Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

</farinuova>

Entwicklung eines HTML5 Internetspiels mithilfe von Cloud-Technologien und Messaging

von

Nicolas Mehlei

dem Fachbereich IV – Wirtschaftswissenschaften II –
der Hochschule für Technik und Wirtschaft Berlin vorgelegte Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)
im Studiengang
Angewandte Informatik

Berlin, 29. Januar 2013

Prüfungskommission

Vorsitzender:	Prof. Dr. Christian Herta	Hochschule für Technik und Wirtschaft Berlin
Gutachter:	Prof. Dr.-Ing. Hendrik Gärtner	Hochschule für Technik und Wirtschaft Berlin
	Prof. Dr. Christian Herta	Hochschule für Technik und Wirtschaft Berlin

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen bedanken, welche mich bei meiner Bachelorarbeit unterstützt haben.

Herrn Prof. Dr.-Ing. Hendrik Gärtner danke ich für die Übernahme der Betreuung, sowie der geduldigen Beantwortung unzähliger Fragen.

Meinem Bruder Janosch Mehlei gilt besonderer Dank für die stetige Ermutigung und für seine vielen Ratschläge.

Meinem Cousin Lucien Mehlei danke ich für seine Hilfe bei der Ideenfindung und -Ausarbeitung bezüglich des Spielkonzepts.

Dank gilt auch meinen Kollegen Constantin, Christian und Boris für die Entlastung am Arbeitsplatz. Ohne Euch hätte ich niemals so viel Zeit in diese Arbeit investieren können.

Des Weiteren danke ich den Designern rund um die *Neue Abteilung*, ohne deren Hilfe das User Interface von *CombatZone* "fürchterlich" ausgesehen hätte.

Nicht zuletzt möchte ich meiner Familie und Freunden für das leidige Korrekturlesen danken.

Nicolas Mehlei
Berlin, im Januar 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AJAX Asynchronous JavaScript and XML

BC Bounded Context

CQRS Command-Query Responsibility Segregation

DDD Domain-Driven Design

DTO Data Transfer Object

ES Event Sourcing

GUID Globally Unique Identifier

JSON JavaScript Object Notation

REST Representational State Transfer

RPC Remote Procedure Call

UI User Interface

1. Einleitung

1.1. Motivation

Die *FariNuova GmbH* – das Unternehmen in dessen Kooperation diese Arbeit entstanden ist – war zum Zeitpunkt der Anfertigung dieser Arbeit noch sehr unerfahren auf dem Gebiet der Entwicklung von Internetspielen. Konfrontiert mit den hohen Anforderungen an Skalierbarkeit und Flexibilität, welche moderne Internetspiele an die Infrastruktur stellen, ergab sich ein Interesse seitens der Firma nach einer Möglichkeit, diesen Anforderungen mittels begrenzter Entwickler- und monetären Ressourcen Herr zu werden.

Die Motivation des Autors hingegen ist begründet auf dem jahrelangen Interesse hinsichtlich der Entwicklung von mehrschichtigen Applikationsservern. Da diese jedoch oft in schwer zu skalierenden monolithischen Strukturen enden, bestand akademisches Interesse an der Findung einer Alternative, welche die Flexibilität und Ausfall-Toleranz von Cloud-Systemen mit der logischen Trennung von klassischen Schichten-Architekturen kombiniert.

Darüber hinaus gibt es bisher nur sehr wenig herausgebrachte Literatur bzgl. der in dieser Arbeit verwendeten Architektur. Diese Arbeit trägt einen Teil dazu bei, nachfolgenden Interessenten den Einstieg zu vereinfachen.

1.2. Ziel der Arbeit

Diese Arbeit verfolgt einen Ansatz, anhand von – zum Zeitpunkt des Verfassens dieser Arbeit – modernen Software-Architekturen wie Command-Query Responsibility Segregation (CQRS) und Cloud-geeigneten Kommunikationsarten wie Messaging ein skalierbares Internetspiel zu entwickeln, dessen Infrastruktur mit geringen Anpassungen auch eine hohe Spielerzahl unterstützen kann. Dieser Ansatz wird anhand einer prototypischen Entwicklung eines solchen Internetspiels gezeigt.

1.3. Gliederung der Arbeit

Zunächst werden im Kapitel *Grundlagen* theoretische Themen erläutert, welche von dieser Arbeit vorausgesetzt werden. Darauf folgt ein Überblick über das zugrundeliegende *Spielkonzept*.

Anhand dieses Spielkonzepts werden im Kapitel *Analyse* die Anforderungen erfasst, denen die Spiel-Software selbst, als auch deren Infrastruktur, gerecht werden muss. Diese Anforderungen werden dann im Kapitel *Entwurf* zu Modellen und Konzepten gewandelt, welche anschließend umgesetzt werden.

Während dieser Umsetzung erlangte Erkenntnisse werden im Kapitel *Implementierung* präsentiert. Daraufhin werden im Kapitel *Test* die Funktionen und Eigenschaften des realisierten Prototyps anhand der in Analyse aufgestellten Anforderungen verglichen. Die Resultate dieses Vergleichs werden dann im Kapitel *Ergebnisbewertung* erläutert. Darauffolgend wird im Kapitel *Zusammenfassung* ein Resümee über die vorangegangenen Kapitel und den erreichten Zustand gezogen.

Den Abschluss bildet ein *Ausblick*, wie das Ergebnis dieser Arbeit über den Kontext dieser Arbeit hinaus erweitert werden könnte.

2. Grundlagen

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen, welche zum Verständnis der in dieser Arbeit verwendeten Technologien und Praktiken vonnöten sind. Während einige davon bereits seit geraumer Zeit im Umlauf sind und in Zeiten von Cloud-Systemen einen "frischen Wind" erhalten, ist z.B. Command-Query Responsibility Segregation sehr neu und Erfahrungswerte sowie Literatur hierzu rar.

2.1. Messaging

Unter *Messaging* versteht man ein Integrationsschema, welches es erlaubt, mehrere autonome Applikationen oder Applikationskomponenten in entkoppelter Art und Weise miteinander zu verbinden. Messaging-basierte Kommunikation ist hierbei inhärent asynchron, was gerade in verteilten Systemen von Vorteil sein kann, da Sender und Empfänger nicht zwingend gleichzeitig für den Nachrichtenaustausch verbunden und funktionsfähig sein müssen. [9]

So lange sowohl Sender als auch Empfänger das Format der Nachricht verstehen, ist die hierfür verwendete Technologie irrelevant. Eine Nachricht kann also *Technologieagnostisch* sein. Dies ermöglicht es, zwei Systeme zu verbinden, welche in verschiedenen Programmiersprachen und für verschiedene Plattformen entwickelt wurden.

Über asynchrones Messaging verbundene Systeme sind von ihrer Geschwindigkeit entkoppelt. Da die Kommunikation nicht-blockierend ist, wird die Leistung des Senders nicht in Mitleidenschaft gezogen, wenn die Latenz der Verarbeitung des Empfängers – z.B. durch hohe Last – ansteigt.

Wenn die Entkoppelung der Systeme so weit gehen soll, dass der Sender der Nachricht den exakten Empfänger nicht wissen kann (oder soll), ist es möglich, einen Nachrichtenrouter einzusetzen. In einem solchen Aufbau muss der Sender nur den Router kennen, welcher wiederum die Nachricht an ihren korrekten Empfänger weiterleitet. [9]

Auf dem Übertragungsweg können Nachrichten transformiert werden, um sie in ein anderes Format umzuwandeln. Dies kann eingesetzt werden, um zwei zueinander inkompatible Systeme nachträglich mit einer Übersetzungsschicht zu verbinden. Diese Umwandlung kann ebenfalls durch einen Nachrichtenrouter implementiert werden. [8]

2.1.1. Messaging Patterns

Die Kommunikation zwischen Sender(n) und Empfänger(n) kann über eine Vielzahl verschiedener *Messaging Patterns* erfolgen. Zwei sehr wichtige Patterns sind *Warteschlangen* und *Publish-Subscribe*. [8]

Warteschlangen werden verwendet, wenn es nur einen konsumierenden Prozess gibt oder es keine Bewandtnis hat, welcher der konsumierenden Prozesse welche Nachricht verwendet. Entfernt ein konsumierender Prozess eine Nachricht von der geteilten Warteschlange, so verschwindet diese auch für alle weiteren darauf zugreifenden Konsumenten. [8]

Das *Publish-Subscribe* Messaging Pattern basiert auf dem "Observer Pattern" und ist dafür ausgelegt, Nachrichten an mehrere Empfänger zu senden. Ein oder mehrere Sender schicken hierbei

eine Nachricht an ein *Topic*. Die Empfänger abonnieren hingegen diese Topics und erhalten fortan für jede hieran gesendete Nachricht eine Kopie. [8]

2.1.2. Zustellungsgarantien

Nachrichten-Infrastrukturen unterstützen üblicherweise entweder eine *at-most-once* oder eine *at-least-once* Zustellungsgarantie.

Mittels einer *at-most-once*-basierten Zustellung kümmert sich die Infrastruktur bereits um die De-Duplizierung der Nachrichten. Dies kann jedoch dazu führen, dass bei einem Absturz oder Fehlverhalten der Nachrichtenverarbeitung Nachrichten verloren gehen, da die Infrastruktur keine Kenntnis darüber haben kann, ob die Nachrichtenverarbeitung erfolgreich war.

Für Systeme, bei denen Verlässlichkeit Priorität hat, ist daher meist eine *at-least-once*-basierte Zustellung besser geeignet. Diese setzt voraus, dass ein Empfänger nach einer erfolgreichen Verarbeitung Nachrichten explizit löscht. Geschieht dies nicht, weil beispielsweise ein Fehler während der Verarbeitung aufgetreten ist, so wird die Nachricht erneut zugestellt. Da es hierbei zu einer mehrfachen Bearbeitung einer Nachricht kommen kann, ist es wichtig, dass entweder die versendeten Nachrichten oder die gesamte Infrastruktur idempotent konzipiert wird. Dies wird im folgenden Abschnitt erläutert.

Während eine *exactly-once*-basierte Zustellung dem Namen nach als die bevorzugte Lösung erscheinen dürfte, ist die Realisierbarkeit einer solchen Zustellungsgarantie in den meisten Nachrichten-Infrastrukturen unpraktikabel, da dies eine Integration von Client-seitiger Logik voraussetzt. [5]

2.1.3. Idempotenz

Eine Operation wird als *idempotent* bezeichnet, wenn sie ohne Schaden oder Nebenwirkungen mehrfach ausgeführt werden kann. Eine Nachricht ist somit idempotent, wenn die Operation, mit welcher sie verknüpft ist, idempotent ist. [16]

Einige Operationen sind natürlich-idempotent. Ausschließlich lesende Operationen sind z.B. stets idempotent, da sich durch das simple Lesen eines Wertes dieser nicht ändern kann.

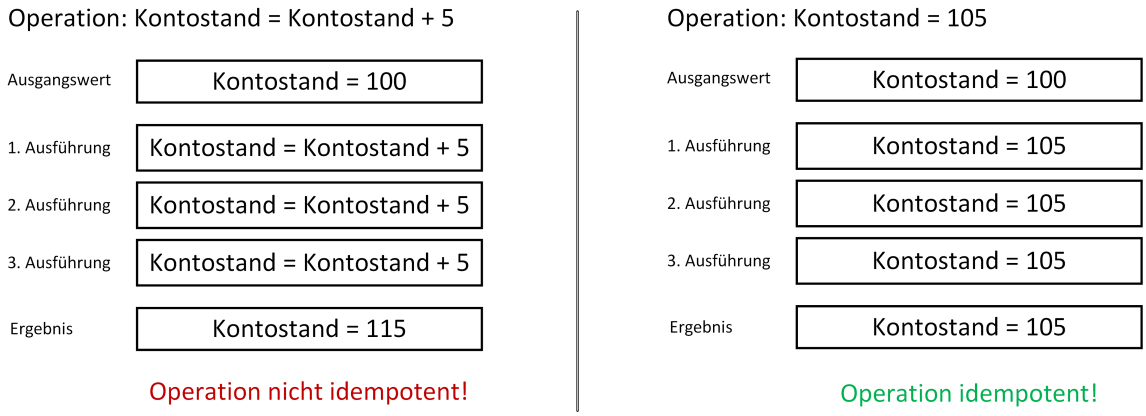


Abbildung 2.1.: Idempotenz-Beispiel

Schreibende Operationen können ebenfalls natürlich-idempotent sein. Eine Operation, welche einen spezifischen Wert setzt, ist idempotent, eine Operation die einen Wert inkrementiert jedoch nicht

(siehe hierzu Abbildung 2.1).

In nachrichtenbasierten Systemen kann auch die Infrastruktur dafür Sorge tragen, dass keine Operationen von nicht-idempotenten Nachrichten mehrfach ausgeführt werden. Dies kann z.B. durch Deduplizierung von Nachrichten geschehen, vorausgesetzt, die Infrastruktur weiß darüber Bescheid, welche Nachrichten bereits erfolgreich bearbeitet wurden. Ein Empfänger, welcher empfangene Nachrichten automatisch de-dupliziert, wird als *idempotenter Empfänger* bezeichnet. [9]

Wenn die Infrastruktur keine Idempotenz garantieren kann, können die Operationen von nicht-idempotenten Nachrichten in den meisten Fällen in äquivalente idempotente Operationen umgewandelt werden [16, S.56, S.374]. Im einfachsten Fall bedeutet dies an dem in Abbildung 2.1 sichtbaren Beispiel, dass durch die Operation kein Wert inkrementiert, sondern der gewünschte Wert gesetzt wird.

2.2. Domain-Driven Design

Domain-Driven Design (DDD) wurde erstmals von Eric Evans in seinem Werk "Domain-Driven Design: Tackling Complexity in the Heart of Software" benannt. DDD beschreibt eine Herangehensweise zur Entwicklung von Software, deren Kern sich extrem nah an die Anforderungen der Business-Domäne anlehnt. Um solch eine Symbiose zu erreichen, ist eine sehr enge Zusammenarbeit und vor allem sehr tiefgehende Kooperation sowie Kommunikation zwischen Entwicklern und Domänen-Experten notwendig. [6, 17]

Ermöglicht wird dies unter anderem aufgrund von zwei wichtigen Elementen: die *Ubiquitous Language* sowie spezifischer *Software-Bausteine (Building Blocks)*.

2.2.1. Ubiquitous Language

Als *Ubiquitous Language* wird die gemeinsame Sprache bezeichnet, welche sich bei korrekter Anwendung der Vorgehensmodelle von Domain-Driven Design zwischen den Team-Mitgliedern entwickelt. Diese Sprache soll die Kommunikation vereinfachen und dafür sorgen, dass trotz der unterschiedlichen Kenntnisbereiche zwischen den Team-Migliedern keine Missverständnisse auftreten. Ein wichtiger Aspekt hierbei ist die Entwicklung eines gemeinsamen Vokabulars, welches unabhängig von technischen Fachbegriffen und Konzepten der Entwickler sowie des Business-Jargons der Domänen-Experten aufgebaut ist. [6, 17]

2.2.2. Software-Bausteine (Building Blocks)

Um die Ideen und Konzepte von DDD effektiv in eine Software einfließen lassen zu können – und die Domänen-Logik hierbei prägnant zu belassen –, bedarf es einiger neuer Praktiken und Konzepte. Eine Schematik dieser *Building Blocks* genannten Bausteine ist in Abbildung 2.2 zu sehen. [6]

In den folgenden Abschnitten werden diese nun nacheinander näher erläutert.

Entities & Value Types

Als *Entities* werden Domänen-Konzepte bezeichnet, welche eindeutig innerhalb des Systems identifiziert werden müssen. Diese sind daran zu erkennen, dass selbst nach Veränderung aller ihrer Werte Entities trotzdem dasselbe Objekt im Kontext des Unternehmens verkörpern. [6, 17]

Im Gegensatz zu Entities ist bei *Value Types* keine Identität erlaubt. Sie werden unveränderlich implementiert, enthalten in den meisten Fällen sehr viel weniger Domänenlogik als Entities und werden ausschließlich durch ihren eigenen Wert identifiziert. [6, 17]

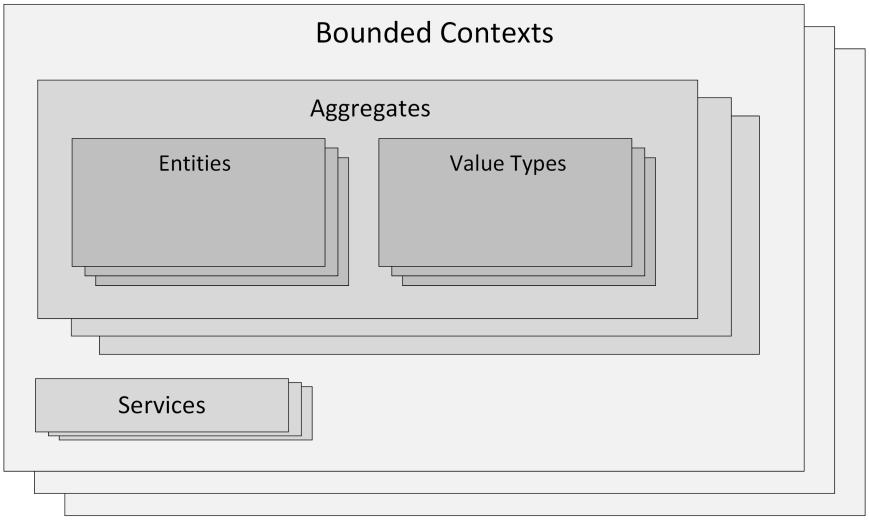


Abbildung 2.2.: Hierarchie der DDD Building Blocks

Aggregates

Aggregates bündeln Entities und Value Types in Konsistenz-Einheiten. Jede Operation, welche innerhalb eines Aggregates ausgeführt wird, muss zu einem konsistenten Zustand führen. Dadurch wird ein Aggregate automatisch eine Transaktionseinheit. [6]

Ein Aggregate darf andere Aggregates via deren Identität lose referenzieren, darf jedoch keine feste (und somit navigierbare) Referenz halten. Eine Referenz – sowohl lose als auch fest – auf die Identität einer Entity von außerhalb des jeweiligen Aggregates ist nicht erlaubt, ausschließlich auf den Aggregate selbst. [6]

Alle Elemente innerhalb eines Aggregates teilen sich dessen Lebenszeit. Wird ein Aggregate gelöscht, müssen auch alle Entities und Value Types, die sich darin befinden, gelöscht werden. [6, 17]

Services

Domänen-Logik, welche keinem Aggregate direkt zugeordnet werden kann – oder Aggregate-übergreifend fungiert – , kann innerhalb von so genannten *Services* implementiert werden. Da der Begriff “Service“ innerhalb des IT-Vokabulars eine Vielzahl an Bedeutungen haben kann, werden diese im Kontext von Domain-Driven Design auch gerne *Domain Services* genannt. [17]

2.2.3. Bounded Contexts

Je komplexer eine Business-Domäne wird, desto unübersichtlicher wird oftmals dessen Implementierung und umso häufiger tritt eine Mehrdeutigkeit der Business-Konzepte auf. Um dem entgegenzuwirken, gibt es *Bounded Contexts*, oft abgekürzt als *BC*. BCs unterteilen den Problemraum der Domäne in Bereiche ein, die zwar untereinander verknüpft sein können, aber jeweils einen anderen Aspekt der Gesamt-Domäne abbilden. Beispielsweise kann ein E-Commerce System die BCs “Katalog“ und “Buchhaltung“ besitzen. [17]

Die genauen Konzepte, nach denen die Trennung von BCs erfolgen kann, sind zu komplex für den Rahmen dieser Arbeit. Hierzu wird auf die weiterführende Literatur [6] und [17] verwiesen.

