# Anmerkung:

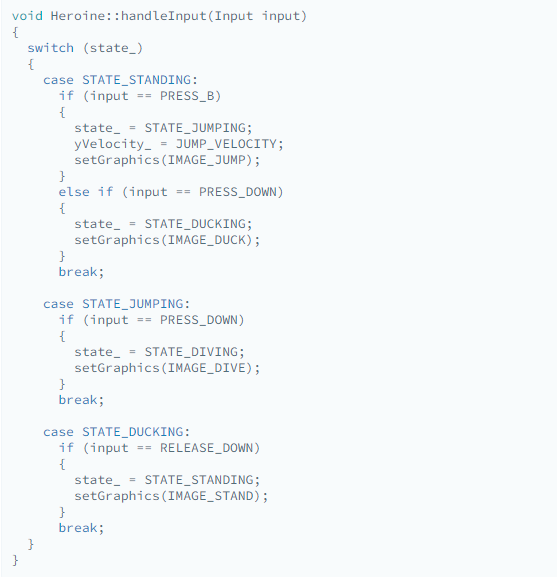
In diesem Dokument befinden sich persönliche Notizen zum State-Pattern, aus denen ich später meinen Vortrag erstellen werde. Dieses Dokument spiegelt die finale Präsentation nicht wieder und enthält möglicherweise weniger oder mehr Informationen als die finale Version. Da es möglicherweise unter die Kategorie „Präsentationsunterlagen“ fällt, reiche ich es sicherheitshalber mit meinem restlichen Portfolio ein.

# Die Idee:

Ein Objekt soll sein Verhalten ändern können, wenn sich sein interner Zustand ändert, **ohne** dass das Hinzufügen von neuen Zuständen das Verhalten existierender Zustände beeinflusst. Das bedeutet auch, dass zustandsspezifisches Verhalten vom Quellcode anderer Zustände unabhängig definiert wird.

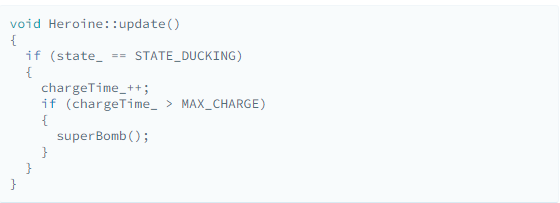
# Konkretes Problem:

(Warum sind die Probleme, die das State-Pattern aufzeigt, ernst zu nehmen)



(Beispiel erläutern)

In dem Fall: Innerhalb der Charakter-Klasse entscheidet ein Switch-Befehl anhand eines Enums, in welchen Zustand (z.B. nach Eingabe) gewechselt werden muss (handleInput-Methode) und welcher Code im jeweiligen Zustand ausgeführt werden muss (Update-Methode).



(Beispiel für die Update-Methode)

Diese Herangehensweise wirft konkrete Probleme auf:

* Wird ein neues Pattern hinzugefügt, müssen sowohl handleInput, als auch Update bearbeitet und angepasst werden. Mindestens ein Enum und zwei cases müssen hinzugefügt werden. Bei vielen Zuständen wird die switch-Anweisung riesig.
* Die Charakter-Klasse enthält für alle möglichen States Texturen, Daten, etc., was die Klasse aufbläht und schnell zu Unübersichtlichkeit führen kann.
* Haufenweise Codedopplung

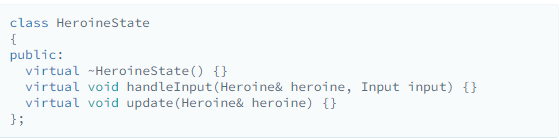
All diese Probleme können mithilfe unseres **Lieblingsparadigmas** gelöst werden: **Objektorientiertheit**.

# Funktionsweise:

So funktioniert das State-Pattern genau:

Das Ziel des State-Patterns ist, das gesamte Verhalten und alle Daten eines Zustands in eine einzige Klasse zu verpacken. Um dieses Ziel zu erreichen, wird eine State-Klassenfamilie gebaut wird. Zunächst wird eine Interface-Klasse erstellt. Für jeden Zustand wird dann eine eigene Klasse als Kind der Interface-Klasse definiert, die das Interface implementiert. Diese konkreten Zustandsklassen definieren das Verhalten des Charakters im jeweiligen Zustand. Das bedeutet in Hinblick auf unser Codebeispiel:

* Für jeden Ort, an dem zuvor eine Switch-Anweisung verwendet wurde, um zwischen Zuständen zu unterscheiden, wird eine virtuelle Methode im Interface erstellt.
* Sämtlicher Code in unseren Case-Anweisungen wandert in die konkreten Zustandsklassen.



