt die Grafikkarte anspricht, wurde in der Spielfigur-Klasse gekapselt, so dass wir die Grafikkarten-Befehle nicht mehr zu kennen brauchen um zu wissen, was der Code macht.

Der **prozedurale Code** hatte den Vorteil, dass es **leicht war, ihn zu schreiben**, weil er eine klare Abfolge hat. Der **objektorientierte** hat den Vorteil, dass er **leichter zu lesen** ist, weil der Vorgang in Bestandteile unterteilt wird, die man losgelöst von einander verstehen kann.

Leider macht sich das auch an anderer Stelle bemerkbar: wenn **Game Designer** Features beschreiben, dann passiert das **oft in Vorgängen**. Bei Managementsystemen wie Scrum macht man sich das sogar zunutze und formuliert Aufgaben in sogenannten **User Stories**. Diese beschreiben eine Abfolge von Ereignissen, die im Spiel passieren sollen (z.B. „Der Spieler geht zu einem Händler und klickt ihn an und kann dann ein Schwert kaufen“). Die Features, die als Folge eingebaut werden müssen, sind aber oft besser als Objekte zu verstehen (ein Händler mit Position, ein Shopfenster, ein Schwert-Item).

Ein anderer wichtiger **Vorteil von objektorientiertem Code** ist, dass man besser **im Team arbeiten** kann, wenn der Code auf mehrere Klassen und auf mehrere Dateien verteilt ist.

Setup, Update und Shutdown

Wenn ein neues Objekt erstellt wird, wird als erstes der Konstruktor aufgerufen. Es gibt allerdings gute Gründe, den Konstruktor so weit es geht leer zu lassen und stattdessen eine neue Funktion namens „Setup“ zu schreiben.

Ein Grund ist, dass ein **Konstruktor keinen Wert zurückgibt** (außer dem neuen Objekt). Wenn also etwas in der Konstruktor-Funktion fehlschlagen sollte (eine Grafik kann nicht geladen werden, eine Position ist versperrt o.ä.), dann gibt es **wenige Möglichkeiten, diesen Fehler anzuzeigen**.
Eine Setup-Funktion kann hingegen true oder false zurückgeben, je nachdem ob sie erfolgreich war.

Außerdem ist es möglich, ein Objekt, nachdem es mit Shutdown „heruntergefahren“ wurde, mit Setup **wieder in einen gültigen Zustand zu versetzen und weiterzuverwenden**. Das kann unter Umständen die Performance verbessern, weil weniger Speicher freigegeben und wieder neu angefordert werden muss.

Ein weiterer Grund ist, dass man in Sprachen, die keinen Destructor haben (wie zum Beispiel Haxe oder C#), eine Symmetrie erzeugen kann. **Zu jedem Setup() gehört ein Shutdown()**, und wenn man sich daran gewöhnt hat, **fällt es einem schnell auf**, wenn eine der Funktionen fehlt.

Ein **Nachteil** ist allerdings, dass es für jedes Objekt **eine Phase gibt, in der es zwar existiert aber nicht verwendet werden darf**, nämlich bevor Setup aufgerufen wurde und nachdem Shutdown aufgerufen worden ist. Es ist damit also etwas leichter, ein Objekt falsch zu verwenden.

Da jedes Objekt in jedem Frame aktualisiert und dargestellt werden muss, macht es Sinn, jedem Objekt gleich eine Update-Methode zu geben, die im Game Loop aufgerufen wird.

Game Objects

Das Setup-Update-Shutdown-Pattern aus dem vorherigen Kapitel wird in den meisten Engines so angewendet, dass es eine **Basisklasse für alle Spielobjekte** gibt, die oft GameObject, Entity oder Actor heißt.

Dadurch, dass es diese Klassen gibt, kann sich die Engine um den Game Loop kümmern, denn jetzt braucht sie nicht mehr zu wissen, wie die Objekte im Spiel programmiert sind. Ihr genügt es, dass alle Objekte von dieser Basisklasse abgeleitet sind und somit eine Update-Funktion haben.

In den meisten Engines gibt es außerdem neben einer Update-Methode noch Methoden, die zu anderen Zeitpunkten aufgerufen werden wie OnCollision etc.

Game Objects sind also ein **wichtiger Bestandteil um Inversion of Control** einbauen zu können.

Außerdem erleichtern sie es, objektorientierten Code zu schreiben, der **besser darstellt, woran der Spieler oder Game Designer denkt** (siehe Seite 6).

Game Object Manager

Durch Game Objects kann also die Engine die Spielobjekte verwalten und selbst Funktionen wie Update aufrufen, ohne wissen zu müssen, wie das jeweilige Spielobjekt programmiert ist.

Dafür werden üblicherweise Manager-Klassen verwendet. Diese Klassen haben die **Aufgabe**, die von ihnen verwalteten Objekte einzurichten, zu updaten und gegebenenfalls wieder herunterzufahren.

