Konzept Review

Team Gold

\* Einleitung \*

Inhalt

[Technologie 2](#_Toc452444196)

[Plattform 2](#_Toc452444197)

[Framework 2](#_Toc452444198)

[Aufbau des Programms 3](#_Toc452444199)

[Generell 3](#_Toc452444200)

[Zustände 3](#_Toc452444201)

[Aufbau der Software 5](#_Toc452444202)

[Aufteilung in Engine und Game 5](#_Toc452444203)

[Entwurf der Engine 5](#_Toc452444204)

[Component Pattern 6](#_Toc452444205)

[Layer 7](#_Toc452444206)

[Core-Modul 8](#_Toc452444207)

[Behavior-Modul 8](#_Toc452444208)

[Service-Modul 10](#_Toc452444209)

[Standard Services 11](#_Toc452444210)

[Entwurf des Games 12](#_Toc452444211)

[Anhang 13](#_Toc452444212)

# Technologie

## Plattform

Zu Beginn des Projekts mussten wir uns damit beschäftigen, für welche Technologie und Plattformen wir uns zur Umsetzung unseres Spiels entscheiden.  
Zur Wahl standen Unity3D und Java, welche beide gewisse Vorteile, aber auch Nachteile bieten.  
Letzten Endes haben wir uns für Java entschieden, da Unity3D einfach viel zu viel für ein kleines simples Spiel ist, außerdem können wir bei Java alles selbst programmieren.

## Framework

Nachdem die Wahl auf Java fiel war zu klären, welches Framework wir uns zur Erstellung des Spiels verwenden. Hier hatten wir 3 Alternativen:

1. Keine Bibliothek (bzw. nur die Java Standard-Bibliothek)
2. LWJGL (Wrapper um OpenGL)
3. Slick2D (Wrapper um LWJGL)

LWJGL sowie Slick2D bieten die Vorteile, dass sie bereits OpenGL unterstützen, wobei LWJGL nur Zugriff auf OpenGL unter Java liefert und bereits ein Fenster-System (auf Basis von GLFW) besitzt.   
Slick2D bietet nochmals mehr Komfort, da es allgemeine Arbeit abnimmt, die man für jedes 2D Spiel braucht. So existiert bereits eine GameLoop, der Zugriff auf OpenGL ist durch Graphics-Objekte gewrappt und es existieren auch Wrapper um das Fenster-System von LWJGL sowie um das Input-Handling von LWJGL. Der einzige Nachteil gegenüber LWJGL ist, dass der OpenGL-Zugriff über die veraltete statische OpenGL-Pipeline gelöst ist, während wir bei LWJGL die Möglichkeit haben die moderne dynamische Pipeline mit Shadern zu benutzen. Dies ist allerdings mit extrem großem Aufwand verbunden und unser Spiel ist nicht ansatzweise komplex genug, als dass wir wirklich Performance Unterschiede merken würden.  
Aber es bestünde auch noch die Möglichkeit ganz auf OpenGL zu verzichten und einfach Javas integrierte 2D Bibliothek benutzten. Diese ist ähnlich komfortabel zu benutzen wie Slick2D, auch wenn hier noch keine GameLoop und kein State-System existiert. Außerdem ist unser Spiel nicht so aufwändig, dass wir wirklich auf eine Grafikkarte angewiesen sind.

Letzten Endes fiel die Wahl auf Slick2D, da es sehr einfach zu benutzen ist, uns einige Arbeit bereits abgenommen wurde und bereits Erfahrung mit Slick2D im Team vorhanden ist.

# Aufbau des Programms

## Generell

Als grundlegen Aufbau für das Spiel haben wir einen Container, der die einzelnen Zustände des Spiels verwaltet. Im Grunde besitzt der Container die GameLoop, das Fenster und alle Ressourcen und leitet die Aufgeben wie Rendering, Updating usw. an den aktuellen Zustand weiter. Des Weiteren hat jeder Zustand die Möglichkeit das Spiel in einen anderen Zustand zu bringen.   
Ein solches Zustands-System hat den Vorteil, dass man das Programm gut gliedern kann, da jeder Zustand unabhängig von den anderen ist.

