Fakultät Informatik

KOM-3 Objektserialisierung

Schriftliche Ausarbeitung im Studiengang Informatik

vorgelegt von

Leon Breidt

Matrikelnummer 377 2588

Betreuer: Prof. Dr. Ingo Scholz

Kurzdarstellung

In dieser Ausarbeitung werden die Konzepte und Methoden der Objektserialisierung untersucht. Diese sind für die Effizienz und Zuverlässigkeit bei der Entwicklung der festgelegten Spielideen von entscheidender Bedeutung. Beginnend mit einer Definition der Objektserialisierung, werten verschiedene Verfahren und Techniken dargelegt. Der Umgang mit referenzierten Daten und die Implementierung von Kompressionsmethoden zur Optimierung der Datenübertragung werden ebenfalls erklärt.

Ein weiterer Teil der Arbeit ist der Vergleich zwischen klassischen, strukturierten Datenformaten wie JSON und XML und effizienteren, auf Leistung ausgerichteten Formaten wie Protobuf, Cap'n Proto und Flatbuffers. Diese Diskussion dient als Grundlage für die Entscheidung, welche Bibliotheken für die beiden Projekte Park Chase und Into The Sky verwendet werden

Abschließend wird die Umsetzung der gewählten Verfahren für die beiden Projekte beschrieben. Neben einer ersten Implementierung sind hierbei auch die Testfälle wichtig, mit der die Funktionalität der Implementierungen sichergestellt werden.

Inhaltsverzeichnis

Kapitel 1

Theoretische Grundlagen

1.1 Begriffsdefinition und Anwendung

Der Begriff "Objektserialisierung" beschreibt in der Informatik den Prozess, der Datenstrukturen in eine Reihe an Bytes übersetzt. Dies ist nützlich, um Objekte in eine Datenbank oder in eine Datei zu persistieren. Außerdem können Objekte so über ein Netzwerk übertragen werden, was eine der Hauptanwendungen ist. [Lore 17]

Das Gegenstück, also Objekte aus einem Datenstrom wieder aufzubauen, nennt sich Deserialisierung.

1.2 Verfahren zur Serialisierung

Es gibt eine große Bandbreite an Verfahren zur Serialisierung. Die Herausforderung ist es, für die hier vorliegenden Projekte die optimale Wahl zu treffen. Eine solche Wahl muss mehrere Eigenschaften optimieren. [Dani 20]

- Bandbreiteneffizienz
- Speichereffizienz
- Geschwindigkeit des Serialisierungsprozesses
- Erweiterbarkeit
- Interoperabilität

Neben der Auswahl eines geeigneten Protokolls, kann auch die Anwendung zusätzlicher Techniken, wie Delta-Serialisierung, sinnvoll sein. Außerdem ist die Vorauswahl von Daten wichtig. Man muss sich also im Vorhinein Gedanken darüber machen, welche Daten überhaupt in welchen Intervallen übertragen werden müssen.

1.3 Umgang mit referenzierten Daten

Daten, die über ein Netzwerk empfangen werden, beziehen sich oft auf Daten, die bereits in der Vergangenheit übertragen wurden, oder die noch gesendet werden. Es gibt zwei Herangehensweisen, die unabhängig von der Serialisierungsart möglich sind: Einbettung von Datenstrukturen oder Referenzierung über eindeutige Kennzeichnungen.

Beim Einbetten von Datenstrukturen werden die zu referenzierenden Daten direkt mit übertragen. Das bedeutet, dass insgesamt möglichst alle relevanten Daten gemeinsam übertragen werden. Dies hat den Vorteil, dass keine dauerhafte Speicherung der übertragenden Daten nötig ist. Die Daten können direkt und vollständig als Block übertragen und verarbeitet werden. Der Nachteil ist, dass so automatisch größere Datenpakete übertragen werden müssen. Dies dauert also insgesamt länger und man muss mit der Verarbeitung warten, bis alle Daten übertragen sind.