2.3. CAP-Theorem

Das CAP-Theorem beschreibt drei Anforderungen, welche üblicherweise an ein verteiltes System gestellt werden:

- Consistency (Konsistenz)
- Availability (Verfügbarkeit)
- Partition Tolerance (Partitionstoleranz)

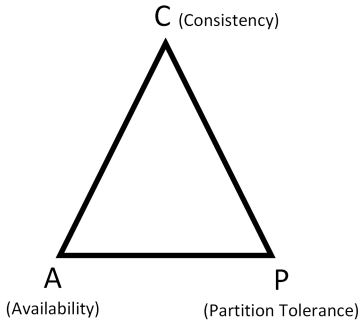


Abbildung 2.3.: Das CAP-Theorem

Konsistenz in einem verteilten System bedeutet, dass alle Knoten des Systems auf denselben Daten agieren. Dies garantiert, dass auf eine an das System gerichtete Anfrage dieselbe Antwort folgt, gleichgültig an welchen Knoten diese gestellt wird.

Verfügbarkeit ist gegeben, wenn die Knoten des verteilten Systems in der Lage sind, Anfragen zu beantworten.

Partitionstoleranz bedeutet, dass das verteilte System weiterhin korrekt funktioniert, auch wenn die Kommunikation zwischen den Knoten unterbrochen ist.

Nach dem Theorem kann ein verteiltes System stets nur zwei dieser Anforderungen erfüllen. [5]

2.4. Eventual Consistency

Eventual Consistency ist ein Konsistenzmodell, welches insbesondere in Cloud-Szenarien gerne verwendet wird. Es besagt, dass, wenn über einen längeren Zeitraum keine Akualisierungen an einem Datenbestand erfolgen, alle Replikate nach und nach konsistent werden. Dies steht im Kontrast zum Konsistenzmodell “Immediate Consistency“, bei dem sich jedes Replikat zu jedem Zeitpunkt in einem konsistenten Zustand befinden muss, weswegen der Grad der Konsistenz von *Eventual Consistency* als schwächer bezeichnet wird. [4, 16]

Durch die Lockerung der Konsistenz-Anforderungen kann eine bessere Performanz – insbesondere in verteilten Systemen – erreicht werden. Nach dem CAP-Theorem können sich auch weitere Vorteile ergeben, welche bei “Immediate Consistency“ nicht möglich sind. [4]

Insbesondere für Anwendungsgebiete, in denen es weit mehr lesende als schreibende Zugriffe gibt, oder es nur einen (oder zumindest sehr wenige) schreibende Prozesse existieren, kann dieses Konsistenzmodell seine Vorteile ausspielen. Ein prominentes Beispiel für die Anwendung von Eventual Consistency ist das Domain Name System (DNS). [16]

2.4.1. Daten-Aktualität

Applikationen, welche auf Daten aus einer *Eventual Consistent*-Datenquelle zugreifen, müssen so konzipiert werden, dass potenziell veraltete Informationen zurückgegeben werden können.

Hierfür gibt es verschiedene Varianten damit umzugehen, z.B. den Benutzer einfach darauf hinzuweisen oder mittels Caching-Techniken dies vor dem Benutzer zu verbergen. In einigen Anwendungsgebieten ist es der Benutzer sogar bereits gewohnt, veraltete Informationen präsentiert zu bekommen. Diese Problematik wird in späteren Kapiteln detaillierter ausgeführt.

2.5. Event Sourcing

Das Pattern *Event Sourcing* beschreibt eine Persistenz-Technik, mit der es möglich ist, alle Änderungen an einem System zu speichern, ohne vorherige Werte zu überschreiben.

Dies setzt voraus, dass System-Änderungen serialisiert werden können. Durch Einsatz von DDD können diese gespeicherten Änderungen, *Events* genannt, die exakte Business-Relevanz der Änderungen abbilden und erhalten somit den Grund für die Änderung bei. Die Menge der Events gibt somit alle Schritte an, die das System seit seiner Inbetriebnahme durchlaufen hat um zu seinem aktuellen Stand zu kommen. Dies erleichtert einerseits ein späteres Überwachen und Nachverfolgen von Informationen (z.B. aus Diagnose- oder Debugging-Gründen), da der Weg, den ein aktueller Wert durchlaufen hat, in jedem Zwischenschritt betrachtet werden kann. Andererseits können hiermit oftmals Informationen extrahiert werden, welche zum Entwicklungszeitpunkt noch nicht als wichtig angesehen wurden, wie z.B. zur Erstellung von Statistiken anhand des Verlaufs von Werten. [17]

Wenn Event Sourcing in Kombination mit DDD verwendet wird, werden die gespeicherten Events oftmals anhand der Aggregates, zu denen sie gehören, eingeteilt. Dadurch ergeben sich Ketten von zusammengehörigen Events, welche *Event Streams* genannt werden. Ein fiktives Beispiel eines Event-Streams für einen Customer-Aggregate ist in Abbildung 2.4 dargestellt.

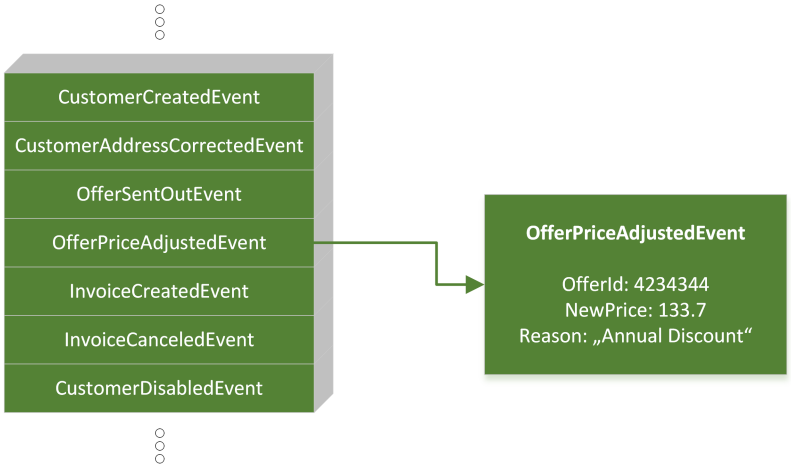


Abbildung 2.4.: Event Stream Beispiel

Da Events abbilden sollen, was dem System seit seiner Inbetriebnahme passiert ist, sind sie nach ihrer Speicherung üblicherweise unveränderlich, da sich die Vergangenheit nicht verändern kann. Änderungen sind daher ausschließlich durch das Hinzufügen weiterer Events möglich, beispielsweise zur Korrektur nach einem Systemfehler. Dies kann jedoch, bei richtiger Verwendung, von Vorteil sein, da ein Fehlverhalten des Systems Auswirkungen auf die Umwelt des Betriebs haben

kann. Beispielsweise könnten durch ein hinzugefügtes Korrektur-Event, welches den inkorrekten Wert anpasst, automatisch Hinweis- oder Entschuldings-E-Mails an die Kunden verschickt sowie verknüpfte Systeme über die Änderung notifiziert werden. [17]

2.5.1. Event Store

Als *Event Store* wird die Komponente bezeichnet, welche für die Speicherung und das Abrufen von zuvor gespeicherten Events zuständig ist. Da Events im Normalfall nicht löschbar sind, kann die Persistierung durch das sequenzielle Anhängen an den bisherigen Datenbestand durchgeführt werden. Aus diesem Grund wird der hierfür verwendete Speicher von manchen Entwicklern auch als Tape-Speicher betitelt, angelehnt an (ebenfalls ausschließlich sequenziell beschreibbare) Magnetbänder.

Wie die Events durch den Event Store gespeichert werden, ist nicht vorgegeben. Dies kann anhand der Performanz-Ansprüche und der vorhandenen Infrastruktur entschieden werden. Beliebt sind hier sowohl SQL-, NoSQL- als auch BLOB-basierte Datenspeicher.

In vielen Implementationen obliegt es dem Event Store dafür Sorge zu tragen, dass neue Events an daran interessierten Parteien (Abonnenten) weitergeleitet werden. Dies geschieht üblicherweise über eine Publish-/Subscribe-basierte Nachrichten-Infrastruktur. [17]

2.6. Command-Query Responsibility Segregation (CQRS)

Command-Query Responsibility Segregation (CQRS) ist eine Architektur, welche auf dem Designprinzip Command-Query Seperation (CQS) aufbaut. Der Begriff CQS stammt von Bertrand Meyer aus seinem Buch “Object-Oriented Software Construction“ [11]. [17]

Nach CQS sollten alle Methoden einer Klasse in zwei Kategorien geteilt werden, Queries (Abfragen) und Commands (Kommandos). *Queries* geben hierbei einen Wert zurück, verändern jedoch nicht das System. *Commands* hingegen ändern einen Wert des Systems, geben jedoch nichts zurück.

2.6.1. Architektur

Während CQS die Unterteilung auf Objekt-Ebene vorsah, geht CQRS einen Schritt weiter und bricht diese Gebiete auf architektonischer Ebene auseinander. Die vormals kombinierte Anwendung wird dabei aufgeteilt in ein *Schreib-Segment* und ein *Lese-Segment*. Durch diese Auftrennung kann die Persistenz-Schicht der beiden Segmente besser an die jeweiligen Anforderungen angepasst werden und die Segmente können unterschiedlich skaliert werden. Insbesondere die Möglichkeit der unterschiedlichen Persistenz-Arten kann sehr wertvoll sein, da Lese-Zugriffe grundlegend andere Anforderungen an Technologie und Schema des Datenspeichers stellen als Schreib-Zugriffe. [17]

Dies ermöglicht es beispielsweise für alle Lese-Zugriffe eine Datenbank zu verwenden, welche denormalisiert und mit gelockerten Konsistenz-Ansprüchen (z.B. via Eventual Consistency) konzipiert wurde, was zu einer besseren Performanz führen kann.

Beliebt ist es hier z.B. bereits alle Daten exakt so zu denormalisieren, dass sie von dem späteren User Interface (UI) ohne große Umwandlung konsumiert werden können.

Bei Verwendung von Event Sourcing als Persistenz-Methode des Schreib-Segments kann die Lese-Datenbank auch zu einem späteren Zeitpunkt vollständig neu erstellt werden. Die hierfür notwendigen Verfahren werden im Abschnitt 2.6.4 genauer erläutert. Durch die dadurch ermöglichte Flüchtigkeit der Lese-Datenbanken ergeben sich ganz neue Möglichkeiten: z.B. kann bei einem

neuen Release die alte Lese-Datenbank einfach verworfen und anhand der Datenstruktur der neuen Version neu erstellt werden. Dies kann theoretisch komplexe und zeitaufwändige Daten- und Schema-Migrationen ersparen.

Eine Skizze, welche den Aufbau einer möglichen CQRS-Architektur unter Verwendung von Event Sourcing zeigt, ist in Abbildung 2.5 zu sehen.

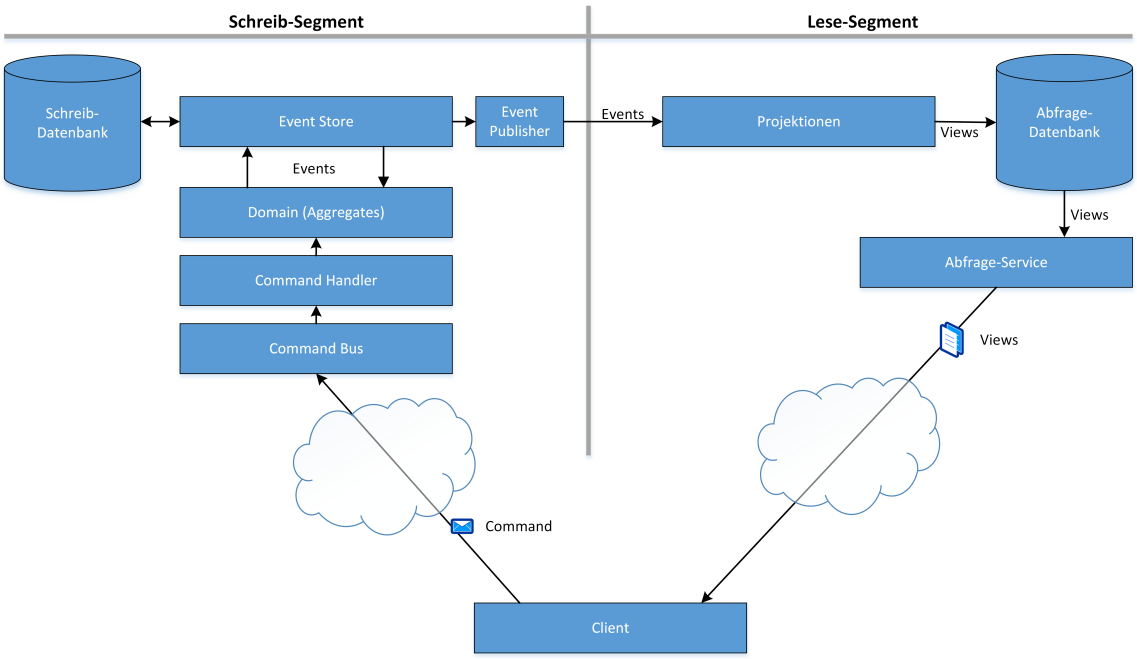


Abbildung 2.5.: Skizze einer typischen CQRS-Architektur

2.6.2. Commands

Ein *Command* ist eine Operation, welche in ein Data Transfer Object (DTO) serialisiert wurde. Die in einem System zur Verfügung stehenden Command-Typen sind durch die Business-Domäne (bzw. bei Verwendung von DDD durch die Ubiquitous Language) vorgegeben und verkörpern das Vorhaben, welchen der User beim Ausführen seiner Aktion vorhatte.

Während seiner Bearbeitung kann ein Command abgewiesen werden, beispielsweise durch das Fehlschlagen einer Validierung. Es ist daher sinnvoll, bereits früh (z.B. Client-seitig) das Command auf dessen Korrektheit zu überprüfen, um die Wahrscheinlichkeit für abgewiesene Commands so gering wie möglich zu halten, da diese trotzdem Server-Ressourcen verbrauchen und potenziell die User Experience beeinträchtigen. Letzteres kommt daher, da bei einem abgewiesenen Command entweder manueller Aufwand erforderlich wäre oder das Command verworfen werden müsste.

Ein Task-basiertes (oder nach Microsoft, Induktives) User Interface (siehe [18, S. 9-16] und [12]) erleichtert die Erzeugung von Kommandos, da der derzeitige Anwendungsfall (Use-Case) – und somit der entsprechende Command-Typ – direkt erkennbar ist. Aus einem Daten-Grid ist es beispielsweise oft nur sehr schwer erkennbar, aus welchem Grund der User einen Wert ändern möchte.

Command Handler

Der *Command Handler* ist die Komponente, welche eingehende Commands entgegennimmt und dafür sorgt, dass die damit verbundene Domain-Aktion ausgeführt wird. Command Handler sind

somit die einzige Schnittstelle zur Domänenlogik.

Der typische Ablauf unter Verwendung von Event Sourcing und Domain-Driven Design lautet:

- Neues Command entgegennehmen
- Command validieren
- Betreffenden Aggregate laden
- Entsprechende Operation vom Aggregate ausführen
- Neue Events an Event Store übergeben

Im Namensschema von DDD gelten Command Handler als Services. Wenn die Applikation viele Command-Typen besitzt, kann es sinnvoll sein, alle Command Handler anhand ihrer Aggregate-Typen gruppiert in Klassen zusammenzufassen, um die Übersichtlichkeit zu verbessern. [17, 1]

Command Bus

Als *Command Bus* wird der Teil der Infrastruktur bezeichnet, welcher für den Transfer der Commands zwischen den Sendern und den Command Handlern zuständig ist. Er gilt generell als der einzige Eintrittspunkt in die Domänenlogik. Der Command Bus kann entweder synchron oder asynchron konzipiert sein.

Ein synchroner Command Bus könnte beispielsweise ein RPC-basierter Webservice sein, welcher das Command direkt an den Command Handler gibt und optional das Ergebnis direkt zurückliefert.

Ein asynchroner Command Bus könnte entweder ebenfalls RPC-basiert sein, jedoch z.B. nichts oder nur das Validierungsergebnis des Commands zurückliefern. Wenn die Validation bereits auf Sender-Seite erfolgt, kann auf Command Bus Ebene theoretisch darauf verzichtet werden. Alternativ kann der Sender auch direkt auf eine Messaging-Infrastruktur zugreifen, um das Command abzusenden, dies wäre dann komplett asynchron.

Hybrid-Systeme sind hierbei auch möglich, welche z.B. je nach Command-Typ entweder synchron oder asynchron mit dem Command verfahren.

2.6.3. Events

Events werden für die Speicherung von Änderungen, welche dem System widerfahren sind, verwendet. Die Definition deckt sich hierbei mit den Events, die bereits im Rahmen von Event Sourcing in Abschnitt 2.5 präsentiert worden sind.

Unabhängig davon, ob Event Sourcing verwendet wird, können Events für die Kommunikation mit Komponenten innerhalb und außerhalb des Systems genutzt werden. Durch das Empfangen der veröffentlichten Events können diese auf Änderungen des sendenden Systems reagieren.