Bsp. Für Interface



Bsp. Für einen konkreten Zustand

Wie werden die Informationen der Zustände allerdings abgerufen?

Der Charakter erhält ein neues Attribut: Einen Pointer, der auf seinen aktuellen Zustand zeigt. Statt den Zustand des Charakters in riesigen Switch-Anweisungen zu überprüfen, kann nun direkt auf das Verhalten des aktuellen Zustands verwiesen werden.



Um den Zustand zu wechseln, muss der Pointer lediglich auf ein anderes State-Objekt zeigen.

Dies wirft jedoch eine weitere Frage auf: Wo werden die Objekte, auf die der Pointer zeigt, instanziiert? Dafür können zwei gängige Methoden verwendet werden:

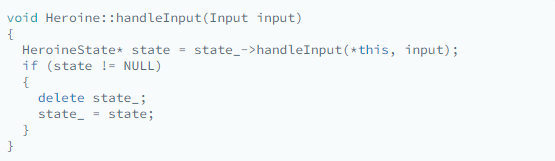
## Statische Zustände

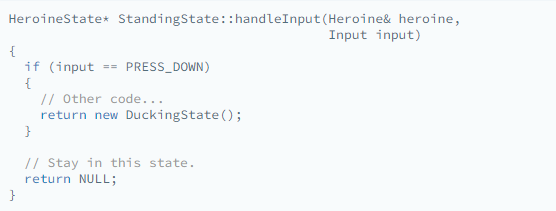
Falls ein Zustand keine neuen Felder verwendet, ergäbe es keinen Sinn, mehr als eine Instanz der Zustandsklasse zu erzeugen, weil jede Instanz sowieso identisch zueinander wäre. Diese statischen Instanzen können nach Belieben an vielen unterschiedlichen Stellen im Code erzeugt werden, beispielsweise in der State-Basisklasse. Dabei sollte natürlich trotzdem darauf geachtet werden, dass ein sinnvoller Ort gewählt wird.

## Instanziierte Zustände

Falls sich in einer State-Klasse bspw. ein Timer befindet und der Zustand (mitsamt Timer) von mehreren Charakteren unabhängig voneinander verwendet wird, wäre es durchaus sinnvoll, mehrere Instanzen dieses Zustands zu erzeugen. Das funktioniert folgendermaßen:

Man gibt einer State-Klasse die Möglichkeit, einen neuen Zustand an die Charakterklasse zurückzugeben, falls der Zustand gewechselt werden soll. Erhält die Charakterklasse einen neuen Zustand, wird der (alte) aktive Zustand einfach mit dem neuen Zustand ausgetauscht. Das kann im Code in etwa so aussehen:





Es sollte erwähnt werden, dass statische Zustände prinzipiell mit weniger Rechenaufwand als instanziierte Zustände verbunden sind, weshalb man, sofern die Möglichkeit besteht, lieber sie verwenden sollte.

Im Grunde war das schon das ganze State-Pattern. Man kann das Ganze aber noch ein wenig erweitern.

# Enter- und Exit-Aktionen

Wie vorhin erwähnt ist das Ziel des State-Patterns, das gesamte Verhalten und alle Daten eines Zustands in eine einzige Klasse zu verpacken. In dem Pattern wie ich es erklärt habe, zeichnet sich allerdings eine unschöne Tatsache ab:

# Warum man das State-Pattern verwenden sollte

Die Vorteile des State-Patterns sind wie folgt:

* Riesige Switch-Anweisungen zur Abfrage von Zuständen werden vermieden
* Übersichtlichkeit:
  + Die Charakter-Klasse bleibt schlank und enthält nur charakterrelevante Daten
  + Alle relevanten Daten und Verhalten eines Zustands lassen sich in seiner Klasse finden
* Neue Zustände können einfach hinzugefügt werden, ohne dass die Charakterklasse geändert werden muss, indem neue Zustandsklassen definiert werden.
* Codedopplung wird reduziert