Das hat zum einen den Vorteil, dass **alle Objekte einer Art in einer Klasse gesammelt** sind. Will man beispielsweise abfragen, wie viele Game Objects es gibt, oder will man alle Game Objects speichern, dann ist das so sehr einfach.

Außerdem lassen sich alle einzelnen **Aufrufe an die verschiedenen Objekte bündeln**. Beispielsweise kann es in einem Spiel wichtig sein, dass alle Gegner geupdatet werden bevor der Spieler geupdatet wird. Das müsste ohne eine Managerklasse direkt im Game Loop so programmiert sein, und jeder, der etwas am Game Loop ändert, müsste das beachten. Daher können Fehler vermieden werden, wenn solche Zusammenhänge in der Managerklasse gekapselt sind.

Managerklassen sind jedoch bei vielen Programmierern nicht mehr gerne gesehen, weil sie oft **für die falschen Sachen missbraucht** werden. Die ungenaue Bezeichnung „Manager“ sorgt oft dafür, dass **Code, für den noch keine wirklich geeignete Klasse existiert**, einfach in die Managerklasse reingeschrieben wird.

Eine Managerklasse ist also nichts schlimmes, es ist allerdings möglicherweise ein schlechtes Zeichen, wenn es neben vielen Managerklassen nur sehr wenige andere Klassen gibt.

Factory Methods

Da ein Game Object Manager die Aufrufe an die in ihm enthaltenen Objekte bündeln soll, sollte er auch **neue Objekte erzeugen** können. Ein Pattern, das das ermöglicht, ist die Factory Method.

Normalerweise werden neue Objekte über den Konstruktor erstellt. Ein neuer Spieler würde zum Beispiel über „new Player()“ erzeugt. Dafür muss allerdings angegeben werden, wie die Klasse heißt, von der eine neue Instanz erzeugt werden soll (in diesem Beispiel Player).
In einem Manager kann stattdessen eine Funktion mit einem Namen wie „createPlayer()“ erstellt werden. Dadurch kann der Code, der den Manager verwendet, den Befehl geben, einen Spieler zu erzeugen, **ohne dabei Details angeben zu müssen, wie welche Klasse** dafür verwendet werden soll.

So könnte man programmieren, dass grundsätzlich jedes unserer Spiele damit beginnt, dass ein Spieler erstellt wird. Wie dieser Spieler jeweils aussieht, steht dann erst im Detailcode in der Factory Method. Auf diese Weise können allgemein gültige Grundprinzipien wiederverwendet werden.

Da ein Konstruktor auch immer nur ein Objekt des angegebenen Typs erstellt, machen Factory Methods den Code flexibler, denn die gleiche Factory Method kann je nach Parameter unterschiedliche Objekte erstellen. So könnte „createPlayer(‚Tank‘)“ eine neue Instanz der Tank-Klasse zurückgeben, „createPlayer(‚Knight‘)“ eine der Knight-Klasse und „createPlayer(‚Ranger‘)“ könnte eine Ranger-Instanz zurückgeben und dabei gleich noch ein Pet erzeugen. Ein Aufruf einer Factory Method kann also sogar mehrere Objekte erstellen, wenn „ein Spieler“ aus mehr als einem Objekt besteht (wie beispielsweise bei einem Ranger mit seinem Pet).

Das Singleton-Pattern

Das Singleton-Pattern sorgt dafür, dass es von einer Klasse **eine Instanz gibt, die wie eine globale Variable** von überall aus zugreifbar ist.

Singletons können sich in Details unterscheiden, insbesondere gibt es **mehrere Varianten, die gleich beim Programmstart, erst beim ersten Zugriff oder manuell erstellt werden**, denn manchmal ist es wichtig, dass alle Objekte von Anfang an verfügbar sind, manchmal ist es wichtig, dass sie nur dann erstellt werden, wenn sie auch wirklich verwendet werden und manchmal ist es wichtig, Kontrolle darüber zu haben, in welcher Reihenfolge sie erstellt werden.

Da das Singleton-Pattern jedoch recht einfach zu verstehen ist, gibt es selten Misverständnisse, wie es zu verwenden ist.

Die **Gemeinsamkeit** aller Singleton-Implementationen ist allerdings, dass es einen **statischen Member** gibt, in dem die eine Instanz gespeichert ist. Dadurch, dass dieser Member statisch ist, und somit nur einmal für die Klasse existiert, lässt sich leicht dafür sorgen, dass an dieser Stelle nur eine gültige Singleton-Instanz gespeichert sein kann.

Das Singleton-Pattern ist ein gutes Beispiel für ein Pattern, das nicht unbedacht angewendet werden sollte, denn die Vorteile können schnell zu **Nachteilen** werden, wenn
- man aus irgendeinem Grund von einem Singleton **mehr als eine Instanz** braucht.
- man verhindern möchte, dass auf eine Klasse **von überall her zugegriffen werden kann**.

Das erste passiert beispielsweise, wenn man mit dem Unity-eigenen Netzwerkcode ein Programm bauen möchte, das sich sowohl als Client verbindet als auch als Server Verbindungen akzeptiert, denn in der noch aktuellen Version gibt es nur ein Network-Singleton, das entweder Client oder Server sein kann.