2.6.4. Event Handler

So wie Command Handler für die Bearbeitung von Commands zuständig sind, so sind *Event Handler* für die Reaktion auf Events konzipiert. Dies kann aus unterschiedlichen Gründen notwendig sein, z.B. für die Aktualisierung der Lese-Datenbank (gemeinhin Projektionen genannt) oder für das Anstoßen von mit dem Event verknüpften Prozessen.

Projektionen

Projektionen erlauben es, mithilfe der durch die Behandlung von Commands erzeugten Events eine Abfrage-Datenbank zu aktualisieren. Da solche Abfrage-Datenbanken zur Leistungssteigerung meist de-normalisiert sind, nennt man Projektionen auch “Denormalisierer“ (Denormalizer).

Basierend auf den Anforderungen an Bearbeitungslatenz und Konsistenz kann die Ausführung der Projektionen entweder synchron (innerhalb der Transaktion des Command Handlers) oder asynchron erfolgen. Es ist hierbei nicht vorgegeben, ob die Projektionen innerhalb desselben Prozesses erzeugt werden oder ob dies in einem eigenen Prozess – möglicherweise sogar auf einer weiteren Maschine – ausgelagert wird.

Je nach Aufteilung in dedizierte Prozesse lassen sich die Projektionen unabhängig von der restlichen Infrastruktur skalieren. Dies ist sinnvoll, wenn mehrere Abfrage-Datenbanken erstellt werden sollen, z.B. um die anfallende Last zu verteilen oder um die Daten in mehreren global-verteilten Rechenzentren zu speichern. Jeder Aktualisierungsprozess würde in solch einem Fall eine Kopie der Events erhalten und würde jeweils seine zugewiesene Abfrage-Datenbank mit den Änderungen versorgen. Eine beispielhafte Darstellung hierfür ist in Abbildung 2.6 zu sehen.

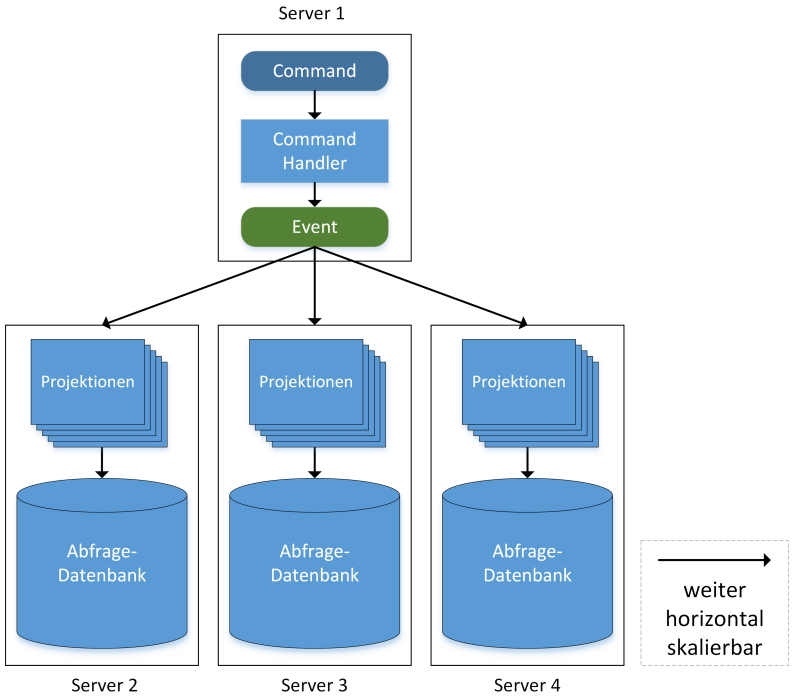


Abbildung 2.6.: Horizontale Skalierung von Projektionen

Prozesse

Event Handler können auch genutzt werden, um langläufige Business-*Prozesse* abzubilden. Hierzu gehen innerhalb der CQRS-Community die Meinungen stark auseinander, sowohl wie solche Event Handler genannt werden als auch wie eine Implementation aussehen könnte. Oft verwendete Bezeichnungen sind entweder “Prozess“, “Workflow“ oder “Saga“. Ebenfalls umstritten ist, ob diese Workflows ausschließlich Commands versenden oder direkten Zugriff auf die Domäne besitzen dürfen, z.B. durch Ausführen von Aggregate-Operationen oder das direkte Erzeugen von Events.

Innerhalb dieser Arbeit wird der Begriff “Workflow“ verwendet. Des Weiteren werden Workflows

zustandslos implementiert und dienen dem Routing zwischen Aggregates sowie zwischen Bounded Contexts.

2.6.5. BC-Kommunikation

Kommunikation zwischen Bounded Contexts kann auf verschiedene Weise erfolgen. Diese Wahl hängt meist von der benötigten Entkoppelung, den verfügbaren Schnittstellen, den vorherrschenden Datenschutz-Richtlinien und der verwendeten Technologie ab.

BC1 und BC2 sind im folgenden zwei verschiedene Bounded Contexts, welche zur Veranschaulichung verwendet werden.

Darf Domänen-Logik zwischen den BCs geteilt werden, so kann BC2 zum Beispiel die Events von BC1 abonnieren und entsprechend darauf reagieren. Dies bindet BC2 jedoch an die derzeitige Implementation von BC1. Sollte sich das Schema der Events von BC1 ändern, so müsste auch BC2 angepasst werden.

Eine Alternative ist die Implementation eines Webservices seitens BC1, auf den BC2 zugreifen kann. Dies entkoppelt BC2 von der Event-Implementation von BC1, denn die Entwickler von BC1 könnten bei einem Upgrade den Webservice ebenfalls anpassen und so die Kompatibilität beibehalten.

Für eine detailliertere Erklärung zur Kommunikation zwischen Bounded Contexts wird auf die Quellen [6] und [17] verwiesen.

3. Spielkonzept

Bevor die Anforderungen an das zu entwickelnde Spiel erfasst werden können, muss zunächst das zugrundeliegende Spielkonzept erörtert werden.

CombatZone ist ein Mehrspieler-Echtzeit-Strategiespiel, angesiedelt im Science-Fiction Genre. Es dreht sich primär um die Erweiterung des eigenen Gebiets, dem Kämpfen um Sektoren sowie dem taktischen Vorgehen mittels der eigenen Raumschiff-Flotten.

Ausgetragen wird das Spiel auf zweidimensionalen Spielfeldern, den *Universen*. Ein Universum hat eine anfängliche Größe von 50x50 (= 2500) Feldern, auch *Sektoren* genannt. Diese Größe kann jedoch im Verlauf des Spiels bis zu einer maximalen Größe von 250x250 (= 62.500) Sektoren anwachsen.

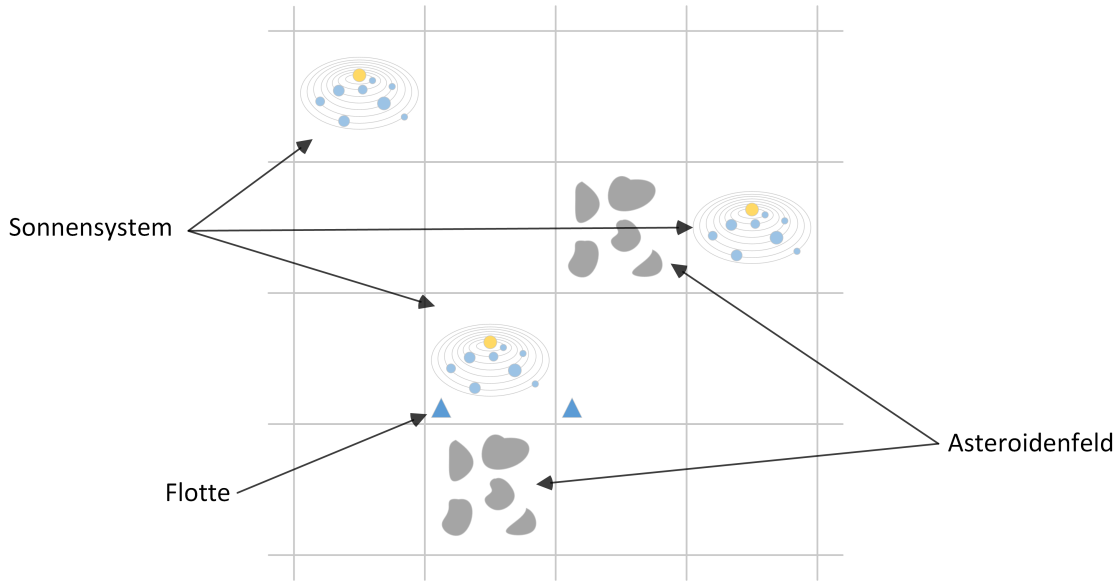


Abbildung 3.1.: Skizze eines Universumsausschnitts

Die verschiedenen Universen sind unabhängig voneinander und können als Instanzen des Kernspiels angesehen werden. Sie sind zeitlich begrenzt und es kann zu jedem Zeitpunkt mehrere parallele Universen geben. Die Anzahl und Dauer der Universen wird durch den Spielleiter/Administrator festgelegt und kann an den Spielkonsum, also die Menge und das Verhalten der Spieler, angepasst werden.

Die Standarddauer eines Universums beträgt 48 Stunden, wobei alle 24 Stunden ein neues Universum gestartet werden soll.

3.1. Sektoren

Die Sektoren eines Universums werden bei dessen Erstellung verteilt. Jeder Sektor kann von einem unterschiedlichen Typ sein. Zur Auswahl stehen:

- Leerer Sektor *(75 % Wahrscheinlichkeit)*
- Sonnensystem *(6,25 % Wahrscheinlichkeit)*
- Wurmloch *(6,25 % Wahrscheinlichkeit)*
- Nebel *(6,25 % Wahrscheinlichkeit)*
- Asteroidenfeld *(6,25 % Wahrscheinlichkeit)*

In der dieser Arbeit beiliegenden Version unterscheiden sich die Sektortypen “Leerer Sektor“, “Wurmloch“, “Nebel“ sowie “Asteroidenfeld“ nur in der Ästhetik; spielerisch gibt es keinen Unterschied, da der Fokus dieser Arbeit auf der Infrastruktur liegen soll. Eine spätere Integration in die Spiellogik könnte so aussehen, dass ein Nebel dem Verteidiger bzw. ein Asteroidenfeld dem Angreifer einen Kampfbonus gibt.

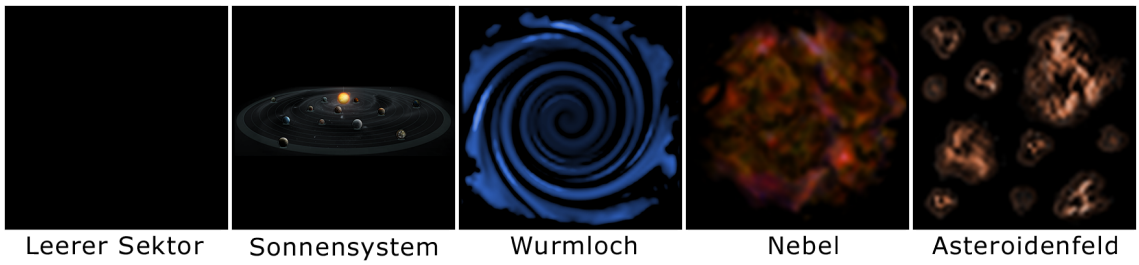


Abbildung 3.2.: Übersicht über die fünf Sektortypen

Dargestellt werden die Sektoren anhand von rechteckigen Feldern. Diese sind in Aussehen und Farbe entsprechend differenziert entworfen worden, um auch bei der gleichzeitigen Anzeige von hunderten Sektoren dem Spieler eine schnelle Unterscheidung zwischen den verschiedenen Sektortypen zu ermöglichen. Die verwendeten fünf Grafiken für die Sektor-Darstellung sind in Abbildung 3.2 ersichtlich.

3.2. Raumschiffstypen

Um eine, dem vergleichsweise einfachen Spielprinzip entsprechende, taktische Vielfalt zu bieten, soll es fünf verschiedene Raumschiffstypen geben. Der Spieler kann für jedes der von ihm kontrollierten Sonnensysteme entscheiden, welcher Raumschiffstyp in diesem Sektor produziert werden soll.

Abbildung 3.3.: Übersicht über die fünf Raumschiffstypen

Die ersten vier Raumschiffstypen sind hierbei Kampfschiffe, wohingegen der verbliebende Typ – der Kundschafter – eine Sonderrolle einnimmt. Er ist der einzige Schiffstyp, welcher Sektoren

übernehmen kann, und ist somit essenziell für die Vergrößerung des eigenen Einflussbereiches. Durch den Mangel an Bewaffnung, die (vergleichsweise) lange Bauzeit und der langsamen Fortbewegung muss besonderer Wert auf den Schutz der eigenen Kundschafter gelegt werden.

Das Aussehen sowie die Werte der Schiffstypen sind Abbildung 3.3 und Tabelle 3.1 zu entnehmen.

	Korvette	Fregatte	Zerstörer	Schlachtkreuzer	Kundschafter
Rolle	Kampf	Kampf	Kampf	Kampf	Sektorübernahme
Fluggeschwindigkeit	1	2	3	4	5
Baugeschwindigkeit	1	3	5	7	10
Kampfstärke	100	400	850	1400	0
Vergleichswert	10500	14000	17850	21000	-

Tabelle 3.1.: Werte der Raumschiffstypen

3.3. Spieler

Nach seiner Registrierung kann der Spieler einem – oder mehreren – derzeit laufenden Universen beitreten. Daraufhin wird dem Spieler ein Startsektor zugeteilt.

Vom Startsektor aus kann der Spieler Schiffe produzieren und benachbarte Sektoren übernehmen. Dies erweitert nicht nur seinen Einflussbereich, sondern erlaubt auch eine größere Flugreichweite seiner Raumschiffe und, wenn er weitere Sektoren vom Typ “Sonnensystem“ einnimmt, die parallele Produktion mehrerer Raumschiffe.

Das Ziel des Spiels ist, am Ende der Laufzeit eines Universums die höchste Punktzahl zu besitzen. Die Punktzahl eines Spielers ergibt sich aus der Anzahl der Sektoren in seinem Besitz, wobei Sektoren vom Typ Sonnensystem und von Gegnern übernommene Sektoren höher gewertet werden.

3.4. Zielgruppe

Durch sehr schnelllebige Runden ist *CombatZone* insbesondere für Menschen geeignet, welche ausreichend Freizeit zur Verfügung haben, um einen großen Teil der Runde aktiv mitzuerleben. Ein längeres Fernbleiben würde schnell dazu führen, dass der Spieler zu stark zurückfällt oder durch den Verlust des letzten eigenen Sektors ganz aus der Runde ausgeschlossen wird.

Durch diese Schnelllebigkeit steht *CombatZone* im starken Kontrast zu vielen Konkurrenzspielen, welche auf lange Spielrunden ausgelegt sind. Des Weiteren werden üblicherweise Spielelemente integriert, welche es vermeiden, dass ein Spieler komplett aus dem aktiven Spielgeschehen herausgenommen werden kann, z.B. indem dessen letzte Basis nicht angegriffen werden kann oder ihm nach Zerstörung des letzten Stützpunkts automatisch ein Ersatz generiert wird. Solch Funktionalität wurde absichtlich nicht integriert, da durch die kurzen (und parallelen) Universen der Spieler stets die Möglichkeit hat, einem weiteren Universum beizutreten.

4. Analyse

In diesem Kapitel sollen die Anforderungen der im Rahmen dieser Arbeit zu entwickelnden Spiele-Applikation *CombatZone* analysiert werden. Hierzu werden anhand des im vorigen Kapitel erläuterten Spielkonzepts die funktionalen und nicht-funktionalen Anforderungen abgeleitet.

4.1. Funktionale Anforderungen

In den folgenden Anforderungen wird absichtlich zwischen Benutzer und Spieler unterschieden. Ein Benutzer ist hierbei der Mensch, der sich entweder für das Spiel registrieren möchte oder bereits registriert ist. Ein Spieler hingegen ist ein Benutzer, welcher bereits einem (spezifischen) Universum beigetreten ist. Ein Benutzer kann somit mehrere Spieler verkörpern. Dies wird in Kapitel 5 genauer ausgeführt.

4.1.1. Benutzer

- Ein Benutzer kann sich registrieren.
- Ein Benutzer muss einen Benutzernamen, eine E-Mail-Adresse sowie ein Passwort für seine Registration angeben.
- Die vom Benutzer bei der Registration angegebene E-Mail-Adresse darf sich nicht bereits im System befinden.
- Der Benutzer kann sich mit seinen Zugangsdaten anmelden.
- Ein Benutzer kann in beliebig vielen aktiven Universen gleichzeitig Spieler sein.
- Benutzer können sich innerhalb der Spieloberfläche gegenseitig Nachrichten schreiben.

4.1.2. Universum

- Die Universumskarte ist quadratisch.
- Die Universumskarte besitzt standardmäßig 50x50 Sektoren.
- Die Universumskarte kann bei entsprechender Auslastung bis zu einer Maximalgröße von 250x250 Sektoren anwachsen.
- Die Universumskarte kann vollständig und übersichtlich vom Spieler betrachtet werden.
- Ein Universum ist bis zu einem festgelegten Enddatum aktiv.

4.1.3. Raumschiffe

- Ein Raumschiff kann eines von 5 Typen sein: Korvette, Fregatte, Zerstörer, Schlachtkreuzer oder Kundschafter.
- Raumschiffe werden von Sonnensystem-Sektoren produziert.
- Raumschiffe unterschiedlichen Typs unterscheiden sich in Fluggeschwindigkeit, Baugeschwindigkeit sowie Kampfstärke.

- Raumschiffe können zwischen Sektoren bewegt werden.
- Das Verschicken von Raumschiffen ist nur möglich, wenn der Spieler Kontrolle über einen an den Zielsektor angrenzenden Sektor besitzt.
- Raumschiffsschlachten werden sofort entschieden.

4.1.4. Sektor

- Ein Sektor besitzt eine im jeweiligen Universum eindeutige X-Y Koordinate.
- Der Typ eines Sektors ist zufällig. Möglich sind: Leerer Raum, Sonnensystem, Asteroidenfeld, Nebel oder Wurmloch.
- Der Startsektor eines Spielers ist stets vom Typ Sonnensystem.
- Nur Sektoren vom Typ Sonnensystem produzieren Raumschiffe.
- Sonnensystem-Sektoren können stets nur ein Raumschiff von einem Typ gleichzeitig produzieren.
- Die Übernahme eines Sektors erfordert mindestens einen Kundschafter im selben Sektor.
- Die Übernahme eines Sektors dauert 10 Minuten.
- Während der Übernahme eines Sektors muss mindestens ein Kundschafter im Sektor bleiben.