Abbildung : Zustandssystem

*In Slick2D gibt es bereits die Klassen StateBasedGame und BasicGameState. Diese werden wir erweitern um ein Zustandssystem zu realisieren.*

## Zustände

Als nächstes galt es zu entscheiden, wie das Programm aufgebaut ist, in welchen Zuständen es sich befinden kann und von welchem Zustand es in welchen anderen wechseln kann.

Das Spiel startet im **Splash-Screen**, dieser wird genutzt um ein Logo oder Ähnliches anzuzeigen, außerdem ist es möglich hier schon bestimmte Ressourcen zu laden.   
Von dort aus gelangt man automatisch in das **Main-Menu**. Hier hat man die Möglichkeit ins Options-Menü zu wechseln, oder das Spiel zu starten.  
Das **Optionsmenü** bietet Einstellungsoptionen sowie eventuell einen erklärenden Text. Aus dem Optionsmenü gelangt man immer in den Zustand zurück, aus dem man gekommen ist.  
Wenn man das Speil startet gelangt man in den ersten **Mini-Game Zustand**. Hier spielt man das Mini-Game, für das man eine gewisse Zeit hat.  
Nachdem man ein Mini-Game beendet hat, gelangt man in den **Post-Game Screen**. Hier wird je nach Erfolg oder Misserfolg eine andere Video-Sequenz oder Ähnliches gezeigt.  
Nachdem diese Anzeige beendet ist wird das **Stats Menü** angezeigt, in dem man den aktuellen Spielstand sehen kann, also verbleibende Leben, Spielzeit und Punkte.  
Je nachdem, ob man keine Zeit mehr übrighat, alle Leben verloren hat, oder noch weiterspielen darf, gelangt man in den entsprechenden nächsten Zustand. Für den Fall, dass man noch weiterspielen darf, gelangt man in den nächsten Mini-Game Zustand.  
Falls man keine Spielzeit mehr hat, gelangt man in den **Time-Over Screen**. Hier wird die Nachricht angezeigt, dass man die Spielzeit aufgebraucht hat und man gelangt wieder ins Hauptmenü.  
Falls man nach einem Mini-Game alle Leben verloren hat, gelangt man in den **Game-Over Screen**. Auch hier wird eine entsprechende Nachricht angezeigt und man kommt wieder ins Hauptmenü.   
Während eines Mini-Games kann man auch immer pausieren und damit kommt man in das **Pause Menü**. Von dort gelangt man zurück ins Spiel, zu den Optionen oder ins Hauptmenü.

Abbildung : Zustände

Splash-Screen

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Main-Menu

Mini-Game

Post-Game Screen

Time-Over Screen

Game-Over Screen

Options

Pause

Stats Menu

# Aufbau der Software

## Aufteilung in Engine und Game

Die Software für das Spiel wird in zwei große Blöcke eingeteilt, die ihrerseits jeweils eine eigene Architektur haben.  
Der erste Block ist die Engine. Sie enthält allgemeine Klassen, die zur Umsetzung von allen ähnlich simplen Spielen wie dem unseren geeignet sind. Die Engine enthält eine simple Implementierung des Component-Patterns, ein Modul zur Unterstützung von Services sowie eine Erweiterung zu dem State-System von Slick2D.  
Der zweite Block ist das Game. Es enthält die spezifischen Components und Services, die wir speziell für unser Spiel benötigen.

## Entwurf der Engine

Generell besteht die Engine aus vier Modulen, die alle einen gewissen Zweck erfüllen und relativ unabhängig von anderen Modulen existieren können. Das Core-Modul liefert eine Implementierung des Component-Patterns und erweitert das State-System von Slick2D. Das Behavior-Modul definiert Schnittstellen, die das Verhalten von Components definieren, z.B. Rendering oder Maus-Input. Das Service-Modul setzt ein Service-Provider-Pattern um. Services sind im Grunde Singletons, bei denen die Schnittstelle von der Implementierung getrennt ist, was es ermöglicht Services einfach auszutauschen und zu debuggen. Das letzte Modul definiert ein paar Services, die man für jedes Spiel gebrauchen kann.