Das zweite kann vor allem dann zu einem Problem werden, wenn man die Singleton-Klasse ändern möchte und dann nicht mehr so leicht nachvollziehen kann, von wo sie überall bereits verwendet wird.

Lazy und eager evaluation

Wie oben erwähnt, gibt es verschiedene Varianten des Singletons, die sich hauptsächlich dadurch unterscheiden, wann das Singleton-Objekt erstellt wird.

Da das Singleton-Objekt „schon immer“ existiert haben soll (es soll ja schließlich global existieren und nicht erst in unserem Spiel), gibt es keinen „richtigen“ Zeitpunkt, um es zu erstellen. Man kann also nur die Erstellung sofort bei Spielstart „nachholen“, oder man kann sie stattdessen herauszögern bis das Singleton-Objekt benötigt wird.

Wird etwas **so früh wie möglich** durchgeführt, spricht man von „**eager evaluation**“, also „ehrgeiziger Auswertung“. Wird etwas **so spät wie möglich** durchgeführt, spricht man von „**lazy evaluation**“, also „fauler Auswertung“. Da in diesem Fall Objekte erstellt werden spricht man hier auch von „lazy initialization“ bzw. „eager initialization“, also von Initialisierung statt ganz allgemein von Auswertung.

Beide Varianten haben ihre Vor- und Nachteile. Eager initialization hat den Vorteil, dass die Resource erstellt wird bevor das Spiel läuft. **Dadurch wird vermieden, dass das Spiel ruckelt**, wenn zu viele Daten geladen oder zu viel Speicher reserviert werden muss.
Lazy initialization hingegen hat den Vorteil, dass **unter Umständen ein Objekt nie erstellt** wird, wenn es nie angefordert wird, beispielsweise weil sich der Spieler einem Gegnertyp niemals nähert.

Gamestates

Im Verlauf eines Spiels gibt es Momente, wo sich das Verhalten des Programms wesentlich verändert, beispielsweise wenn man ins Menü geht oder in einem klassischen Rollenspiel in einen Kampf gerät.

Der einfachste Ansatz, um das umzusetzen, wäre an einer Stelle zu speichern, in welchem Zustand sich das Spiel befindet, und dann in jeder Funktion etwas anderes zu machen, je nach aktuellem Spielzustand. Beispielsweise würde in einem Rollenspiel beim Update die Spielerposition geupdatet, es sei denn er befindet sich gerade in einem Shop-Menü.

Diesen Ansatz bezeichnet man als „**stateful object**“, denn es gibt ein **Objekt** (in diesem Beispiel das Spiel), **das sich unterschiedlich verhält**, je nachdem in welchem Zustand es ist.

Das hat einige **Nachteile**, denn eine Klasse muss jetzt den Code für mehrere Zustände enthalten um aus ihnen auswählen zu können. Außerdem macht ein und derselbe Befehl unterschiedliche Sachen, was von außen nicht ersichtlich ist.

Eine elegante **Alternative sind Gamestates**. Ein Gamestate ist eine Klasse, die nur **Code für das Spiel in einem Zustand** enthält. Wenn sich der Zustand des Spiels ändern soll, muss nur auf einen anderen Gamestate gewechselt werden. Der wichtigste Vorteil ist, dass dadurch der Code übersichtlicher auf Klassen und Dateien verteilt werden kann.

Messaging

Ein besonderes Problem bei Computerspielen entsteht daraus, dass Befehle oft nicht in derselben Geschwindigkeit abgearbeitet werden können in der sie erteilt werden.

Wenn man beispielsweise in einem rundenbasierten Strategiespiel eine Einheit bewegt und gleich die Runde beendet, dann wird in den meisten Spielen zuerst eine Bewegungsanimation abgespielt bevor die Runde tatsächlich beendet wird.

Dieses Problem tritt in ähnlicher Form an vielen Stellen auf, beispielsweise wenn ein Replay gespeichert werden soll (bei dem ja auch alle Befehle gespeichert und dann viel später ausgeführt werden) oder wenn bei einem Online-Spiel eine Aktion auf dem Server veranlasst aber auf dem Client dargestellt wird.

Eine Lösung stellen Messages dar. **Messages**, in der Netzwerkprogrammierung auch RPCs („remote procedure calls“) genannt, **sind Objekte, die alle Informationen für einen Funktionsaufruf speichern**. Aus einem Aufruf wird also ein Objekt. Im Gegensatz zu einem Funktionsaufruf kann ein Objekt aber **gespeichert oder übers Netzwerk verschickt werden**.

Bildlich gesprochen ist ein Funktionsaufruf so etwas wie ein gesprochener Befehl. Wer sich nicht zur selben Zeit am selben Ort befindet, wird ihn nicht hören. Eine Message ist so etwas wie ein Zettel, auf den der Befehl aufgeschrieben wurde. Dieser Zettel kann in einem Brief verschickt, auf einem Stapel gelagert oder kopiert und archiviert werden.