4.1.5. Spieler

- Ein Spieler sammelt Punkte durch die Übernahme von Sektoren.
- Ein Spieler kann über eine Rangliste die Punktzahl der anderen Spieler im aktuellen Universum betrachten.
- Ein Spieler wird anhand von Systemmeldungen über die für ihn relevanten Vorgänge auf dem Laufenden gehalten.

4.2. Nicht-funktionale Anforderungen

Um eine weitreichende und angenehme User Experience, sowie die Möglichkeit zur Pflege und Erweiterung der Software zu bieten, müssen folgende nicht-funktionale Anforderungen beachtet werden.

4.2.1. Geräteunterstützung

Um eine breite Masse an Spielern ansprechen zu können, müssen Internetspiele in der heutigen Zeit eine breite Palette an Geräten, Plattformen und Webbrowsern unterstützen. Da dies jedoch den Rahmen dieser Arbeit weit übersteigen würde, musste hiervon Abstand genommen werden. *CombatZone* unterstützt daher ausschließlich Windows-basierte Geräte sowie den Internetbrowser “Google Chrome“ ab Version 23.

Mit weiterer Optimierung wäre die Unterstützung auf zusätzliche Geräte und Internetbrowser ergänzbar. Die Unterstützung von mobilen Geräten (wie z.B. Apples iPad oder Smartphones) wäre hierbei grundsätzlich ebenso möglich.

4.2.2. Performanz

Bei der Anforderungsanalyse für die Spielperformanz muss zwischen dem User Interface und der serverseitigen Berechnung unterschieden werden.

Die Performanz-Anforderung der Benutzeroberfläche ist dann erfüllt, wenn unter normaler Auslastung des darstellenden Gerätes die Darstellung der Anwendung eine flüssige User Experience bietet. Wichtig hierfür ist, dass die Oberfläche stets reaktionsfähig bleibt.

Auf Seite des Servers ist die Performanz oft stark verknüpft mit der Skalierbarkeit. Auch wenn das Skalierbarkeitsziel voll ausgeschöpft wird, darf die Performanz nicht unter das definierte Latenzminimum von 300 ms sinken.

Eine schlechte serverseitige Performanz kann eine Verschlechterung der Performanz der Benutzeroberfläche nach sich ziehen. Daher ist es wichtig, keine der beiden Aspekte zu vernachlässigen.

4.2.3. Skalierbarkeit

Der Erfolg von Internetspielen kann nur sehr schwer im Vornherein abgeschätzt werden. Daher ist es wichtig, dass sich die Infrastruktur flexibel an sich ändernde Anforderungen anpassen kann.

Als Hosting-Plattform bietet sich hierfür die Cloud an, da die der Anwendung zur Verfügung stehenden Ressourcen (virtuelle Maschinen, Warteschlangen, etc.) leicht angepasst werden können. Eine Anwendung muss jedoch entsprechend skalierbar konzipiert sein, um sich an die sich ändernden Ressourcen anpassen zu können, sei es nun eine Erhöhung oder Verringerung der Ressourcen.

Für ein Internet-Spiel ist das Hauptmerkmal, an dem die Skalierbarkeit gemessen werden kann, die Anzahl der Benutzer, die das System mit akzeptabler Latenz verkraften kann.

Das Skalierbarkeitsziel für *CombatZone* beträgt 1500 registrierte Benutzer, wovon durchschnittlich 200 Benutzer gleichzeitig angemeldet sind.

4.2.4. Verfügbarkeit

Die Gründe für eine Beeinträchtigung der Verfügbarkeit einer Software lassen sich in zwei Kategorien unterteilen, ungeplante und geplante Ausfälle.

Ungeplante Ausfälle

Durch die hohe Spielgeschwindigkeit von *CombatZone* ist eine hohe Verfügbarkeit essenziell. Wenn die Auswirkungen der meisten Spieleraktionen (wie z.B. das Verschicken von Raumschiffen zu einem angrenzenden Sektor) bereits nach wenigen Sekunden oder Minuten abgeschlossen sind, so würde eine mangelnde Verfügbarkeit das aktive Spielgeschehen nachhaltig negativ beeinflussen.

Geplante Ausfälle

Wie jede Software wird auch *CombatZone* aktive Wartung und dementsprechende Aktualisierungen benötigen. In vielen Anwendungsgebieten können diese leicht ohne Beeinflussung des produktiven Betriebs außerhalb der Geschäftszeiten durchgeführt werden.

Da die Laufzeit der Universen des Spiels sich jedoch überlappen, bietet die Anwendungsdomäne von *CombatZone* keine natürliche Wartungszeiträume. Dementsprechend wäre es vorteilhaft wenn eine Möglichkeit implementiert werden würde, welche es ermöglicht, auch bei einem (partiell) laufendem System Aktualisierungen durchzuführen, ohne die User Experience der aktiven Spieler zu

beeinflussen.

Optimal wäre selbstverständlich eine Architektur bzw. ein Deployment-Schema, welches eine 100-prozentige Verfügbarkeit während geplanter Ausfälle bieten kann.

4.2.5. Datensicherheit

Das Fortbestehen der Spieldaten und der Schutz der Konsistenz dieser Daten ist essenziell für das Schicksal des Spiels. Durch Hardware-Ausfall oder Datenkorruption zerstörte Spieldaten würden nicht nur den Ruf der Entwickler-/Betreiberfirma schaden, sondern auch den potenziellen Verlust von Spielern bedeuten.

Es ist daher zwingend notwendig, dass eine Sicherung der Spieldaten durchgeführt werden kann. Dies sollte, wenn möglich, ohne Unterbrechung des normalen Spielbetriebs geschehen. Eine Automatisierung der Sicherung wäre vorteilhaft, um Administrator-Aufwand und das Risiko für menschlichen Fehler (Sicherung vergessen etc.) zu minimieren.

4.2.6. Wartung

Die Anwendung soll nur wenig Wartung erfordern. Dies senkt den Administrator-Aufwand und dementsprechend auch die Ressourcen-Auslastung bzw. die Kosten. Ist jedoch Wartung notwendig, so soll diese zielgerichtet und effizient durchgeführt werden können. Zu diesem Zweck ist es notwendig, dass sich ein Administrator oder Entwickler einen Überblick über die aktuelle Auslastung und Vorgänge machen kann.

5. Entwurf

Das Kapitel Entwurf befasst sich mit der Umsetzung der in der *Analyse* herausgefundenen Anforderungen zu implementierbaren Modellen. Hierzu wird zunächst die Domänenlogik erarbeitet, dann die verwendete Architektur ausgewählt und anschließend einige spezifischen Komponenten genauer erläutert.

5.1. Überführung in Domänenlogik

Der Kern einer jeden Anwendung ist deren Domänenlogik, im Falle von *CombatZone* also die abstrahierten Spielelemente. Diese müssen zunächst betrachtet werden, ehe die hierfür benötigte Infrastruktur konzipiert werden kann.

5.1.1. Bounded Contexts

Bereits beim Erfassen der Anforderungen an *CombatZone* fiel auf, dass es zwei mögliche Bedeutungen für den Begriff "User" gibt. Einerseits kann hiermit der Mensch gemeint sein, welcher sich auf die Homepage des Spiels begibt, um Interesse zu bekunden und sich für das Spiel zu registrieren. Andererseits wird innerhalb des späteren Spiels stark zwischen den verschiedenen Universen, in denen der User mitspielt, differenziert. Da es sinnvoll ist, diesen Grundgedanken auch in den Code mit einfließen zu lassen (bzw. DDD dies sogar vorschreiben würde, da es sich um Domänenlogik handelt), hat man es hier eindeutig mit einem Begriffs-Konflikt zu tun. Einige User-relevante Operationen sind Universum-übergreifend (z.B. Passwort ändern), andere wiederum sind fest mit der Instanz eines Universums verknüpft (z.B. Flotte verschicken).

Diese Erkenntnis führte zur ersten großen Entscheidung in der Konzeption der Domänenlogik, die Unterteilung in die zwei Bereiche (nach DDD: Bounded Contexts) "Identity" und "Realm/Universe".

Während der Bounded Context "Identity" sich primär um die User-Verwaltung, inklusive der damit einhergehenden Verwaltungsfunktionen (Verifikation, Sperrlogik, usw.) kümmert, enthält "Realm" jedwede Funktionalität, welche direkt mit Spielelementen assoziiert ist.

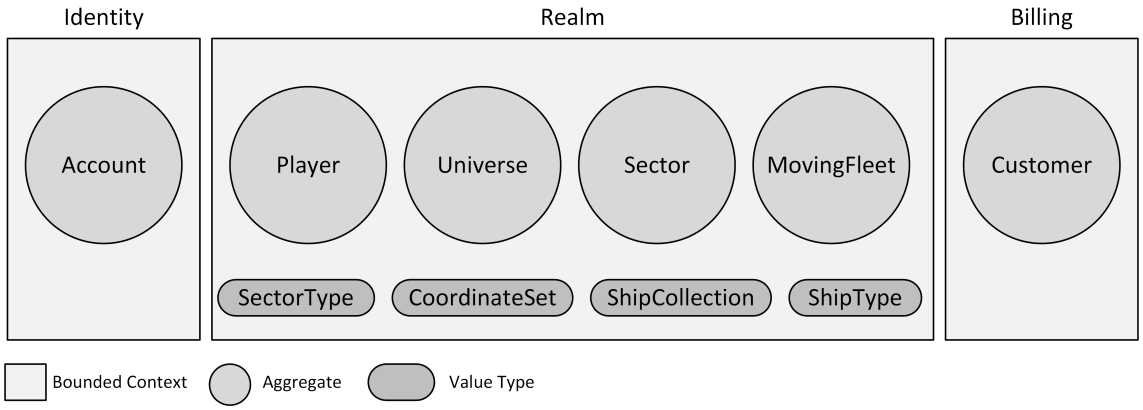


Abbildung 5.1.: Übersichtsbild der Bounded Contexts

Des Weiteren gibt es noch den Bounded Context "Billing". Dieser beinhaltet die wirtschaftlich-orientierte Domänenlogik rund um bezahlte Inhalte.

Einen Überblick über die vorhandenen Bounded Contexts und deren Bestandteile ist in Abbildung 5.1 zu betrachten. In den folgenden Abschnitten wird nun genauer auf ausgewählte Elemente der Domänen-Logik eingegangen.

5.1.2. Bounded Context: Identity

Der Bounded Context *Identity* bündelt alle Aspekte der Domänen-Logik, welche sich um die Identitätsverwaltung der Applikation drehen.

Der Aggregate *Account* ist der einzige Aggregate im "Identity" BC. Er verkörpert einen registrierten Benutzer. Jedwede Authentifizierung und Autorisierung läuft über diesen Aggregate.

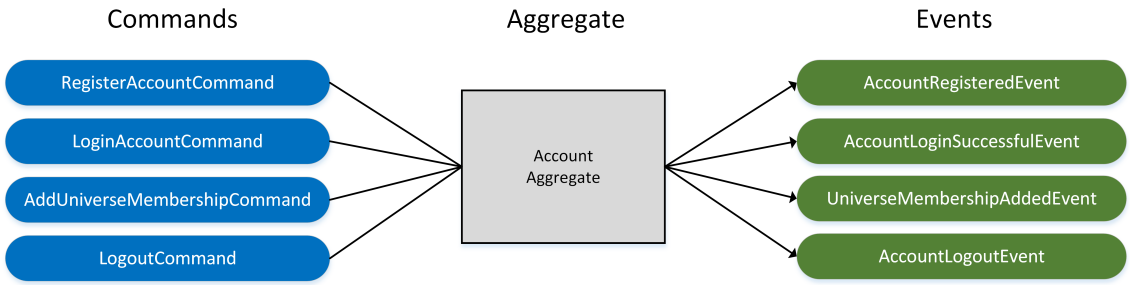


Abbildung 5.2.: Aggregate "Account"

In Abbildung 5.2 sind die mit Account assoziierten Commands und Events abgebildet.

5.1.3. Bounded Context: Realm

Der Bounded Context *Realm* beinhaltet alle Universen von *CombatZone*. Der Aggregate "Universe" ist eine Instanz eines Universums, alle weiteren Aggregates verkörpern jeweils ein weiteres, dem Universum untergeordnetes Konzept des Spiels.

Aggregates

Der größte Aggregate in Realm ist *Sector*. Wie der Name bereits preisgibt, verkörpert dieser Aggregate einen Sektor innerhalb eines Universums. Die Aggregate-Grenzen wurden hierbei so gewählt, dass die Konsistenz innerhalb eines Sektors stets gewahrt wird; folglich kann es nicht zu einer Situation kommen, in der zwei unterschiedliche Spieler versuchen, denselben Sektor gleichzeitig zu übernehmen. Da ein Aggregate automatisch als Konsistenzgrenze (siehe Abschnitt 2.2.2) gilt, ist solch eine Situation bereits von vornherein ausgeschlossen worden. Die zum Sector-Aggregate gehörenden Commands und Events sind in Abbildung 5.3 dargestellt.

Domain Services

Für die Implementierung der Aggregate-übergreifenden Domänenlogik wurden fünf Domain Services vorgesehen. Diese sind in Tabelle 5.1 aufgeführt.

Workflows

Für Domänen-Aktivitäten, welche über den Zuständigkeitsbereich von einzelnen Aggregates hinausgehen, werden Workflows verwendet. Für Bounded Context "Realm" wurden die vier in Tabelle 5.2 aufgeführten Workflows vorgesehen.

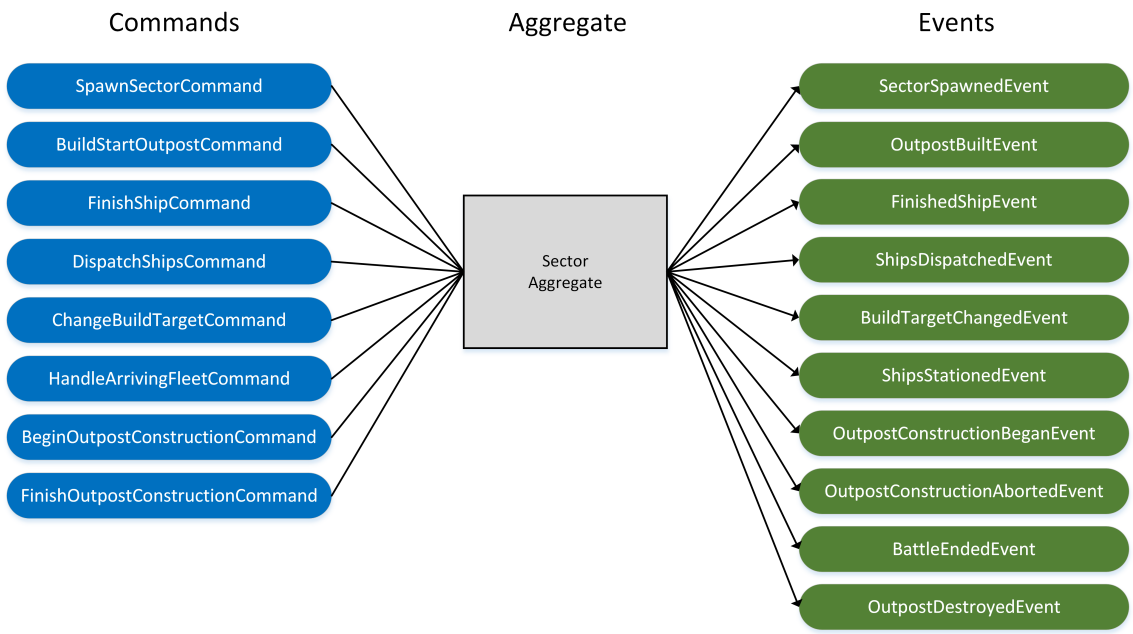


Abbildung 5.3.: Aggregate “Sector“

Bezeichnung	**Aufgabengebiet**
Battle Domain-Service	Auswertung von Sektor-Kämpfen
Fleet Movement Domain-Service	Berechnung von Flotten-Bewegungen
Random Domain-Service	Zufallswert-Erzeugung
Sector Management Domain-Service	Erzeugung und Verwaltung von Sektoren
Ship Build Domain-Service	Berechnung von Schiffsproduktionen

Tabelle 5.1.: Domain Services von BC “Realm“

5.1.4. Bounded Context: Billing

Der Bounded Context *Billing* kapselt die Domänen-Logik der wirtschaftlichen Aspekte des Systems. Der einzige Aggregate von Billing ist *Customer*. Im Customer-Aggregate werden genug Informationen vorgehalten, damit kostenpflichtige Aktionen von Kunden notiert und ausgelesen werden können.

Der “Billing“ BC enthält exakt einen Domain Service, den *RealmInputDomainService*. Dieser empfängt die veröffentlichten Events vom “Realm“ BC und ist dafür zuständig, kostenpflichtige Aktionen für spätere Verwendung zu hinterlegen.

5.1.5. Unterschiede in Skalierbarkeitsanforderungen

Die Bounded Contexts “Identity“ und “Billing“ werden ausschließlich von selten auftretenden User-Aktionen (Registrierung, Login, Universumsbeitritt etc.) verwendet, daher sind sie keiner allzu hohen Last ausgesetzt.

“Realm“ hingegen wird mit jedem Spieler, welcher einem Universum beitritt, stärker beansprucht, selbst wenn dieser Spieler gar nicht eingeloggt ist (z.B. Bauprozesse). Für diesen BC muss also weit mehr Bedacht auf Skalierbarkeit gesetzt werden, sodass die Erreichbarkeit und Performanz des Spiels nicht durch steigende Spielerzahlen in Mitleidenschaft gezogen wird. Auf diese Problematik wird später detailliert eingegangen, u.a. in den Abschnitten 5.4.1 und 10.4.

Bezeichnung	Aufgabengebiet
Fleet Movement Workflow	Flotten-Bewegung starten
Grow Universe Workflow	Wachstum von Universen ermöglichen
New Player Workflow	Neue Mitgliedschaften an Identity weiterleiten
Sector Creation Workflow	Sektor-Erzeugung vorantreiben

Tabelle 5.2.: Workflows von BC “Realm“

5.2. Architektur

Für alle drei Bounded Contexts von *CombatZone* soll Command-Query Responsibility Segregation (CQRS) aus Kapitel 2.6 als Architektur angewendet werden. Dies ermöglicht es, den gegebenen hohen Anforderungen an Skalierbarkeit und Flexibilität gerecht zu werden. Durch die Möglichkeiten der zwischen den BCs differenzierten Skalierbarkeit kann die zur Laufzeit anfallende Last leichter verteilt und als solches eine bessere User Experience geboten werden.