Wenn die Daten untereinander referenziert werden, so geschieht dies durch einen eindeutigen Kennzeichner. Oft werden hierfür einfache Zahlenwerte benutzt, aber auch die Verwendung von UUIDs (Universally Unique Identifiers) ist sehr geläufig. Diese Werte sind entweder in ihrem Gültigkeitsbereich oder sogar ganz allgemein ein eindeutiger Kennzeichner für eine Datenstruktur. Wenn man nun für ein Attribut einer Datenstruktur diesen Wert einträgt, so kann der Empfänger die zu diesem Wert gehörende Datenstruktur separat akquirieren. Diese wird dann entweder angefordert oder wurde in der Vergangenheit bereits übertragen und wird aus dem Speicher heraus verwendet.

1.4 Kompression

Um die Menge der zu übertragenden Daten zu reduzieren, also um den Bandbreitenbedarf zu verringern, werden Daten vor dem Übertragen oft komprimiert. Es gibt hierzu allgemeine Verfahren wie GZIP, allerdings auch spezielle Formate wie JPG und PNG, die Bilddaten komprimieren. Je nach Datenart sind also unterschiedliche Verfahren empfehlenswert. Besonders Bilddaten, die viel Speicher einnehmen, sollten effizient komprimiert werden, um die Netzwerklast so gering wie möglich zu halten.

Verfahren wie Protobuf erlauben es, die Daten direkt komprimiert zu übertragen, ohne dass auf der Empfängerseite eine umständliche Dekomprimierung nötig wird. Neben Protobuf gibt es weitere Technologien wie Thrift oder Avro, die ähnliche Vorteile bieten und in verteilten Systemen und bei der Kommunikation zwischen Microservices zur Anwendung kommen. Diese Technologien sind besonders für die Verwendung in “Park Chase“ interessant.

Es gibt mehrere Techniken, die zur Kompression verwendet werden können. Neben klassischen Verfahren, die oben erwähnt wurden, ist es bei vielen Daten sinnvoll sich über die

Notwendigkeit der Übertragung Gedanken zu machen. Besonders in Software, in der Latenz eine Rolle spielt, ist die Reduktion der zu übertragenden Datenmenge ein wichtiger Beitrag zur Optimierung.

Kapitel 2

Datenformate

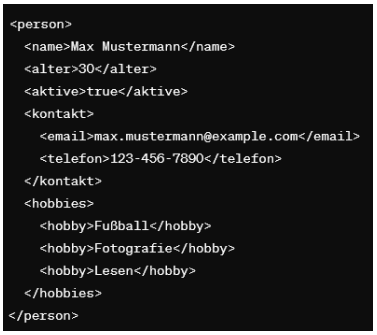
Grundsätzlich unterscheidet man zwischen binären und strukturierten Formaten. Binäre Formate vereinbaren die Struktur der zu übertragenden Daten im Vorhinein und übertragen anschließend einen Binärdatenstrom. Strukturierte Verfahren hingegen senden die Struktur der Daten bei jeder Übertragung mit. Diese Formate basieren meist auf einem Textformat und sind daher wesentlich leichter zu debuggen. Dieser Vorteil wird jedoch mit einer erhöhten zu übertragenden Datenmenge erkauft.

2.1 Strukturierte Datenformate

JSON (JavaScript Object Notation), XML (Extensible Markup Language) und YAML (YAML Ain't Markup Language) sind führende Datenformate, die für den strukturierten Datenaustausch über das Internet entwickelt wurden. JSON, entstanden aus der JavaScript-Programmiersprache, wird seit den frühen 2000er Jahren hauptsächlich für den Datenaustausch zwischen Servern und Webanwendungen sowie in Konfigurationsdateien und NoSQL-Datenbanken wie MongoDB verwendet. Es ist effizienter und einfacher zu handhaben als XML, welches seit den späten 1990er Jahren von W3C standardisiert wurde und sich durch eine strenge, erweiterbare Struktur auszeichnet, ideal für komplexe Anwendungen wie Unternehmenssoftware. YAML, eingeführt im Jahr 2001, bietet eine einfache Struktur, die in Konfigurationsdateien weit verbreitet ist und eine klare, effiziente Organisation von Daten ermöglicht.

```
{
  "name": "Max Mustermann",
  "alter": 30,
  "aktive": true,
  "kontakt": {
    "email": "max.mustermann@example.com",
    "telefon": "123-456-7890"
  },
  "hobbies": ["Fußball", "Fotografie", "Lesen"]
}
```