Diese Architektur der Engine sieht noch keine Standard Components vor, diese könnte gegebenenfalls später noch hinzugefügt werden.

Abbildung : Module

*Ein vollständiges UML-Diagramm der Engine ist im Anhang.*

### Component Pattern

Bei dem Component Pattern hat jeder Zustand des Spiels eine Ansammlung von Objekten, die beschreiben, was in den aktuellen Zustand passiert. Jedes von diesen Objekten hat seinerseits eine Ansammlung von Components, die das Objekt definieren. Dabei hat jede Component genau eine Aufgabe, zum Beispiel kann eine Component die Position des Objekts speichern, eine weitere enthält eine Grafik und eine dritte benutzt die Daten dieser zwei Components um die Grafik an der richtigen Stelle auf dem Bildschirm zu zeichnen. Hier ist auch zu sehen, dass jede Component Zugriff auf die anderen Components eines GameObjects haben muss und dass ein GameObject nur eine Instanz jeder von jeder Component haben darf (welche Position würde die Render-Component aus dem Beispiel sonst nehmen?).

Abbildung : Component-Pattern

Für unsere Zwecke verwenden wir aber eine Vereinfachte Version des Component-Patterns. Normalerweise sind GameObjects so aufgebaut, dass sie neben Components auch noch weitere Kind-GameObjects enthalten. Das hat den Vorteil, dass man mit verschiedenen kleineren GameObjects ein großes definieren kann, bei dem die Daten der kleinen Objekte relativ zu dem des komplexen Objekts sind. Die Vorteile, die ein solches komplexeres Pattern liefert, sind aber für unsere Zwecke nicht nötig und würden den Aufbau der Engine nur komplizierter machen.

Eine weitere Alternative wäre gänzlich auf das Component-Pattern zu verzichten. Eine mögliche andere Lösung wäre, dass es ein abstrakte Klasse GameObject gibt, von der alles erbt, was in einem Zustand zu sehen ist. Hier würden dann die Kind-Klassen von GameObject ihre eigenen Daten und Verhaltensweisen definieren und keine Components benötigen.  
Allerdings hat das Component-Pattern den Vorteil, dass man allgemeine Verhaltensweisen sehr gut auslagern und wiederverwenden kann, so hat zum Beispiel jedes GameObject, das ein Bild auf dem Bildschirm zeichnen soll eine ImageRendererComponent. Ohne das Pattern würde hierdurch entweder sehr viel Code dupliziert werden oder es müssten komplizierte Vererbungs-Hierarchien erstellt werden.

Das Component-Pattern, wie wir es umsetzen hat allerdings den Nachteil, dass es bei vielen GameObjects eventuell zu Performance-Problemen kommen kann. Das liegt daran, dass es sehr einfach gehalten wird und keine Optimierungen vorgenommen wurden. Aus diesem Grund werden wir die Performance im Auge behalten und falls wir Probleme haben, werden zum Beispiel die Verhaltens-Methoden der Components zum GameState hinzugefügt anstatt den ganzen GameObjects. Dadurch werden die Anweisungen nicht durch den GameState, dann durch die GameObjects und dann schließlich zu den entsprechenden Methoden der Components weitergeleitet, sondern direkt von dem GameState aufgerufen.

### Layer

Jeder Zustand des Spiels hat mehrere Ebenen, in denen sich GameObjects befinden können. Es existiert der Hintergrund, der üblicherweise ein Bild hinter allen anderen Objekten zeichnet, eine oberste Ebene für die GUI, welche über allen anderen Objekten gezeichnet wird, und mindestens eine Ebene, in der das gesamte Spielgeschehen abläuft.