CQRS ist eine sehr anpassbare Architektur, weswegen Architektur-Diagramme verschiedener auf CQRS-basierender Systeme oftmals sehr unterschiedlich aussehen. In Abbildung 5.4 ist die Architektur für einen spezifischen Bounded Context dargestellt.

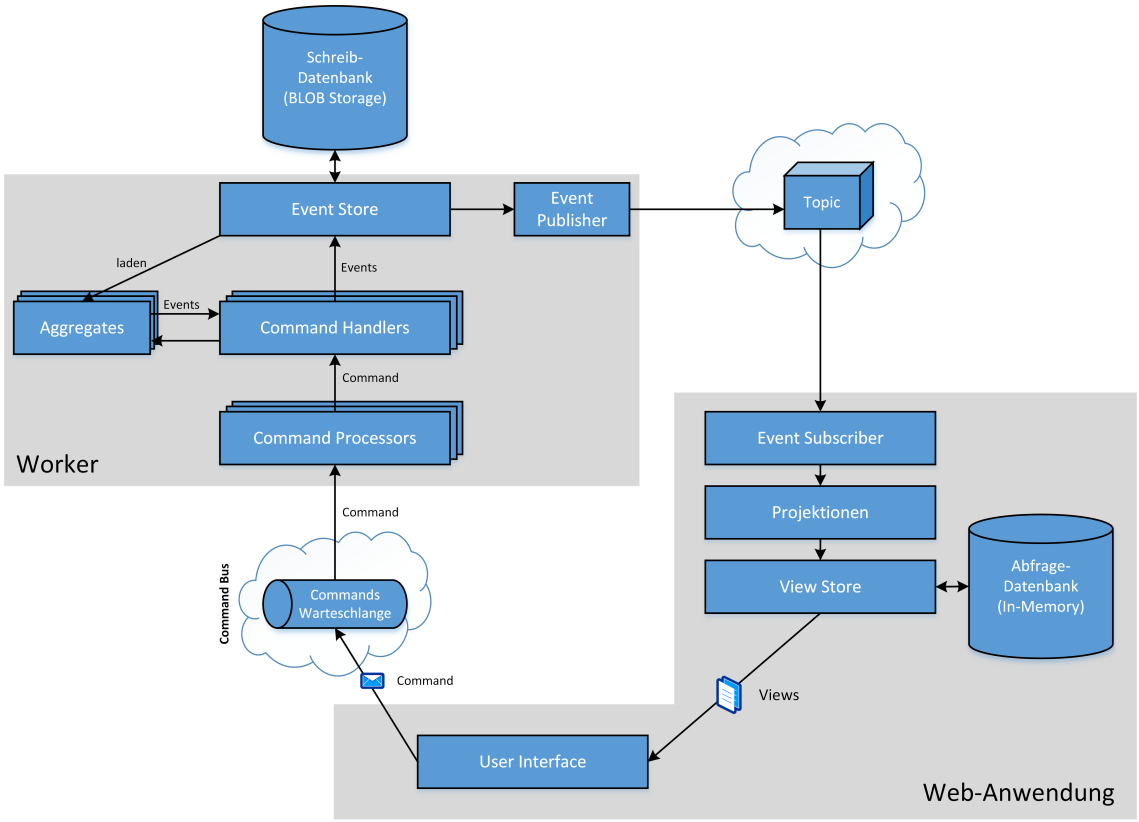


Abbildung 5.4.: Skizze der CombatZone-Architektur

Die *CombatZone*-Infrastruktur ist aus zwei Anwendungen aufgebaut, der Web-Applikation und dem Worker-Prozess. Das User Interface der Web-Anwendung kann Commands an den Command Bus versenden und Views vom View Store abfragen. Die Commands werden vom Worker verarbeitet und die daraus resultierenden Events im Event Store gespeichert, sowie im Topic der Publish-Subscribe-Infrastruktur veröffentlicht. Diese Events werden von der Web-Anwendung wie-

derum verwendet, um die Views im View Store zu aktualisieren.

In den folgenden Abschnitten wird zunächst auf diese beiden Anwendungen spezifischer eingegangen und anschließend die komplexeren Kern-Komponenten, wie z.B. das Command Handling und der Event Store, näher beleuchtet.

5.3. Web-Anwendung

Die Web-Anwendung soll Server-seitig auf ASP.NET MVC und ASP.NET Web API basieren. Diese beiden Technologien ermöglichen es, zusammen einen REST-basierten Web Service für die Client-seitige Spiel-Oberfläche zu bieten.

Als Client-seitige Spiel-Oberfläche ist eine HTML5-basierte *Single Page Application* geplant, d.h. alle zusätzlichen Daten und Ressourcen werden im Hintergrund nachgeladen und alle weiteren Oberflächen-Elemente werden als Dialoge über die vorhandene Ansicht gelegt. Der Zugriff auf den Web Service geschieht via jQuery und JSON als Format der transferierten Daten.

Die Spiel-Oberfläche und der Web Service besitzen eine konstante Verbindung, damit Änderungen der Spiele-Daten mit geringer Latenz eingepflegt werden können.

5.3.1. Datenzugriff

Beim Start einer Instanz lädt sich diese alle aktuellen Events aus dem Event Store, generiert daraus via den Client-seitigen Projektionen alle benötigten Views und abonniert sich in der Publish-Subscribe Messaging-Infrastruktur, um auch alle nachfolgenden Events einpflegen zu können. Jeder Datenzugriff, welcher durch die Web Services notwendig ist, kann so hochperformant direkt aus dem Arbeitsspeicher erfolgen.

5.4. Worker-Prozess

Der Worker-Prozess hostet ein oder mehrere Bounded Contexts auf einem Windows-basierten Server. Diese Bounded Contexts beherbergen Command Processors, die eingehende Commands verarbeiten.

5.4.1. Partitionierung

Ausgehend von den drei Bounded Contexts, die *CombatZone* besitzt, bieten sich verschiedene Schemata an, nach denen partitioniert werden könnte.

Nachfolgend wird von einer "Worker-Instanz" gesprochen. Damit ist sowohl der Prozess an sich gemeint, als auch die virtuelle Maschine, auf welcher dieser Prozess gestartet wurde. Dies ist damit begründet, dass unter Windows Azure üblicherweise nur ein User-Prozess pro virtueller Maschine (bzw. Web Rolle) vorgesehen ist.

Schema A

In Abbildung 5.5 ist mit Schema A das simpelste Schema sichtbar, welches angewendet werden kann. Hier befinden sich alle drei Bounded Contexts innerhalb einer einzigen Worker-Instanz mit jeweils einem Command Processor. Dieses Schema kann ausschließlich vertikal skaliert werden. Da die BCs sich einen Prozess teilen, betrifft ein Ausfall (sowohl geplant als auch ungeplant) stets alle BCs.

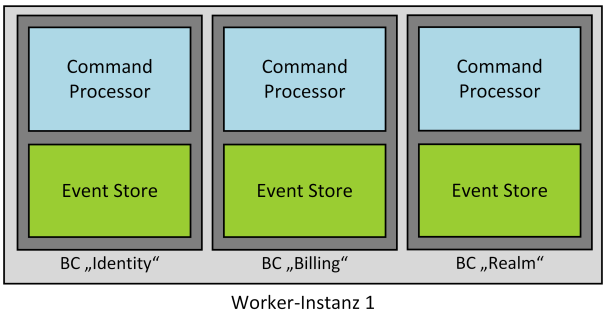


Abbildung 5.5.: Partitionierungsschema A

Schema B

Schema B in Abbildung 5.6 erweitert Schema A, indem die Command Processors vom “Realm“ BC in vier Partitionen unterteilt wurden.

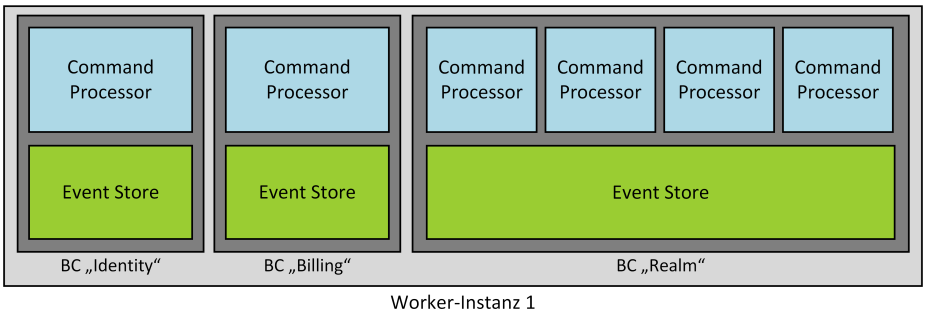


Abbildung 5.6.: Partitionierungsschema B

Da jeder Command Processor auf einem einzelnen Thread basiert, kann er stets nur einen Command gleichzeitig abarbeiten. Durch eine Erhöhung auf mehrere Command Processors kann diese Anzahl erhöht werden und somit ein höheres Command-Aufkommen bearbeitet werden, wie es bei “Realm“ erwartet wird.

Um zu verhindern, dass eine parallele Verarbeitung von Commands die Konsistenz innerhalb von Aggregates gefährdet, muss sichergestellt werden, dass jede Berechnung einer Aggregate-Instanz stets auf demselben Command Prozessor (und somit Thread) geschieht. Der einfachste Weg, dies zu bewerkstelligen, ist ein Hashing-Verfahren zu verwenden, welches jeder Aggregate ID (GUID) eine eindeutige Partitionsnummer zuweist. Ein Beispiel hierfür ist in Abbildung 5.7 zu sehen.

$$\text{GetPartition(AggregateId)} = \text{AggregateId[0]} \begin{cases} \text{wenn } 0 \vee 1 \vee 2 \vee 3 & \text{dann } 0 \\ \text{wenn } 4 \vee 5 \vee 6 \vee 7 & \text{dann } 1 \\ \text{wenn } 8 \vee 9 \vee A \vee B & \text{dann } 2 \\ \text{wenn } C \vee D \vee E \vee F & \text{dann } 3 \end{cases}$$

Abbildung 5.7.: Simpler Hashing-Algorithmus

Die Unterteilung in vier Partitionen ist hier nur exemplarisch. So lange der Hashing-Algorithmus dies unterstützt, kann eine beliebige Anzahl an Partitionen eingerichtet werden. Da jede Command Processor Instanz jedoch Systemressourcen verbraucht und sich auch in Schema B alle BCs einen Prozess teilen, ist es oftmals sinnvoller eines der weiteren Partitionierungsschemata zu verwenden, wenn sehr viele Partitionen gewünscht sind.

Schema C und D

Dies kann entweder durch bündeln von BCs in einzelne Prozesse erfolgen (siehe Abbildung 5.8), vergleichbar einer n:1 Bindung, oder der Zuweisung dedizierter Prozesse zu jedem BC (siehe Abbildung 5.9), welches einer 1:1 Bindung gleichkäme.

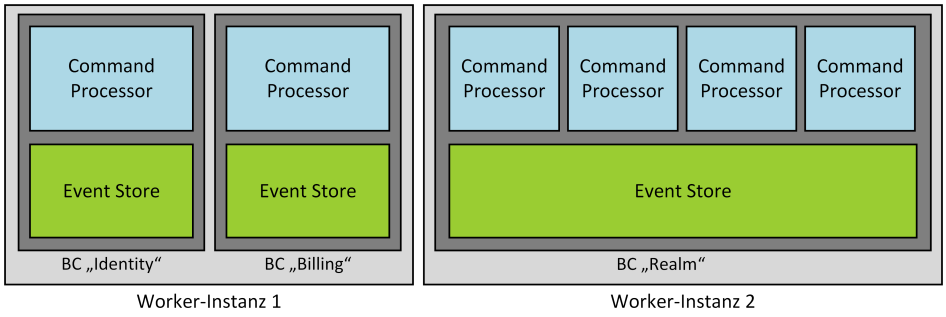


Abbildung 5.8.: Partitionierungsschema C

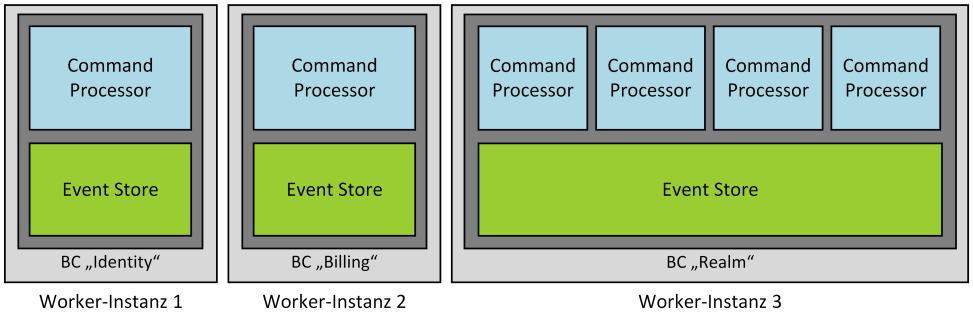


Abbildung 5.9.: Partitionierungsschema D

Weitere Schemata

Alle weiteren Möglichkeiten zur Partitionierung würden es entweder erforderlich machen, dass eine Event Store Instanz auch außerhalb eines Prozesses angesprochen werden kann oder, dass der Event Store eines BCs ebenfalls partitioniert wird. Hierauf wird im Kapitel *Ausblick* kurz eingegangen.

Anwendung

In Version 1.0 von *CombatZone*, welche dieser Arbeit beiliegt, wird das Partitionierungsschema B verwendet, da dies ein gutes Mittelmaß zwischen physischer Aufteilung und Komplexität darstellt. Eine Erweiterung in Schema D wäre jedoch nachträglich ohne großen Aufwand realisierbar, da die Bounded Contexts entkoppelt konzipiert werden. Diese Aufteilung wäre jedoch mit weiteren Kosten seitens Windows Azure verbunden, daher wird dies verschoben bis steigende User-Zahlen diesen Aufwand notwendig machen.

5.5. Messaging-Elemente

Wie bei der Verwendung von CQRS üblich, können die Nachrichten unterteilt werden in Commands und Events. Für diese Applikation wurde diese Unterteilung noch weiter geführt, siehe hierzu die Abbildung 5.10.

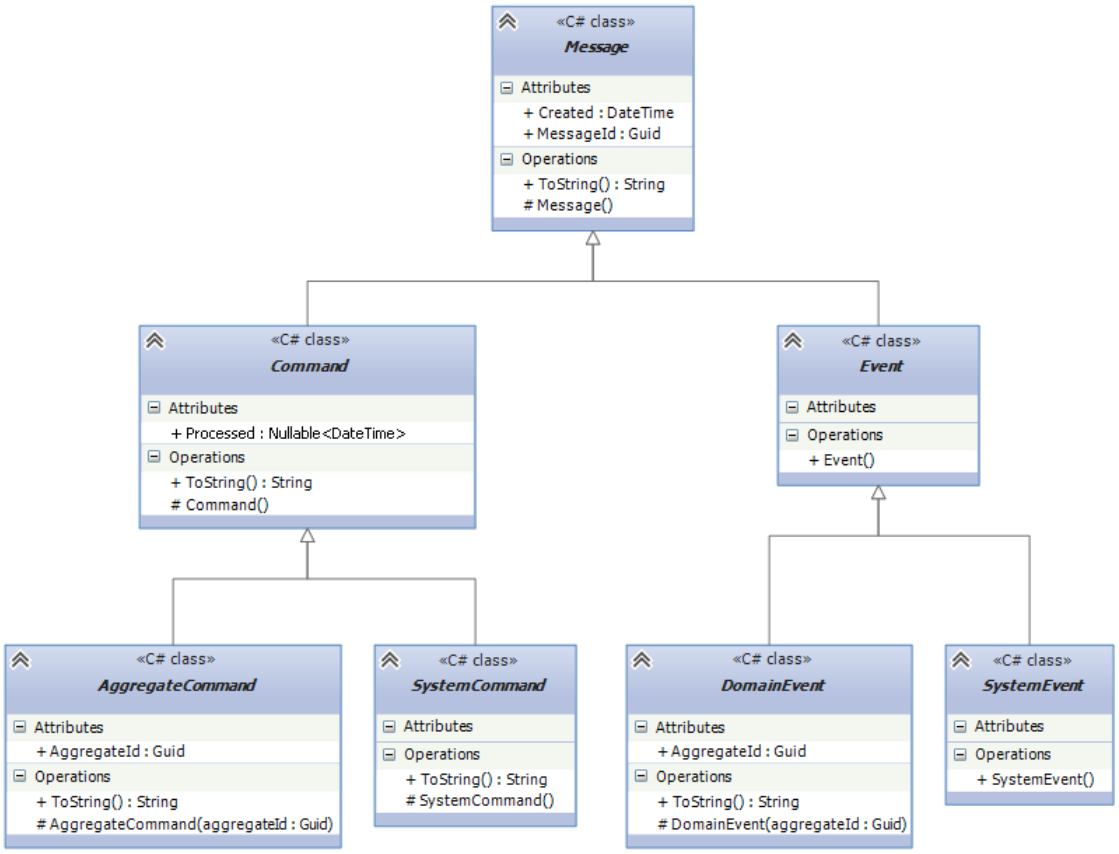


Abbildung 5.10.: Nachrichtenklassen

AggregateCommand bezeichnet hierbei ein Command, welches mit einer spezifischen Aggregate-Instanz assoziiert ist. Dies ist relevant, da das Routing der Nachricht anhand der Aggregate-ID erfolgt. Das ist essenziell, damit – trotz der Verteilung der Berechnung – die Konsistenzgrenzen von Aggregates eingehalten werden können. Würden Commands derselben Aggregate-Instanz durch Parallelisierung gleichzeitig ausgeführt werden, müssten Sicherheitsmaßnahmen wie z.B. *Optimistic Concurrency* implementiert werden.

SystemCommands werden für Funktionen der Infrastruktur verwendet, welche nicht Teil der Geschäftsdomäne sind. Diese Art von Commands haben dementsprechend keine Konsistenzgrenzen, daher ist deren korrektes Routing irrelevant und benötigt dementsprechend keine weiteren Attribute. Das Versenden von E-Mails wird in *CombatZone* via SystemCommands implementiert.