Abbildung 2.1: JSON-Daten



```
<person>
  <name>Max Mustermann</name>
  <alter>30</alter>
  <aktive>true</aktive>
  <kontakt>
    <email>max.mustermann@example.com</email>
    <telefon>123-456-7890</telefon>
  </kontakt>
  <hobbies>
    <hobby>Fußball</hobby>
    <hobby>Fotografie</hobby>
    <hobby>Lesen</hobby>
  </hobbies>
</person>
```

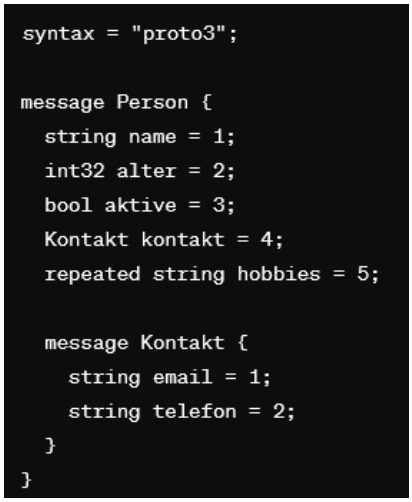
Abbildung 2.2: XML-Daten

```
name: Max Mustermann
alter: 30
aktive: true
kontakt:
  email: max.mustermann@example.com
  telefon: "123-456-7890"
hobbies:
  - Fußball
  - Fotografie
  - Lesen
```

Abbildung 2.3: YML-Daten

2.2 Binäre Datenformate

Protocol Buffers (Protobuf), Cap'n Proto und Netcode for Game Objects (NfGO) sind binäre Datenformate für effiziente Datenübertragung in kritischen Anwendungsbereichen. Protobuf, von Google entwickelt, optimiert den Datenaustausch in verteilten Systemen und mobilen Anwendungen durch Codegenerierung aus .proto Dateien, bietet strikte Typsicherheit und Kompatibilität. Cap'n Proto, eine Weiterentwicklung von Protobuf, ermöglicht das Lesen von Daten ohne Umwandlung in eine Zwischenrepräsentation, wodurch die Leistung verbessert wird. NfGO wird in Multiplayer-Videospielen verwendet, die mit Unity entwickelt wurden. Es unterstützt synchronisierte Interaktionen durch effiziente Serialisierung und Differential Compression, um Netzwerkeffizienz und Spielerlebnis zu optimieren.

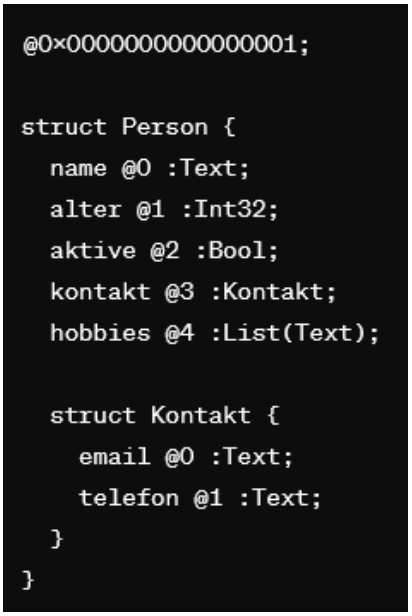


```
syntax = "proto3";

message Person {
  string name = 1;
  int32 alter = 2;
  bool aktive = 3;
  Kontakt kontakt = 4;
  repeated string hobbies = 5;

  message Kontakt {
    string email = 1;
    string telefon = 2;
  }
}
```