Background

Game-Layer

GUI-Layer

Abbildung : Layer

Es gibt mehrere Möglichkeiten diese Layer zu realisieren. Eine Möglichkeit ist es, dass jeder Zustand drei Ansammlungen von GameObjects hat, die entsprechend Ihrer Tiefe auf dem Bildschirm nacheinander den Update- und Render-Vorgang ausführen.  
Da sich diese Layer aber nur beim Rendern bemerkbar machen und jede Components selbst wissen sollte, in welches Layer sie gehört werden diese Layer bei uns durch drei verschiedene Render-Methoden realisiert.  
Für jedes Layer Existiert eine eigene Render-Methode, wobei zuerst die für den Hintergrund ausgeführt wird, dann die für das Game-Layer und am Schluss wird die GUI-Render-Methode ausgeführt. Das hat den Effekt, dass der Hintergrund vom Game übermalt wird und die GUI über allem anderen liegt. Dadurch, dass wir nur verschiedene Render-Methoden benutzen, haben wir folgende Vorteile:

* Der Code ist einfacher  
  *Innerhalb des GameStates haben wir nur eine anstatt drei Listen von GameObjects. Außerdem können wir die GameObjects einfach dem GameState hinzufügen und müssen nicht überlegen, in welches Layer wir das GameObject stecken.*
* Es ist effizienter  
  *Für die meisten Sachen sind die drei Layer nicht wichtig. Würden wir die Layer durch verschiedene Listen darstellen, müssten wir über drei Listen iterieren um alle Objekte zu updaten oder um auf Input zu reagieren.*
* Der Code ist besser strukturiert  
  *Dadurch, dass wir nicht entscheiden, in welche Schicht ein GameObject gehört, wenn wir es dem Zustand hinzufügen, können die Components selbst bestimmen, in welche Schicht sie gehören.*

### Core-Modul

Das Core-Modul der Engine besitzt eine Implementierung für das Component-Pattern.  
Es gibt das Interface GameObject, das angibt, dass jedes GameObject die Fähigkeit haben muss Components aufzunehmen und Aktionen auf diesen Components auszuführen. Die Klasse Component erbt von allen Behaviors aus dem Behavior-Modul und definiert Dummy-Implementierung für diese Methoden. Das hat den Vorteil, dass eigen definierte Components alle Arten von Behavior definieren können, aber nicht gezwungen sind welche zu implementieren, die sie nicht benötigen. Des Weiteren existiert die abstrakte Klasse Scene, welche von BasicGameState aus Slick2D erbt. Da jeder GameState durch eine Sammlung von GameObjects definiert wird, wird in der Scene-Klasse diese Sammlung angelegt sowie das Rendering, Updating und Input-Handling an die GameObjects delegiert. Da es umständlich ist jedes Mal, wenn man eine GameObject mit seinen Components braucht, dies Objekt anzulegen und manuell alle Components hinzuzufügen, gibt das Core-Modul die Möglichkeit Prefabs zu definieren. Prefabs sind Factory-Klassen für GameObjects. Sie besitzen eine Methode getGameObject die das Objekt samt seiner Components zurückgibt. Hierbei wird aber nicht festgelegt, auf welche Weise ein Prefab sein GameObject erzeugt. Es ist möglich einen Prototyp zu erzeugen und zu klonen oder jedes Mal ein neues Objekt zu erzeugen. Das Core-Modul beinhaltet auch schon ein Prefab zur Erzeugung eines neuen GameObjects, welches noch keine Components besitzt.