Dasselbe Schema ist auf *DomainEvents* und *SystemEvents* übertragbar. Beispiele für SystemEvents sind BoundedContextStartedEvent und BoundedContextStoppedEvent, welche beim Starten und Stoppen der jeweiligen BCs erstellt werden.

5.6. Command Handling

Jeder der drei Bounded Contexts ist in der Lage, Commands entgegenzunehmen. Diese werden von einem Client erstellt und an die Warteschlangen des Command Bus übermittelt.

Da ein direkter Zugriff vom Client-seitigen UI auf den Command Bus zu unsicher wäre, werden Anfragen der UI zunächst (via HTTP POST) an eine Web API gestellt, welche diese validiert und bei Erfolg diese in Commands umwandelt und an den Command Bus weitergibt. Der Prozess der Web API ist vertrauenswürdig genug, um diesen direkten Zugriff zu erlauben.

Für eine bessere Skalierbarkeit und Lastverteilung soll das Command Handling asynchron geschehen. Hierdurch können leichter Spitzen in der Anzahl der erstellten Commands ausgeglichen werden und es erleichtert die Entkoppelung zwischen den verschiedenen Komponenten.

5.6.1. Ablauf einer Command-Transaktion

Beim Start eines Bounded Context generiert dieser einen oder mehrere *Command Processors*. Diese horchen innerhalb der ihnen zugewiesenen Warteschlage auf eingehende Commands. Trifft ein Command ein, so wird dieses deserialisiert und der für den Command-Typ zuständige *Command Handler* gesucht und aufgerufen.

Der *Command Handler* lädt nun die Command-betreffende Aggregate-Instanz aus dem Event Store und führt auf dieser die zum Command passende Domänen-Methode aus. Anschließend gibt der Command Handler das Command und die neuen vom Aggregate erzeugten Events an den Event Store zur Persistierung weiter.

Nach der erfolgreichen Bearbeitung des Commands wird die entsprechende Nachricht aus der Warteschlange gelöscht. Anschließend bemerkt ein Hintergrundprozess die neu hinzugefügten Events im Event Store und veröffentlicht diese. Dies wird in Abschnitt 5.7.4 genauer erläutert.

5.6.2. Poison Handling

Sollte bei der Bearbeitung eines Commands ein Fehler auftreten, so wird die Bearbeitung der Nachricht erneut beginnen. Sollte das Problem, weswegen die Bearbeitung fehlschlägt, nicht temporär sein (z.B. weil ein Bug im Programm-Code vorliegt), so könnte dies zu einer Endlosschleife führen.

Um dies zu verhindern, wird nach dem dritten fehlgeschlagenen Versuch die Nachricht in eine spezielle Warteschlange, die sogenannte *Poison Queue*, verschoben und ein Administrator oder Entwickler benachrichtigt. Dieser muss nun herausfinden, ob ein wirkliches Problem vorliegt oder ob die Nachricht verworfen werden kann.

Ist die Ursache in einem Bug begründet, so kann – nachdem der Bug korrigiert wurde – die Nachricht aus der Poison Queue wieder in die normale Command-Warteschlange zurückgeschoben werden. Der – jetzt korrekt funktionierende – Bounded Context würde nun die Nachricht erneut empfangen und bearbeiten, ohne dass Daten verloren gingen.

In der Implementation von *CombatZone* besitzt jeder Bounded Context seine eigene Poison Queue.

5.6.3. Umgang mit asynchroner Verarbeitung

Eine asynchrone Verarbeitung kann jedoch einen negativen Einfluss auf die User Experience haben, wenn die damit einhergehenden Auswirkungen nicht entsprechend bedacht werden. User gehen gewöhnlich davon aus, dass Änderungen, welche sie in einem System durchführen, auch sofort Wirkung zeigen. Da sowohl durch die asynchrone Verarbeitung als auch durch “Eventual Consistency“

dies bei *CombatZone* jedoch nicht der Fall ist, muss hierfür eine Lösung gefunden werden, welche die User Experience des Menschen nicht beeinflusst.

Hierfür gibt es unterschiedliche Varianten:

Blockierendes UI

Nach dem Versenden des Befehls, welcher asynchron verarbeitet wird, verhindert das UI weitere Verwendung seitens des Users und blendet einen Hinweis ein, dass dieser auf die Fertigstellung der Befehlsbearbeitung zu warten hat. Erforderlich ist hierfür, dass das UI eine Möglichkeit hat, entweder in einem Intervall das Ergebnis der Bearbeitung zu prüfen (Polling) oder über dieses benachrichtigt zu werden (Push).

Diese Variante erlaubt es zwar, seitens des Servers, die Vorteile der asynchronen Verarbeitung auszunutzen, für den User bietet es jedoch keinen Vorteil. Je nach Art des Befehls und der Dauer der Wartezeit kann die Blockierung des UI zu Unmut oder Unverständnis seitens des Users führen. Handelt es sich beispielsweise um einen Login oder eine langwierige Kalkulation, bei der der User nachvollziehen kann, dass eine weitere Benutzung die Fertigstellung dieses laufenden Prozesses erforderlich macht, mag er vielleicht noch Verständnis zeigen. Da dies oft jedoch nicht der Fall ist, muss diese Variante mit Bedacht gewählt werden.

Hintergrund-Verarbeitung mit Hinweis

Eine Alternative zu dem blockierenden UI ist es, den User weiterarbeiten zu lassen, ihn jedoch darauf hinzuweisen, dass seine letzte Aktion gegebenenfalls noch nicht umgesetzt wurde. Dies kann beispielsweise durch einen simplen Hinweis erfolgen, dass die Änderung 15 Minuten dauern kann. Wahlweise kann innerhalb des User Interface auch ein Zeitstempel angezeigt werden, der angibt, auf welchem Datum die derzeit präsentierten Daten basieren. Da dieser Zeitstempel beim Darstellen bereits in der Vergangenheit liegt, erklärt dies dem User automatisch, warum seine Änderungen noch nicht sichtbar sind.

Die Umsetzbarkeit dieser Variante ist ebenfalls stark davon abhängig, ob der User Verständnis für die Zeitverzögerung hat. In vielen Geschäftsdomänen gibt es Aktivitäten, in denen der User bereits davon ausgeht, dass diese nicht sofort umgesetzt werden. Stornierungen sind hierfür ein beliebtes Beispiel.

Hintergrund-Verarbeitung ohne Hinweis

Auf den Hinweis über die potenziell noch nicht erfolgte Änderung kann in manchen Fällen auch verzichtet werden.

Wenn die vom User gestartete Operation keinen Einfluss auf die nachfolgenden Schritte innerhalb des UI hat, der User also nicht bemerken würde, dass seine Änderung noch nicht durchgeführt wurde, so kann dies einfach verschwiegen werden.

Ist dies jedoch nicht der Fall, kann die Änderung innerhalb der Client-Anwendung zwischengespeichert werden. Bei der Anzeige von Daten können diese zwischengespeicherten Daten dann in die eigentlichen Daten integriert werden und dem User so vorgegaukelt werden, dass diese sich bereits im eigentlichen Datenbestand befinden. Dies kann jedoch leicht zu einer dramatisch erhöhten Komplexität der Client-Anwendung führen, weswegen diese Variante ebenfalls mit Bedacht gewählt werden sollte.

Kombination

Für *CombatZone* werden zwei verschiedene Varianten eingesetzt.

Der Login ist eine blockierende Operation, bei der die Applikation im Hintergrund via Polling auf eine Beendigung wartet und den User anschließend in die Spiel-Oberfläche weiterleitet.

Für die Operationen innerhalb der Spiel-Oberfläche wird dem User die Verzögerung verschwiegen und stattdessen im lokalen Spieldaten-Cache die Veränderung vorzeitig durchgeführt. Sobald die Änderung Server-seitig umgesetzt wurde, wird der Client benachrichtigt und passt seinen lokalen Cache an die wahren Daten an.

5.6.4. Wahl der Warteschlangenlösung

Für die spätere Übermittlung von Nachrichten wird eine zuverlässige Warteschlangenlösung benötigt.

Um auch bei synchronen Aktionen eine annehmbare User Experience zu bieten (s. Abschnitt 4.2.2), wird eine maximale Latenz von 100 Millisekunden für den Transfer von Nachrichten vorausgesetzt.

Windows Azure bietet zwei verschiedene Lösungen für persistente Warteschlangen an, den *Windows Azure Queue Service* sowie den *Windows Azure Service Bus*. Während erstere Lösung auf dem Windows Azure Storage Service aufbaut (welcher ohnehin von der Anwendung verwendet werden wird), basiert der Windows Azure Service Bus auf einem eigenen System, welches explizit für hochskalierbare Messaging-Szenarien mit hoher Ausfallsicherheit und Enterprise-Messaging Features (wie z.B. De-Duplizierung) konzipiert wurde.

Eine Alternative wäre, ein eigenes Queueing-System auf einer Windows Azure-basierten virtuellen Maschine zu betreiben, beispielsweise auf Basis von RabbitMQ oder NServiceBus. Dies würde jedoch zusätzlichen Implementations- und Wartungsaufwand mit sich bringen und ist zur Erfüllung der gestellten Anforderungen weder notwendig noch sinnvoll.

Funktionen

Das Funktionsspektrum der beiden untersuchten Warteschlangenlösungen unterscheidet sich in vielen Bereichen. Anhand von [13] werden nachfolgend die wichtigsten Unterschiede dargestellt.

	Queue Service	Service Bus Queue
Empfangsart	Polling	Polling & Long Polling
Gebündeltes empfangen	Ja (Explizit)	Ja (Implizit)
Gebündeltes versenden	Nein	Ja
Maximale Nachrichtengröße	64 KB (Base64: 48 KB)	265 KB
Erfassbare Inhaltsmenge	Schätzwert	Genauer Wert
Durchschn. Latenz	10 ms	100 ms
Kosten pro 1 Million Nachrichten *	0,08 €	0,71 €

Tabelle 5.3.: Vergleich der Warteschlangenlösungen

Die Kosten enthalten nicht eventuell anfallende weitere Gebühren für Speicherplatz und Transfer der Daten.

Die Möglichkeit der Abfrage der genauen Anzahl an Nachrichten innerhalb einer Warteschlange kann sich als sehr hilfreich erweisen bei der Umsetzung einer Wartungsanwendung, welche es Administratoren erlaubt, sich eine bessere Übersicht über die Auslastung des Systems zu verschaffen.

Kosten

Im Abschnitt 4.2.2 des vorherigen Kapitels *Analyse* wurde die maximale Zeit, welche die Verarbeitung eines Commands benötigen darf, auf 250 Millisekunden beschränkt. Damit dies eingehalten werden kann, darf die Nachrichten-Infrastruktur keine Latenz höher als 50 Millisekunden aufweisen.

Auf den ersten Blick bietet die Warteschlangenlösung vom Windows Azure Queue Service sowohl eine weitaus bessere Latenz als auch einen niedrigeren Preis.

Abgerechnet wird bei beiden Lösungen jedoch anhand der Anzahl der Anfragen an den Dienst. Dies beinhaltet jedes Versenden und Empfangen einer Nachricht inklusive aller Anfragen an eine leere Warteschlange.

Eine Warteschlange vom Queue Service unterstützt ausschließlich Polling, also das explizite Nachfragen nach neuen Nachrichten. Ausgehend von einer erwünschten Latenz von nicht mehr als 100 ms müsste eine Queue Service Warteschlange also mindestens 10-mal in einer Sekunde abgerufen werden. Dies ergibt ungefähr 26.784.000 Abrufe in einem Monat. Bei einem Preis von 0.08 € pro 1 Million Abrufen ergibt dies einen minimalen Kostenaufwand pro Warteschlange pro Monat von 2,14 €. “Minimal“ daher, da diese einfache Rechnung nicht die Erhöhung der Kosten durch versendete Nachrichten oder den erneuten Abruf innerhalb von 100 ms nach dem Empfang einer Nachricht einschließt.

Um die Kosten zu senken, wäre es möglich, die Anzahl der Abrufe anhand von Auslastung oder Zeitplänen partiell zu senken. Dies würde jedoch die erlaubte Latenz überschreiten.

Die Verwendung von Long Polling der Azure Service Bus Queues umgeht das Problem des ständigen Polling durch die Aufrechterhaltung einer Verbindung mit der Warteschlange, bis eine Nachricht eingeht oder eine Zeitüberschreitung (Timeout) eintrifft. Durch eine maximale Verbindungslänge von 24 Tagen verringert sich die Anzahl der unnötigen Zugriffe auf die Warteschlange auf ein vernachlässigbares Minimum. [13]

Aufgrund der besseren Auswahl an Funktionen und des niedrigeren Preises (mittels Long Polling) wurde die Azure Service Bus Queue als Warteschlangenlösung für dieses Projekt ausgewählt.

5.7. Event Store

Da jeder der drei Bounded Contexts von *CombatZone* CQRS als Architektur einsetzt, ist es sinnvoll, auch in allen drei BCs Event Sourcing für die Persistierung der Daten zu setzen.

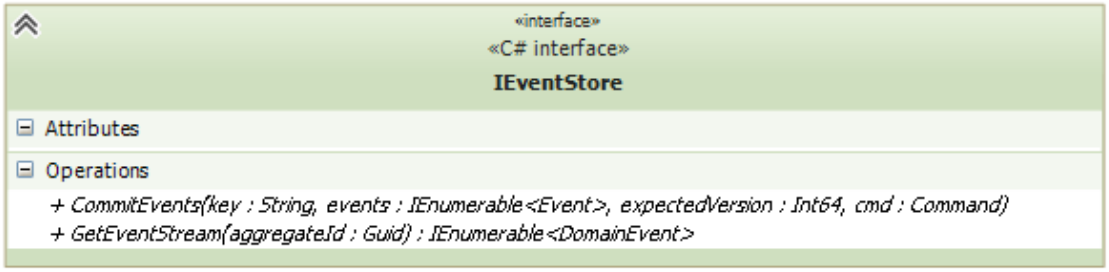


Abbildung 5.11.: UML-Diagramm des Event Store Interface

Die Auswahl an vorhandenen Implementierungen ist überschaubar. Während einige zu neu sind, um deren Verlässlichkeit zu bewerten, scheinen andere nicht weiter gepflegt zu werden. Beispiels-

weise ist der – auch kommerziell vertriebene – Event Store von Greg Young (Namensgeber von CQRS) laut eigener Aussage noch weitgehend undokumentiert und die Fertigstellung von wichtiger Funktionalität noch ausstehend.

Daher wurde entschieden, für dieses Projekt eine minimalistische eigene Implementation zu kreieren. Diese soll die im Kapitel 4 gestellten Anforderungen erfüllen; sie kann aber – sollten sich die Anforderungen ändern – auch gegen eine leistungsfähigere oder weiter-skalierbare Implementation ausgetauscht werden.

5.7.1. Persistierungsoptionen

Als Persistierungsoptionen für den geplanten Event Store bietet Windows Azure mehrere Optionen an, z.B. die SQL Datenbank *SQL Azure*, den NoSQL-Datenspeicher *Table Storage* sowie *BLOB Storage*.

Da die Speicherung der Events keine relationale Datenbank benötigt und auch Table Storage keine sichtbaren Vorteile gegenüber dem simplen BLOB Storage bietet, wurde mit *BLOB Storage* die simpelste Persistierungsoption gewählt.

Eigene Tests zeigten, dass der Windows Azure BLOB Storage eine Zugriffszeit von 8-12 Millisekunden besitzt. Eine simple Hochrechnung ergibt somit einen theoretischen Durchsatz von 83 - 125 Schreibtransaktionen pro Sekunde pro "Tape". Der Quelltext für das Testprogramm, welches für diese Tests verwendet wurde, ist im Anhang A.1 zu finden.

Mittels Schätzung (A.2) wurde erfasst, dass selbst bei voller Ausschöpfung des Skalierbarkeitsziels (s. 4.2.3) nicht mehr als 50-60 Schreib-Operationen erreicht werden sollten. Demnach ist die Leistung von Azure BLOB Storage ausreichend für den geplanten Event Store.

5.7.2. Speicherschema

Die Anforderungen an das Speicherschema können zusammengefasst werden in:

- Performante Lesezugriffe, sowohl von einem Event Stream (Command Handling) als auch von allen gleichzeitig (Projektionen)
- Hohe Toleranz bei Abstürzen
- Ausreichende Möglichkeiten für Backup und Wiederherstellung

Diese Anforderungen decken sich stark mit denen, welche an das *Bitcask* genannte Speichersystem gestellt wurden, dass in [15] erläutert wird. Ein Bitcask-Datenspeicher basiert auf einem Verzeichnis, in dem nachfolgend Datensätze in Dateien abgelegt werden. Zu jedem Zeitpunkt gibt es jeweils nur eine einzige aktive Datei, in die geschrieben werden kann, sowie ebenfalls nur einen einzelnen schreibenden Prozess. Bei einem Absturz wird eine neue Datei angelegt und als aktive Datei verwendet. Dieses Prinzip kann 1:1 auf unseren Blob Speicher übertragen werden, ein Verzeichnis ist hierbei ein Blob-Container und die Dateien sind die jeweiligen Blobs.

Als grobes Vorbild wurde hierbei die Event Store Implementation des "Lokad.CQRS Sample Project" (siehe [10]) verwendet, welche ebenfalls nach diesem Prinzip vorgeht, jedoch an einigen Punkten einen anderen Fokus verfolgt, wie z.B. erweiterte Möglichkeiten zum Streaming der Daten sowie die Unterstützung multipler Persistierungsoptionen.

Speicher-Operationen von Azure's Blob Storage sind stets atomar. Damit die Änderungen der jeweilig aktuellen Transaktion ebenfalls atomar bleiben, muss diese Information in einem zusammenhängenden Block geschrieben werden. Wie solch ein Block aufgebaut sein könnte, ist in Abbil-

dung 5.12 skizziert.

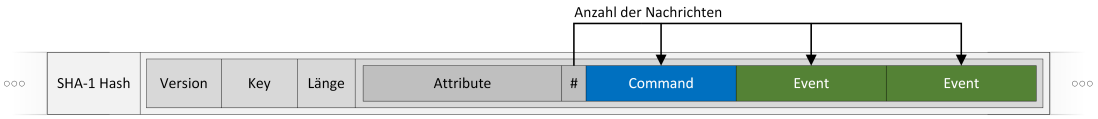


Abbildung 5.12.: Aufbau eines Speicher-Blocks

Die Nachrichten werden hierbei in JavaScript Object Notation (JSON) hinterlegt, da dieses Format bereits für die Nachrichtenübertragung verwendet wird und JSON durch die Beibehaltung der Lesbarkeit für Menschen die Wartung und Entwicklung vereinfachen kann. Durch binäre Serialisierungs-Arten, wie z.B. ProtoBuf, könnte die Serialisierung zwar minimal beschleunigt werden, die kleine Differenz dürfte jedoch – zumindest in diesem Projekt – nicht den Verlust der einfacheren Wartung rechtfertigen [14].