Abbildung 2.4: Protobuf Datenstruktur



```
@0x0000000000000001;

struct Person {
  name @0 :Text;
  alter @1 :Int32;
  aktive @2 :Bool;
  kontakt @3 :Kontakt;
  hobbies @4 :List(Text);

  struct Kontakt {
    email @0 :Text;
    telefon @1 :Text;
  }
}
```

Abbildung 2.5: Cap'n Proto Datenstruktur

2.3 Serialisierung in Unity

Unity bietet eine Reihe an Möglichkeiten an, mit denen Entwickler ihre Daten serialisieren können. Viele dieser Methoden, wie NfGO, sind jedoch vollständig in das Unity-Universum integriert und können daher nicht außerhalb verwendet werden. Insgesamt sind für beide Projekte also einige Basiskonzepte vorhanden, die in beiden Projekten verwendet wurden.

2.3.1 Byte-Arrays

Ganz allgemein haben wir uns in den Projekt-Teams bei der Definition der Interfaces für die verschiedenen Aufgabenabschnitte darauf geeinigt, dass Objekte in Byte-Arrays zu serialisieren sind. Diese Byte-Arrays werden dann von der Netzwerkschnittstelle übertragen. Aus diesem Grund zielt die Umsetzung in beiden Projekten darauf ab, Objekte in Byte-Arrays zu übersetzen und zurück.

Kapitel 3

Bibliotheken und Assets

3.1 Into The Sky

Aufgrund der notwendigen geringen Latenz ist es bei MMOFPS-Spielen empfehlenswert, ein Binärformat zur Serialisierung von Datenstrukturen zu verwenden. Für Into The Sky habe ich mich daher für die Verwendung von Protobuf entschieden.

Im Vergleich zu neueren Verfahren wie Cap'n Proto ist deutlich mehr Dokumentation vorhanden, was die Arbeit an dem Projekt beschleunigt. Im Gegensatz zu Formaten wie JSON ist Protobuf im Hinblick auf die Datenpakete deutlich effizienter. Die Verwendung eines kompakten Binärformats verringert die zu übertragene Datenmenge erheblich. Der Nachteil von Strukturierten Formaten wie JSON ist, dass Daten wie Feldnamen oder Klassennamen mit übertragen werden. Das macht die Daten zwar leichter wartbar, in diesem Kontext ist allerdings die Effizienz von höchster Priorität.

Der Ansatz, der zu Beginn der Implementierung vorgelegen hat, basiert auf Reflection. Protobuf ist als etabliertes Verfahren jedoch vorzuziehen und bietet diverse Optimierungen, die durch einen Eigenbau nicht erreicht werden können. In der Präsentation zu diesem Thema habe ich ursprünglich vorgeschlagen Netcode for Game Objects zu verwenden. Diese Bibliothek ist jedoch nicht mit den Architekturvorschlägen und den ersten Umsetzungen kompatibel.

3.1.1 Entwicklung des Proof-of-Concept

Der Prozess, um ein Proof-of-Concept zu realisieren, ist einfach und geradlinig. Als erstes wird ein neues Projekt in Visual Studio angelegt. Dabei handelt es sich um eine Class Library. Der Unity-Version ist zu entnehmen, dass die Objekte den .NET Standard 2.1 unterstützen. Diese Zielversion wird auch für die Class Library verwendet. Über NuGet wird Protobuf installiert, dabei ist die aktuellste Version benutzbar (zum aktuellen Zeitpunkt 3.26.1). Anschließend wird die Struktur der Netzwerkbibliothek angelegt.