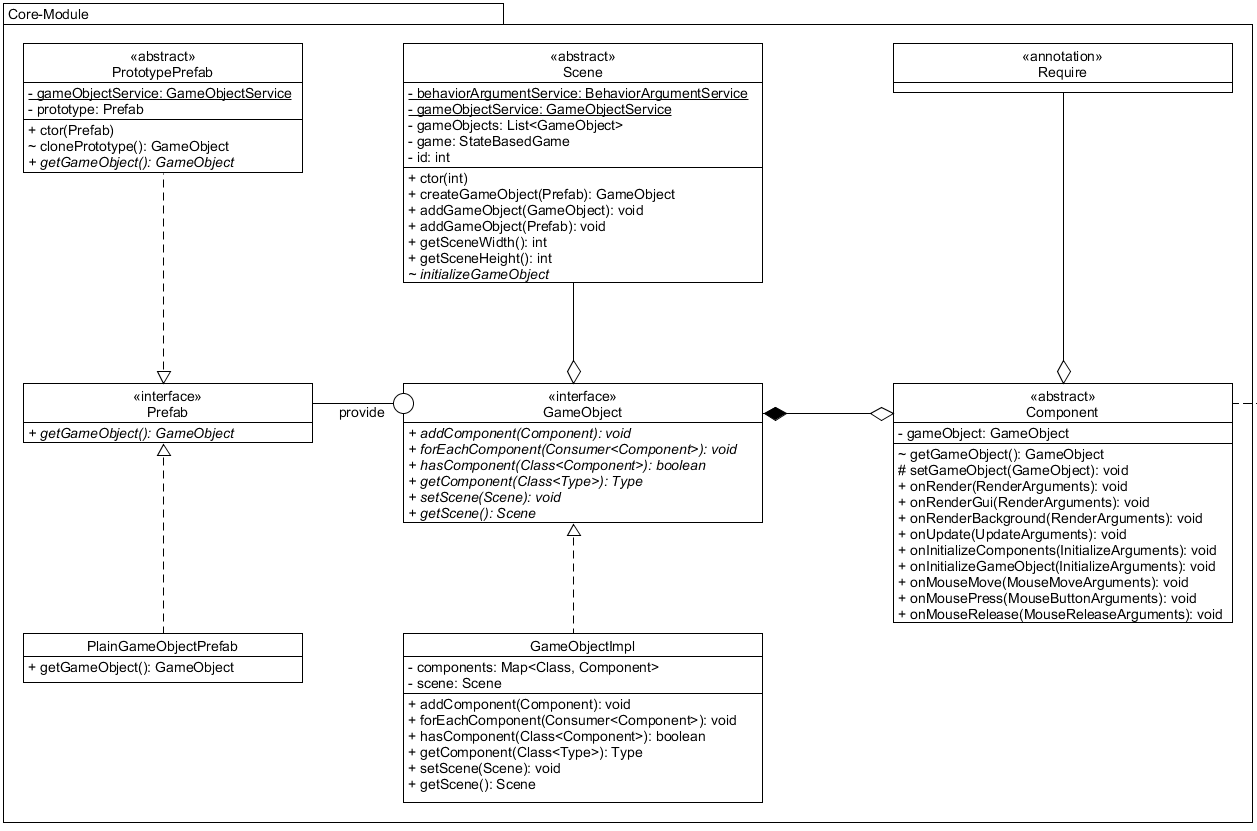


Abbildung : Core Modul

### Behavior-Modul

Das Behavior-Modul spezifiziert die Signaturen der Methoden, die das Verhalten eines GameObjects definieren. Die erste Verhaltensweise, die eine Component definieren kann, ist das Initialisierungs-Verhalten. Sobald ein GameObject erzeugt und zum GameState hinzugefügt wurde, wird dieses Objekt initialisiert. Des Weiteren existiert ein Verhalten für Update-Methoden. In das Update-Verhalten gehören Sachen wie die Physik Berechnung. Hier gäbe es die Möglichkeit, zwischen verschiedenen Arten von Update-Methoden zu unterscheiden, es existiert ein Updaten, dass vor jeden Render-Vorgang ausgeführt wird und ein fixes Update, dass 60 Mal pro Sekunde ausgeführt wird. Für unsere Zwecke reicht das Updaten, dass so häufig wie möglich ausgeführt wird, durch Übergabe der Vergangenen Zeit seit dem letzten Update kann man damit genau so genaue Rechnungen machen, wie mit einem fixen Updating. Das Input-System der Engine beschränkt sich auf den Maus-Input. Je nachdem ob die Maus bewegt oder ein Button geklickt wurde, wird die entsprechende Maus-Input-Methode ausgeführt. Als letztes Verhalten gibt es noch das Render-Verhalten. Hier gibt es drei verschiedene Render-Methoden, die in dem Abschnitt über Layer schon erläutert wurden.  
Die abstrakte Component-Klasse aus dem Core-Modul implementiert all diese Verhaltensweisen mit Dummy-Methoden, so dass die eigenen Komponenten diese Verhalten definieren können.   
Um bessere Lesbarkeit und Unterscheidung zu normalen Methoden zu sichern, beginnen alle Verhaltens-Methoden mit „on“.