Attribute werden in der beiliegenden Version von *CombatZone* noch nicht verwendet, würden es aber erlauben, Meta-Informationen zu den Events zu persistieren.

- Debug-Informationen: Welche Prozesse waren an der Berechnung beteiligt?
- Performanz-Informationen: Wie viele Millisekunden haben die Berechnungsschritte gedauert? Stichwort: Latenz
- Autorisierungsdaten: Welche Benutzer-Session hat diese Transaktion ausgelöst?

Der “Key“ gibt den jeweiligen Event Stream an. Dies ist üblicherweise die Aggregate-ID, kann jedoch auch für andere Werte verwendet werden, z.B. wenn ein weiterer Event Stream alle von einem externen Bounded Context empfangenen Events für die spätere Verwendung in Projektionen hinterlegen soll.

Um Datenkonsistenz zu gewähren, wird jeder Block zusammen mit einem SHA-1 Hash geschrieben. Dieser wird beim späteren Auslesen mit dem Hash der rekonstruierten Informationen verglichen.

Damit ein späteres Auslesen eines einzelnen Event Streams performant erfolgen kann, werden die Informationen im Arbeitsspeicher zwischengespeichert. Andernfalls würde ein Index oder die Aufsplittung in unterschiedliche Bitcask-Speicher je Aggregate-ID erforderlich werden.

5.7.3. Event Bus

Um neu in den Event Store eingetragene Events an alle interessierte Komponenten verteilen zu können, wird ein Nachrichtensystem benötigt, welches nach dem Publish-Subscribe Messaging Pattern (siehe Abschnitt 2.1.1) aufgebaut ist. Solch ein System wird meist *Event Bus* genannt.

Jeder Bounded Context erhält innerhalb dieses Event Bus einen eigenen *Topic*, in welchem neue Events veröffentlicht werden können.

Mittels dieses Event Bus kann nun der Bounded Context “Billing“ benachrichtigt werden, wenn ein Spieler im Bounded Context “Realm“ eine kostenpflichtige Aktion ausführt. Billing könnte daraufhin entsprechende eigene Prozesse ausführen.

5.7.4. Event Publisher

Für die Veröffentlichung aller neuen Events muss ein Dienst erstellt werden, welcher Neueintragungen im Event Store bemerkt und an den Event Bus weitergibt, damit dieser die neuen Events

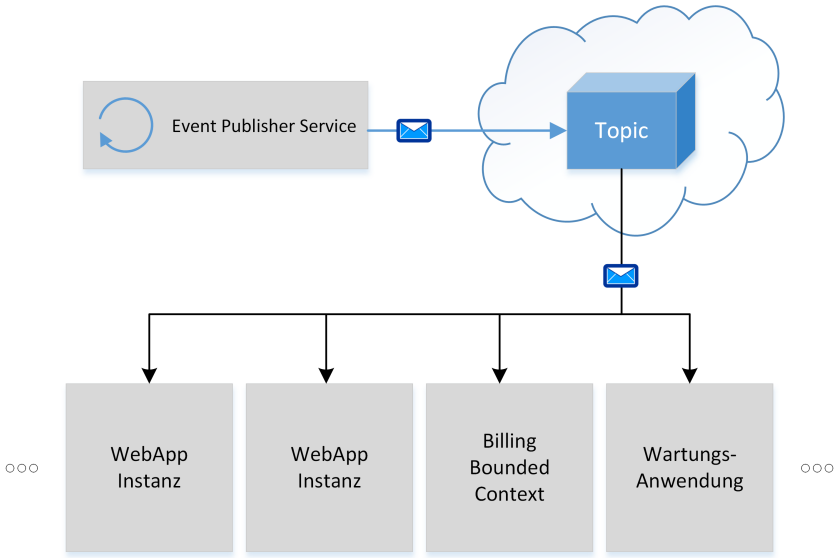


Abbildung 5.13.: Skizze der Event-Verteilung

an alle Abonnenten veröffentlichen kann.

Diese Veröffentlichung geschieht asynchron zum Command Handling. Dies steigert die Performanz, da die Latenz der Veröffentlichung nicht die Bearbeitung von weiteren Commands blockiert. Des Weiteren kann der Dienst nun zu veröffentlichende Events bündeln, wodurch die Performanz seltener unter der Latenz zur Messaging-Infrastruktur leiden muss.

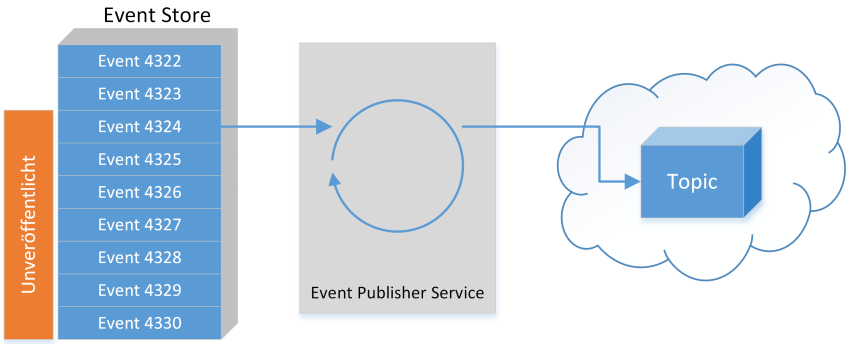


Abbildung 5.14.: Skizze der Event-Veröffentlichung

Es gibt mehrere Probleme, welche während des Veröffentlichungsprozesses auftreten können, wie z.B.:

- Keine Konnektivität zur Messaging-Infrastruktur.
- Worker-Prozess ist unerwartet abgestürzt.
- Worker-Prozess soll beendet werden, ehe alle neuen Events veröffentlicht wurden.
- Virtuelle Maschine wurde von Windows Azure recycled.

Um bei einem solchen vorzeitigen Abbruch nach einem Neustart die Arbeit fortsetzen zu können, muss für den Dienst erkennbar sein, welche Events bereits weitergegeben wurden. Der einfachste Weg, dies zu ermöglichen, ist nach jedem Veröffentlichungsvorgang den letzten Versionsstand des Event Stores zu notieren. Da jede Eintragung in den Event Store diese Version inkrementiert, und

das Interface des Event Stores eine Abfrage ab einer spezifizierten Versionsnummer vorsieht, ist ein performantes Fortsetzen leicht möglich.

Da eine lokale Persistierung dieser Information nicht alle o.g. potenziellen Probleme überstehen würde, wird alternativ die Versionsnummer in einen BLOB-Speicher hinterlegt. Nach dem Start des Dienstes wird zunächst der letzte Stand abgefragt und anschließend die Arbeit anhand der letzten Versionsnummer aufgenommen.

Es ist möglich, dass Events veröffentlicht wurden, ohne dass die Versionsnummer im BLOB-Speicher angepasst wurde. Dies kann z.B. passieren, wenn die Verbindung zum BLOB-Speicher temporär nicht möglich war (was u.a. durch Azure-seitige Dienst-Probleme geschehen kann). Würde der Veröffentlichungsdienst neu gestartet werden (z.B. weil der Worker-Prozess neu gestartet wurde), ehe die Verbindung wieder aufgebaut werden konnte, so würden alle Events seit dem Abbruch der Verbindung neu veröffentlicht werden. Dies ist jedoch unproblematisch, da die Infrastruktur auf "At-Least-Once" ausgerichtet ist und alle Abonnenten empfangene Nachrichten de-duplizieren.

Zusätzlich zu dem letzten Versionsstand wird im BLOB-Speicher auch das Datum und die Uhrzeit der letzten Veröffentlichung gespeichert. Dieser Zeitstempel kann später als Diagnose-Information ausgewertet werden.

5.8. Event Handler

Die konzipierte Infrastruktur von *CombatZone* sieht mehrere Stellen vor, an denen Event Handler benötigt werden.

5.8.1. Projektionen

Projektionen werden verwendet, um anhand der erzeugten Events neue Views zu erstellen bzw. vorhandene zu aktualisieren. Views in *CombatZone* werden in zwei Kategorien unterteilt, *Domain-Views* und *Client-Views*. Diese Unterscheidung ist primär semantisch, technisch gesehen sind beide View-Arten simple Klassen.

Domain-Views werden innerhalb der *Command Processors* erstellt und konsumiert, wohingegen Client-Views auf den späteren Webservern erstellt und auch ausschließlich dort konsumiert werden.

In beiden Fällen werden Views jedoch in einem *View Store*-genannten Datenspeicher hinterlegt. In der dieser Arbeit beiliegenden Version ist dieser View Store anhand einer In-Memory-basierten Speicherstrategie implementiert. Dies erlaubt eine sehr hohe Performanz und niedrige Latenz beim Datenzugriff, es erfordert jedoch dass die Views bei jedem Start der jeweiligen Applikationen neu generiert werden müssen. Dies erfolgt durch die Ausführung aller Projektionen anhand des kompletten Event Streams der jeweiligen Bounded Contexts. Eigene Tests ergaben, dass selbst bei sechsstelligen Mengen an Events dies nur wenige Sekunden dauert. Da dies in keinem Verhältnis zur restlichen Startzeit einer Azure-basierten virtuellen Maschine steht, ist diese Erstellungszeit akzeptabel.

Ein UML-Diagramm, welches die Interfaces für den View Store sowie die darin gespeicherten Views zeigt, ist in Abbildung 5.15 zu sehen. Alle View-Klassen werden das IView-Interface implementieren.

5.8.2. Scheduling

CQRS sieht es prinzipiell vor, dass Commands entweder durch User-gestartete Aktionen oder durch Einbindung von externen Systemen erzeugt werden. Für *CombatZone* ist es jedoch erforder-

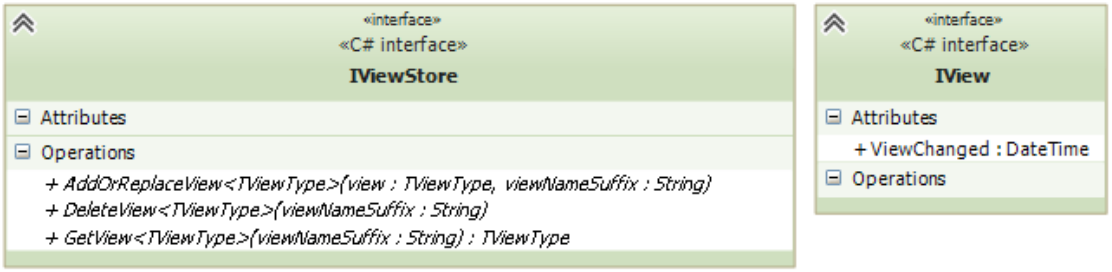


Abbildung 5.15.: UML-Diagramm des View Store Interface

lich, dass das System zu bestimmten Zeitpunkten automatisch Aktionen ausführt.

Das Spielkonzept sieht drei dieser Szenarien vor, nachfolgend *Spielprozesse* genannt:

- Fertigstellung von Raumschiff-Produktionen
- Fertigstellung von zu errichtenden Außenposten
- Erreichen des Zielsektors durch sich bewegende Flotten

Die verwendeten Architektur-Muster gaben keine festen Vorgehensmuster für die Lösung dieses Problems vor, daher musste zwischen den folgenden Varianten abgewogen werden.

Variante: Tick-basierte Commands

Bei der ersten gefundenen Lösungsvariante war es angedacht, allen Aggregates, welche an den o.g. Spielprozessen beteiligt sind, ein "Tick-Command" zu senden. Beim Empfangen solch eines Commands würde der Aggregate überprüfen, ob die eigenen Spielprozesse (bspw. Ankunft bei sich bewegender Flotte) abgeschlossen sind und – wenn dies der Fall ist – ein Event generieren.

Bei der genaueren Betrachtung dieser Lösung stellte sich jedoch heraus, dass diese Variante nur sehr schwer zu skalieren ist. Damit die vielen, bei höheren Benutzerzahlen anfallenden, Tick-Commands vom System zu verkraften wären, dürften Ticks nur in einem größeren Intervall generiert werden, beispielsweise jede Minute ein Tick. Ein Großteil der Ticks würde jedoch zu keinen Events führen, da die jeweiligen Spielprozesse noch nicht fertiggestellt wurden, d.h. werden sehr viele System-Ressourcen verschwendet. Des Weiteren würde ein hohes Tick-Intervall das Spielprinzip von einem Echtzeit-Spiel zu einem Runden-basierten Spiel wandeln.

Variante: Verzögerte Commands

Diese Lösungsvariante sah vor, verzögerte Commands einzusetzen. Am Beispiel der Raumschiff-Produktion würde bereits beim Start des Bau-Prozesses der Command, welcher die Produktion abschließen würde, abgesendet. In den Meta-Informationen der Nachricht würde jedoch eingetragen werden, dass diese erst zu einem späteren Zeitpunkt (dem Datum der Fertigstellung) zugestellt werden soll. Durch die Verwendung externer Funktionalität könnte diese Variante mit geringem Aufwand umgesetzt werden.

Nachteilig an der Benutzung von verzögerten Commands ist jedoch, dass die verwendete Messaging-Infrastruktur dies unterstützen muss. Da dies nicht selbstverständlich ist, erzeugt dies schnell eine Bindung an eine spezifische Infrastruktur. Des Weiteren steigt durch die Versendung solcher Commands die von den Warteschlangen ausgegebene derzeitige Warteschlangenlänge. Dies ist jedoch nachteilig, da dieser Wert oft zur Messung der System-Auslastung verwendet wird, die zwischengespeicherten Commands jedoch diesen Wert verfälschen.

Variante: Scheduler-Dienst

Eine weitere Alternative wäre die Implementierung eines *Scheduler-Dienstes*. Dieser würde als Hintergrund-Prozess des “Realm“-BC laufen und in einem bestimmten Intervall die derzeit geplanten Spielprozesse auf ihre Fertigstellung prüfen. Damit solch eine Prüfung stattfinden kann, muss der Scheduler-Dienst jedoch Zugriff auf die derzeit laufenden Spielprozesse haben.

Diese Informationen werden in Domain Views abgelegt und von Projektionen auf dem neuesten Stand gehalten. Wie ein Eintrag für einen Spielprozess in einem solchen Domain View aussehen kann, ist am Beispiel der Raumschiff-Produktion in Abbildung 5.16 dargestellt.

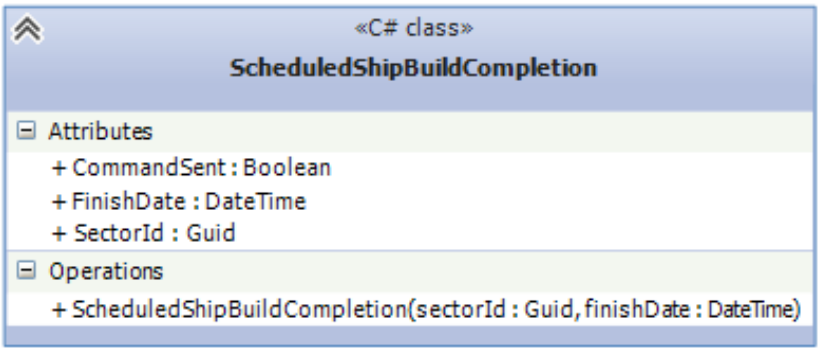


Abbildung 5.16.: Scheduling-Information für Raumschiff-Produktion

Das grobe Schema solch eines Scheduler-Dienstes ist in Abbildung 5.17 skizziert.

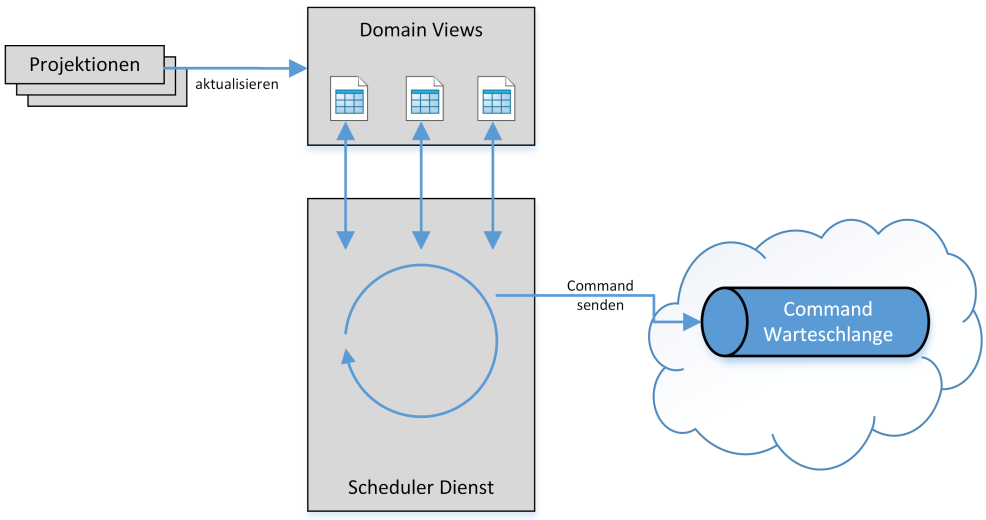


Abbildung 5.17.: Skizze vom Scheduler-Dienst

Nach jedem erfolgreichen Versenden von Commands wird in den Domain Views notiert, dass der jeweilige Spielprozess bereits bearbeitet wurde. Dies verhindert, dass bei jedem Durchlauf – der je nach Konfiguration mehrfach innerhalb einer Sekunde durchgeführt wird – ein erneutes Command versendet wird. Da bei dem Start eines BCs die Domain Views neu erstellt werden, würde hier die Information, dass ein Command bereits versendet wurde, verloren gehen. Dies betrifft jedoch nur einen winzigen Teil der jeweils laufenden Spielprozesse und führt durch die implementierte Idempotenz zu keinerlei Problemen.

Auswertung

Aus den gefundenen drei Varianten erscheint der Scheduler-Dienst als die effizienteste. Sie benötigt keine Änderung des Spiel-Prinzips, ist nicht bindend an spezifische Infrastruktur-Funktionen und erschwert nicht die Wartung.