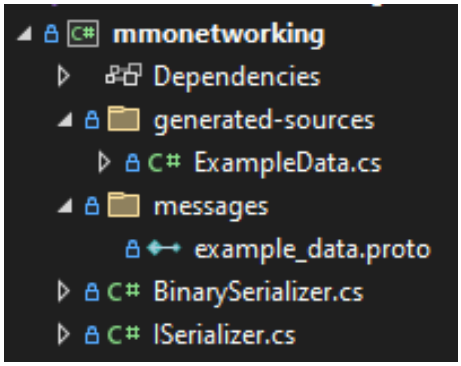


Abbildung 3.1: Struktur der Netzwerkbibliothek

Der Ordner `generated-sources` wird zunächst leer gelassen. Dort landen die generierten C#-Dateien von Protobuf. Um die Dateien zu erstellen muss zunächst `protoc` installiert werden. Anschließend muss das Bauen der Protobuf Dateien in den Build-Prozess von Visual Studio eingebunden werden. Dies geschieht, indem man den folgenden Befehl als Build-Event einfügt:

protoc -I="$(ProjectDir)messages"
–csharp_out="$(ProjectDir)generated-sources"
"$(ProjectDir)messages.proto"

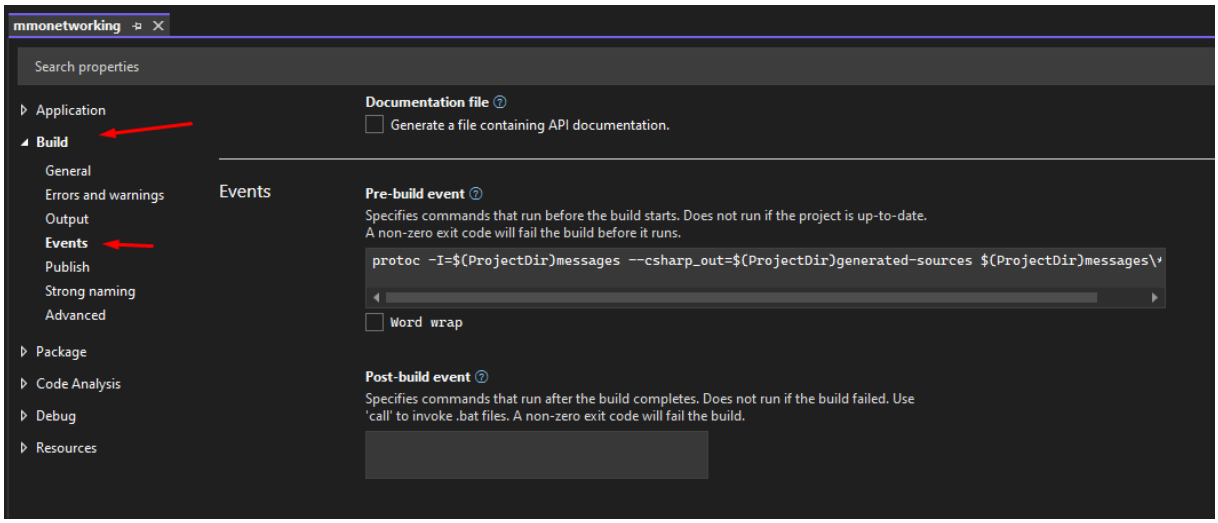


Abbildung 3.2: Build-Befehl

Dieser Befehl kompiliert die Proto-Dateien im messages Ordner und erzeugt die C# Quelldateien im generated-sources Ordner. Durch das Build-Event wird dies vor jeder Kompilierphase durchgeführt.

Nach dem Erstellen einer Muster-Datei zur Datenübertragung die einige Beispielfelder enthält, wird diese zu einer C#-Datei kompiliert. Nun kann ein einfacher Serialiser geschrieben

werden, der die von Protobuf bereitgestellten Methoden nutzt. Im Anschluss kann das Projekt mit `Build -> Build Solution` gebaut werden. Im Output ist nun erkennbar wo sich die gebaute DLL befindet. Neben dieser DLL müssen die DLL's von Protobuf und einer Abhängigkeit von Protobuf (System.Runtime.CompilerServices.Unsafe) heruntergeladen oder aus dem System entnommen werden. Dabei wird die Version 6.0.0 der Abhängigkeit benötigt.