Die Argumente, die die Methoden übergeben bekommen sind in Argument-Objekte gekapselt. Das hat den Vorteil, dass man die Argumente der Methoden ändern kann, ohne dass sich die Signatur der Methoden ändert.

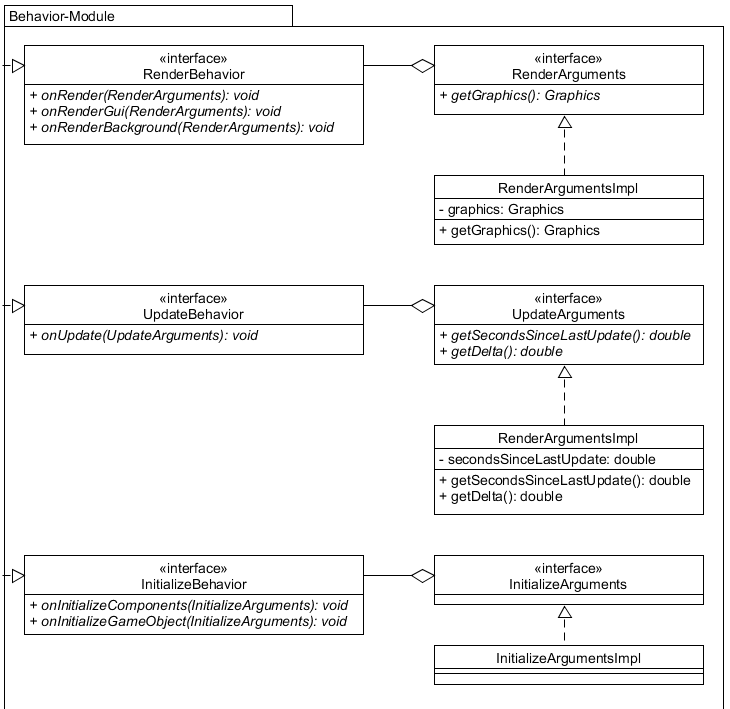


Abbildung : Ausschnitt aus Verhaltens Modul

### Service-Modul

Innerhalb der einzelnen Zustände eines Spiels sind alle Daten in GameObjects gekapselt. Aber es gibt auch Daten, die über alle Zustände hinweg existieren und es gibt Funktionen, die in mehreren Zuständen benötigt werden. Um dies zu erreichen haben wir ein Service-Modul.  
Services erfüllen im Grunde den gleichen Zweck wie Singletons oder statische Klassen, bieten aber eine strikte Trennung zwischen dem Interface eines Service und seiner Implementierung.

Jeder Service wird durch ein Interface definiert, dass von Service erbt. Hier werden alle Methoden deklariert, die der Service haben muss. Dann wird eine Implementierung dieses Service geschrieben und schließlich ein Provider, der diesen Service erzeugt. Den Provider registriert man dann in der statischen Service Klasse, die einen Cache der bereits erzeugten Services besitzt. Sobald man einen Service haben möchte, gibt man bei der Services Klasse an, welche Art von Service man haben möchte. Falls solch ein Service schon erzeugt wurde, bekommt man diesen, ansonsten wird durch den Provider ein neuer erzeugt. Zum einfacheren Gebrauch wird auch eine Inject-Annotation definiert, mit der man Felder versehen kann, sodass die Services Klasse diese Service automatisch setzen kann.

Dadurch, dass die Implementierung von dem Interface getrennt ist, kann man die Implementierung sehr einfach anpassen, man muss nur im Provider ändern, welche Klasse zurückgegeben wird. Der Code ist auch sehr sauber, denn um etwas Ähnliches mit Singletons zu erreichen, müsste das Interface die Implementierungen kennen. Man kann die Services auch einfach debuggen, da man Debug-Implementierungen der Services schreiben kann und diese dann von den Providern erzeugen lassen. Außerdem muss man sich nicht bei jeder Klasse selbst darum kümmern, dass nur genau eine Instanz der Klasse existiert, da die Services Klasse das durch ihren Cache übernimmt.

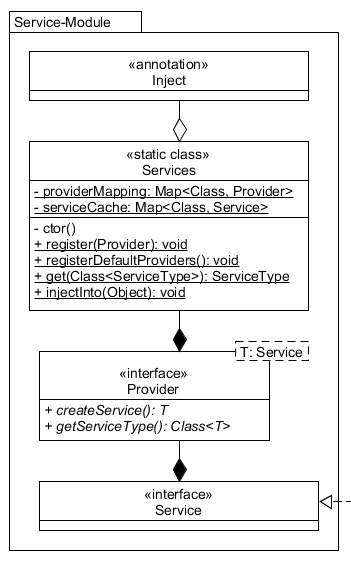


Abbildung : Service Modul

### Standard Services

Es gibt Funktionalität, die in jedem Spiel gebraucht wird, das mit dieser Engine entwickelt wird. So muss es einen Service geben, der GameObjects erzeugt, einen Service der Ressourcen lädt und cached, sowie interne Hilf-Services.

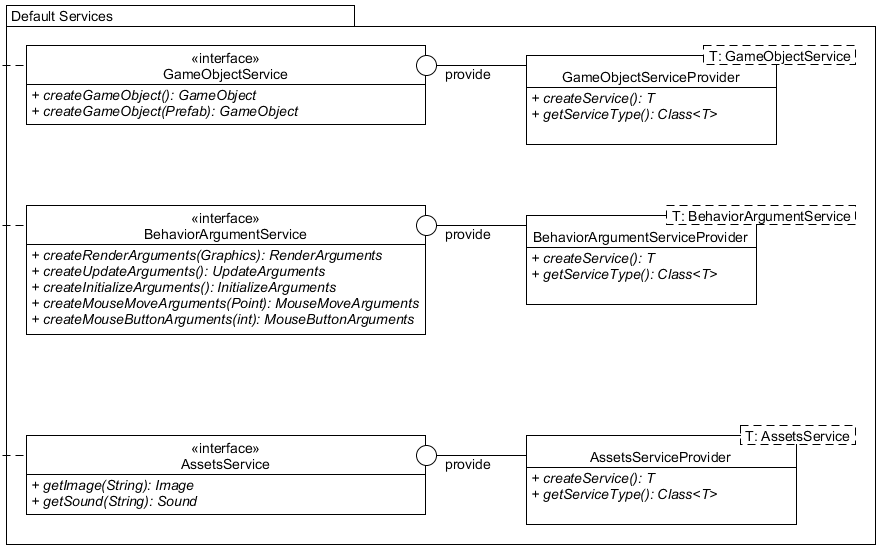


Abbildung : Standard Services

## Entwurf des Games

In der Engine liegt die Grundfunktionalität, die jedes Spiel braucht. Die spezifischen Sachen für unser Spiel liegen im Game. Dieses hat einen sehr simplen Aufbau.

Auf der obersten Ebene liegen die Game-Klasse, die einen Container für die GameStates darstellt, die Main-Klasse sowie ein Interface zur Speicherung der State-ID-Konstanten.  
Es folgen Unterpakete für allgemeine Components und GameObjects, die von allen GameState gebraucht werden können. Dies könnte zum Beispiel eine ImageRendererComponent sein oder eine CountdownComponent. Auf dieser Ebene liegt auch ein Unterpaket für die Services, welche a generell von allen GameState benötigt werden.  
Weitere Unterpakete sind ein Paket pro GameState, mit jeweils Unterpakten für spezifische GameObjects und Components. Dies könnten zum Beispiel die Wasserhähne für das Wasserhahn-Spiel sein.