Diese Abhängigkeiten werden im Client-Projekt von Into The Sky eingefügt. Der Zielordner ist `Assets/Plugins`. Dorthin werden die insgesamt drei DLL's Kopiert.

Der Proof-of-Concept wird durch das Erstellen eines neuen Scripts abgeschlossen. Dort kann nun der Serialiser benutzt werden. Auch die Klassen, die durch den Protobuf-Compiler erstellt wurden, sind verfügbar und können zur Serialisierung der Daten benutzt werden. Dies ist im aktuellen Projektstand in einer eigenen Szene "SerializationTestScene" im "Serialization-SanityChecker"-Skript testweise realisiert.

3.1.2 Tests

Um die Serialisierung zu testen, habe ich ein zweites Projekt der Solution hinzugefügt. Dieses wird als `MSTest Test Project` erstellt und bringt die nötigen Mittel mit, um eine Class Library zu testen. Nachdem die Class Library als Abhängigkeit eingetragen wurde kann man nun Unit-Test-Klassen schreiben. In diesem Fall habe ich folgende Testfälle initial angelegt, diese werden vermutlich im Laufe des Projekts erweitert. Die Tests werden nach einer stets gleichen Struktur benannt: Zuerst wird die Situation beschrieben, anschließend die Eingabe die getestet werden soll und zuletzt das zu erwartende Ergebnis notiert.

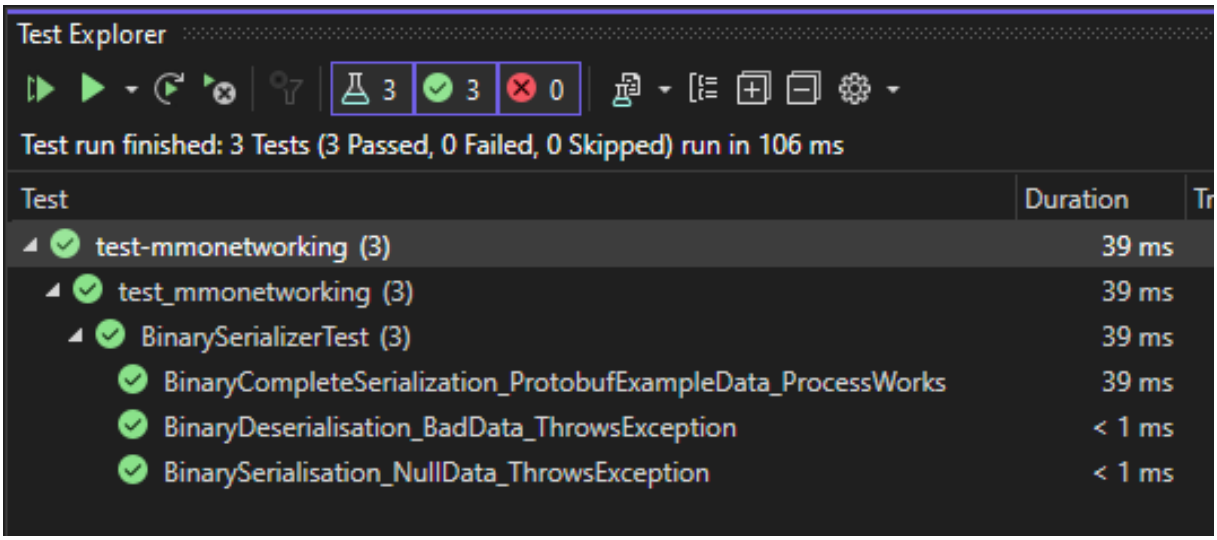


Abbildung 3.3: Beispielhafte Test-Cases (Into The Sky)

3.2 Park Chase

Für Park Chase habe ich mich für einen einfachen JSON-Serialiser entschieden. Übertragungsgeschwindigkeit und Latenz sind aufgrund des Spielkonzepts von Park Chase weniger kritisch. Außerdem wird das Spiel auf verschiedensten mobilen Endgeräten gespielt werden. Um dort vergleichsweise einfach das Spiel zu testen ist es wichtig, lesbare Formate zu benutzen. In diesem Fall habe ich mich für JSON aufgrund seiner einfachen Struktur und direkten Unterstützung durch Unity entschieden.

3.2.1 Entwicklung der Serialisierungs-Klasse

Unity enthält bereits eine Klasse, welche die JSON Serialisierung durchführt. Dabei handelt es sich um die Klasse `JsonUtility`. Diese eigene Klasse beinhaltet im wesentlichen nur Aufrufe der Methoden vom `JsonUtility` und dem Umwandeln des JSON-Strings in ein UTF-8 kodiertes Byte-Array.

3.2.2 Tests

Die Testfälle zu dieser Klasse sind ähnlich zu dem der Tests von Into The Sky aufgebaut. Hier werden allerdings die Test-Strukturen von Unity direkt verwendet.

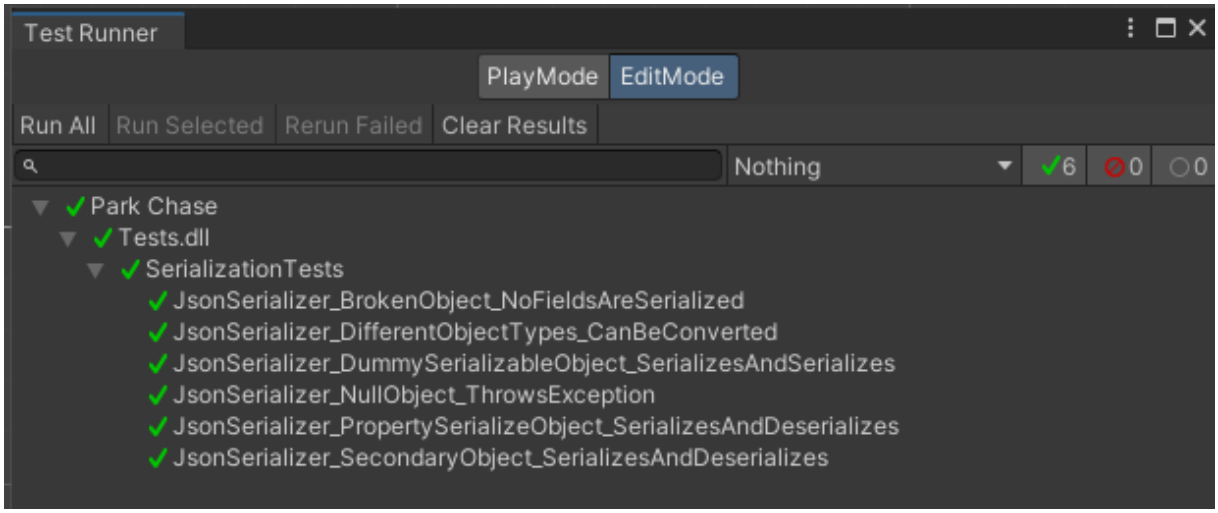


Abbildung 3.4: Beispielhafte Test-Cases (Park Chase)

Die Test-Strukturen erlauben es auch, Tests im laufenden Spielbetrieb laufen zu lassen. Dies ist für diese Klasse jedoch nicht sinnhaft, weshalb ein Unit-Test vollkommen ausreichend ist.

Literaturverzeichnis

[Dani 20] N. C. Daniel Persson Proos. *Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV*. Linköping University, 2020.

[Lore 17] B. P. M. Loren Kohnfelder, Elisa Heymann. *Introduction to Software Security*. University of Wisconsin-Madison, 2017.