Abbildung : Paketstruktur Game

# Anhang

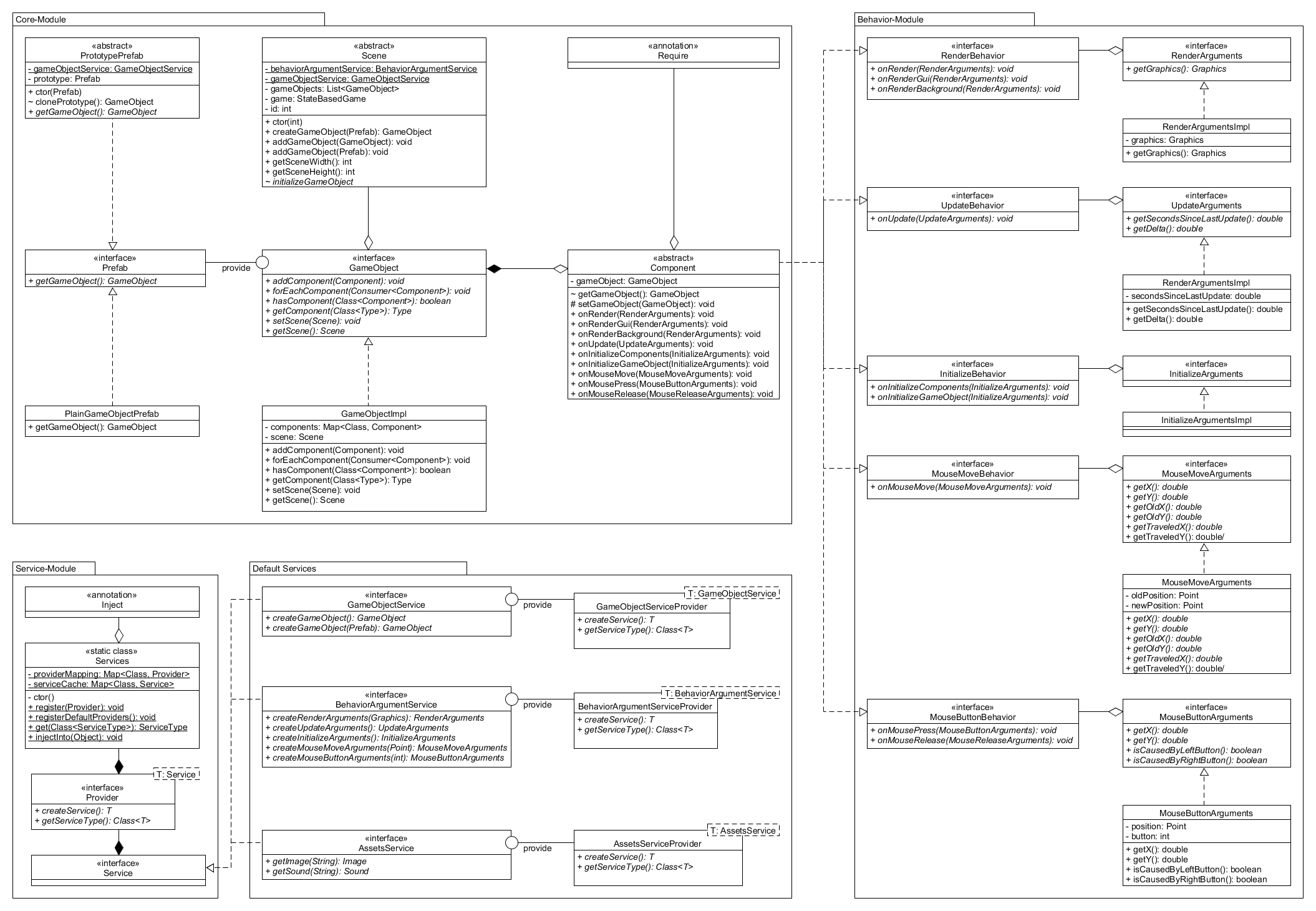


Abbildung : Vollständiges UML-Diagramm Engine