6. Implementierung

Im Zuge dieses Kapitels sollen einige ausgewählte Themen, welche sich während der Implementierung von *CombatZone* ergaben, näher betrachtet werden. Dies beinhaltet Probleme, deren Lösung zusätzlichen Aufwand notwendig machten, sowie eine genauere Ausführung über die Funktionen des zusätzlich entwickelten Wartungsprogramms *CZ.Manager*.

Zum Zeitpunkt der Abgabe dieser Arbeit ist *CombatZone* in einer prototypischen Version erreichbar unter http://www.combatzone-game.de. Ob das Spiel auch darüber hinaus öffentlich zugreifbar bleibt, ist ungewiss.

6.1. Sektor-Koordinaten

Der initiale Plan, Koordinaten von Sektoren mit einer üblichen Zahl wie 0 oder 1 starten zu lassen, stellte sich schnell als problematisch heraus. Da das Universum im Verlauf seiner Lebenszeit wachsen kann, würde es zusätzliche Komplexität bedeuten, die Konsistenz der Koordinaten bei deren Veränderung zu bewahren. Daher wurde das Koordinatensystem während der Entwicklung dahingehend umgestellt, dass die Koordinaten der Anfangs-Sektoren mittig im Koordinatensystem angeordnet sind (siehe Abbildung 6.1). Hierdurch können bei der späteren Ergänzung von Sektoren diese Lücken aufgefüllt und somit das Koordinatensystem komplettiert werden.

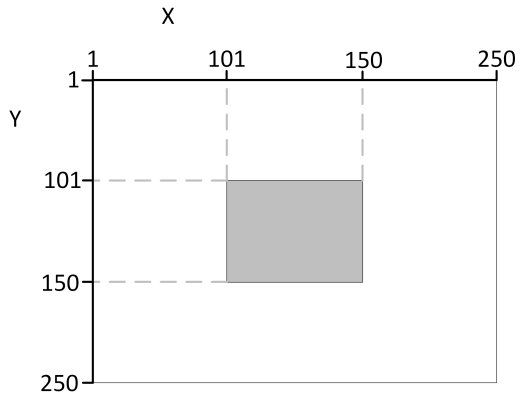


Abbildung 6.1.: Koordinatensystem

6.2. User Interface

Das User Interface eines Spiels ist ein essenzielles Element, welches maßgeblich über die User Experience entscheidet. In den folgenden Abschnitten wird nun dargestellt, wie das User Interface eingeteilt ist und welche Aspekte besondere Aufmerksamkeit während der Entwicklung erforderten.

6.2.1. Unterteilung

Das User Interface wurde bereits am Anfang der Entwicklung skizziert, um einen groben Überblick über den Aufbau selbiger zu erhalten. Diese Skizze ist in Abbildung 6.2 dargestellt. Das Interface

ist in drei Zonen unterteilt, welche nachfolgend genauer beleuchtet werden.

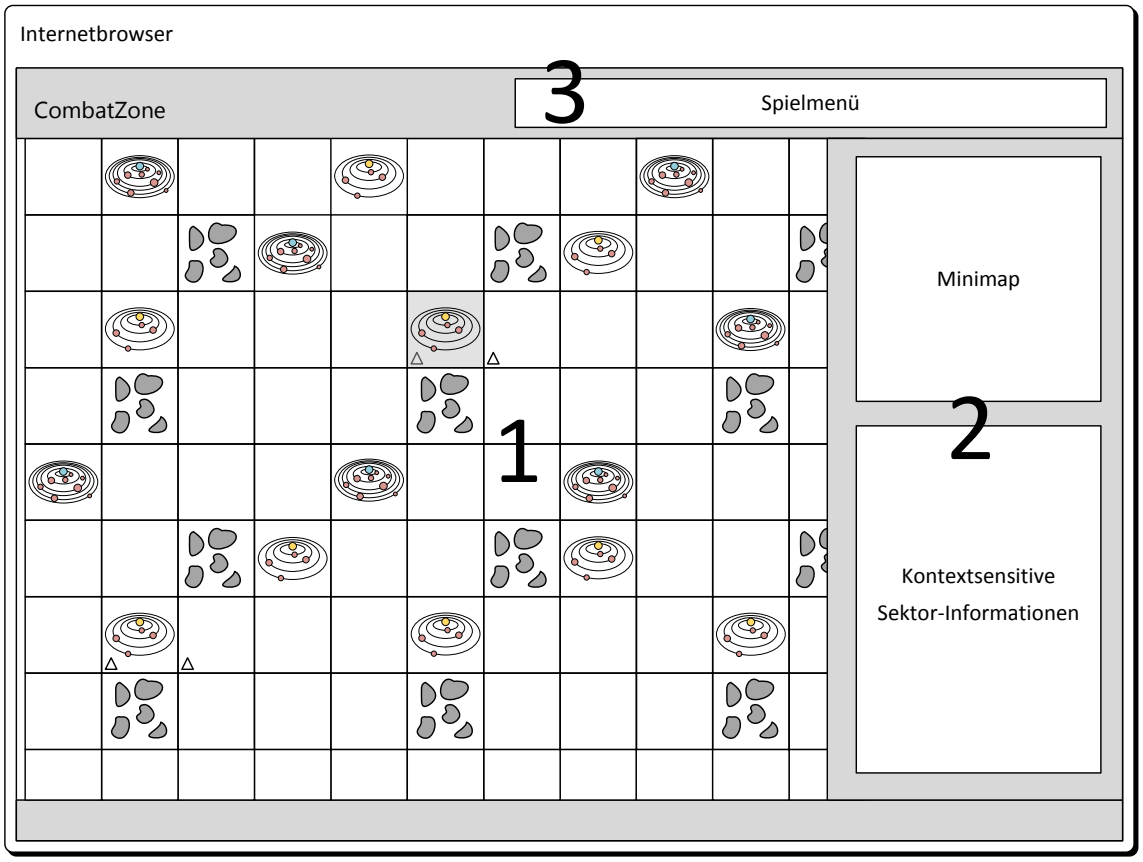


Abbildung 6.2.: Skizzierung des User Interface

Universumskarte (Zone 1)

Das für den Benutzer wichtigste Interaktionselement ist die Universumskarte (Zone 1). Diese erlaubt es ...

- ... sich innerhalb des derzeitigen Universums zu bewegen,
- ... die eigenen und gegnerischen Sektoren zu sehen,
- ... die eigenen Flotten zu sehen und zu bewegen,
- ... die taktische Vorgehensweise zu planen.

Die Interaktion mit diesem Element kann entweder über die Pfeiltasten der Tastatur als auch über eine "Zieh-Geste" mittels der Maus erfolgen. Da diese Geste bereits seit langer Zeit in vielen weit verbreiteten Geräten und Oberflächen von Apple, Google und Microsoft verwendet wird, ist davon auszugehen, dass dies vom späteren Benutzer als sehr intuitiv empfunden wird.

Rechte Leiste (Zone 2)

Der rechte Bereich (Zone 2) beherbergt sowohl die Minimap, als auch kontextsensitive Sektor-Informationen zum derzeit auf der Universumskarte ausgewählten Sektor.

Die Minimap bietet stets einen Überblick über das gesamte Universum. Man kann hierüber die Größe des eigenen Gebiets einschätzen, die eigene Position feststellen sowie schnell die Universumskarte bewegen.

Obere Leiste (Zone 3)

Die obere Leiste (Zone 3) beherbergt alle Bedienelemente und Ausgaben, welche nicht kontextsensitiv zur aktuellen Universums-Ansicht sind. Beispielsweise wird hier die aktuelle Punktezahl des Spielers im aktuellen Universum angezeigt. Des Weiteren befinden sich hier Bedienelemente, um das aktuelle Universum zu wechseln oder sich abzumelden.

Umsetzung

Wie das User Interface nach der Umsetzung der Skizze aussieht, ist anhand eines Screenshots im Anhang A.4 ersichtlich. In den folgenden Abschnitten wird nun auf einige spezifische Bereiche der Oberfläche eingegangen.

6.2.2. Universumskarte

Während der Implementierung der in 6.2.1 skizzierten Universumskarte musste abgewogen werden, ob dies via üblicher HTML-Elemente oder mittels eines Canvas-Element erfolgen soll.

Es stellte sich jedoch schnell heraus, dass die Anzahl der HTML-Elemente, welche für die vielen Sektoren eines Spiel-Universums notwendig wären, leicht zu einem Performanzproblem führen können. Mittels eines frühen Prototyps zeigte sich bereits bei einem Universum mit geringerer Größe als die festgelegte Startgröße, dass die CPU-Auslastung von Google Chrome beim Betrachten auf ein zu hohes Maß anstieg, um eine angenehme User Experience zu bieten.

Aus diesem Grund, sowie den Vorteilen der besseren Gestaltungsoptionen für Overlay-Elemente, wurde entschieden, ein Canvas-Element für das dynamische Rendering der Universumskarte zu verwenden.

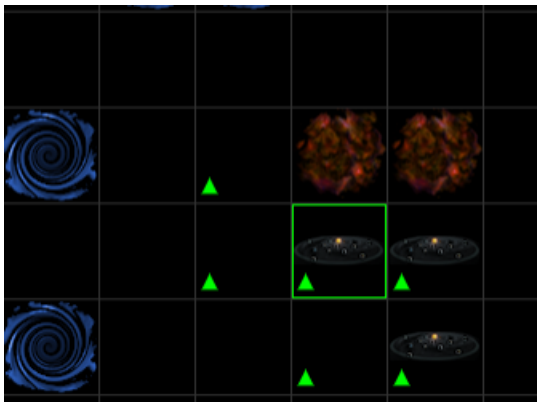


Abbildung 6.3.: Ausschnitt der Universumskarte

6.2.3. Flottenbewegung

Da die – für die Bewegung oft übliche – Zieh-Mausgeste bereits für die Bewegung innerhalb der Universumskarte verwendet wurde, stellte sich bei der Entwicklung die Frage, wie es am elegantesten zu lösen wäre eine Flotte zu verschicken.

Üblicherweise wird solch eine Aktion durch das Ziehen der Maus bei gedrückter Maustaste implementiert. Da solch eine Maus-Bewegung jedoch schon für die Bewegung des Sichtfeldes auf der Karte verwendet wurde, ergab dies zunächst einen Konflikt. Dieser wurde gelöst, in dem das Verschicken von Flotten über eben diese Mausgeste nur möglich ist, wenn ein Sektor zuvor explizit angewählt wurde. Zwar setzt dies einen zusätzlichen Klick seitens des Benutzers voraus, dies war

jedoch die sinnvollste Variante welche keinen zusätzlichen UI-Aspekt – wie z.B. weitere Menüs – erforderte.

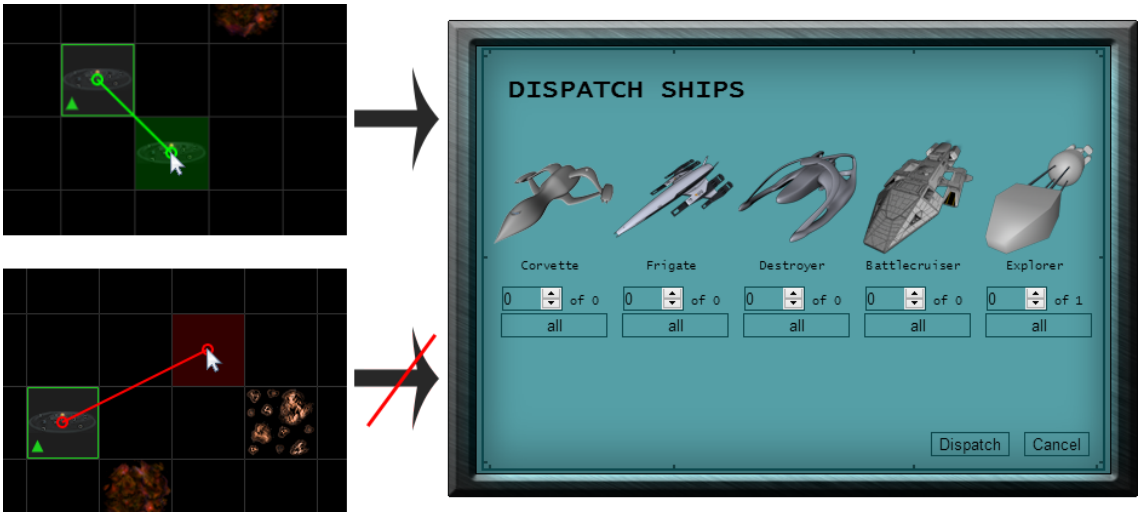


Abbildung 6.4.: Flottenbewegung

6.3. Initialisierung der Web-Applikation

Windows Azure bietet für verteilte Web-Applikationen Möglichkeiten an, um Initialisierungen durchzuführen, ehe eine Instanz der Applikation in den Load Balancer eingetragen wird, und somit Anfragen erhält. Dies sollte verwendet werden, um bei einem Start der Anwendung den aktuellen Stand des Event Stores abzufragen und anhand der Projektionen die Arbeitsspeicher-basierten Views zu erstellen. Zwar dauert diese Aktion nur eine kurze Zeit, jedoch kann die Applikation innerhalb dieser Zeitspanne keine Anfragen korrekt beantworten.

Leider stellte sich während der Entwicklung heraus, dass die Web-Applikation und dessen Initialisierungssystem – obwohl es sich um dieselbe Anwendung handelt – von Azure nicht im selben Prozess gestartet werden. Hierdurch teilen sich diese zwei Bereiche keinen Speicherraum und die Initialisierung kann nicht hinreichend erfolgen.

Um dieses Problem zu umgehen, wurde die Initialisierung in die Start-Methode der ASP.NET Applikation verlegt. Leider ist zu diesem Zeitpunkt die Applikation bereits in Azure's Load Balancer eingetragen, weshalb für wenige Sekunden Anfragen erhalten werden könnten, welche nicht korrekt beantwortet werden können.

6.4. Umgang mit Zeit

Bei der Entwicklung ergaben sich zwei Problemfälle, welche mit dem Server- bzw. Client-seitigen Umgang mit der Zeit assoziiert waren.

6.4.1. Server

Beide Server-Anwendungen benötigen einen verlässlichen Zugriff auf die jeweils aktuelle Uhrzeit, u.a. für die Berechnung der Fertigstellungszeiten der Spielprozesse. Leider ist die Windows-Systemzeit innerhalb von Azure-gehosteten virtuellen Maschinen oft sehr unzuverlässig, je nachdem wie stark die Auslastung des Host-Servers ist. Die technischen Hintergründe für dieses Problem

sind unter [3] nachzulesen. Dieses Problem tritt insbesondere bei den günstigeren/kleineren virtuellen Maschinen auf, wie sie während der Entwicklung verwendet wurden.

Der Windows-Dienst, der für die Zeit-Synchronisierung zuständig ist, sieht es jedoch standardmäßig nur vor, einmal in der Woche die Zeit zu synchronisieren. Dies ist zur Ausgleichung bei den starken Abweichungen jedoch zu wenig. Um dieses Problem zu umgehen, wird während dem Deployment beider Applikationen ein Skript ausgeführt, welches den Zeitplan der Zeit-Synchronisierung innerhalb der Windows Registry dahingehend anpasst, dass diese Synchronisierung alle 10 Minuten stattfindet. [2]

6.4.2. Clients

Durch die Verwendung von Client-seitiger Javascript-Spiellogik ist es unumgänglich, auch in diesen Bereichen der Applikation auf die aktuelle Zeit zuzugreifen. Da die Systemzeit der zugreifenden Computer jedoch nicht genügend Zuverlässigkeit besitzt, musste ein alternativer Weg gefunden werden.

Aus diesem Grund stellt die *CombatZone*-Spiel-Oberfläche bei deren Betreten eine Anfrage nach der aktuellen Zeit an den Server. Diese wird mit der lokalen Client-Zeit verglichen und die Differenz gespeichert. Bei allen nachfolgenden Client-seitigen Zeit-berechnungen wird dieser Wert einbezogen, sodass effektiv mit der Server-Zeit gearbeitet wird, ohne diesen in die Berechnung direkt mit einzubeziehen.

6.5. Wartung und Auditing

Damit ein verteiltes System ordnungsgemäß und zuverlässig funktionieren kann, bedarf es Hilfsmittel, welche einem Administrator bei der Diagnose und einem Entwickler bei der Fehlerbehebung unterstützen können. Da CQRS-Systeme meist sehr individuell konstruiert sind, gibt es keine allgemein anwendbare Wartungsanwendungen, welche einem Administrator ausreichend Unterstützung bieten kann. Für diesen Zweck wurde die *CZ.Manager* Wartungsanwendung während der Entwicklung von *CombatZone* mitentwickelt.

Auditing beschreibt den Vorgang, den vorhandenen Datenbestand zu überprüfen. Diese Überprüfung kann unterschiedlicher Natur sein, z.B. auf Korrektheit, Nachverfolgung von Benutzeraktionen oder zum Abgleichen mit einem weiteren Datenbestand (bspw. Papierakten).

6.5.1. Kontext-Auswahl

Sowohl in Entwicklungs- als auch in Debug-Szenarien kommt es oft vor, dass die Software in mehreren Instanzen gleichzeitig läuft. Bereits beim Start vom *CZ.Manager* wird der Benutzer aufgefordert, zwischen dem Kontext “Dev“ oder “Production zu wählen (siehe Abbildung 6.5). Je nachdem welche Auswahl hier erfolgt, verbindet die Wartungsanwendung sich mit dem lokalen Emulator oder direkt mit Windows Azure.

In der aktuellen Version sind die Werte hinter den beiden Kontexten fest definiert. Bei einer Weiterentwicklung wäre es sinnvoll, diese Einstellungen z.B. in eine XML-Datei auszugliedern.

6.5.2. Darstellung von Event Streams (Auditing)

Mittels eines simplen Klicks lassen sich alle Commands und Events eines Bounded Contexts ausgeben.

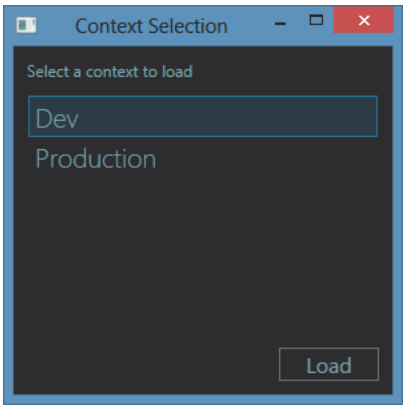


Abbildung 6.5.: Kontext-Auswahl

Auf Wunsch des Benutzers kann *CZ.Manager* sich auch im Event Bus eintragen, um die dargestellten Informationen aktuell zu halten. Neu hinzukommende Commands und Events werden dann dynamisch an die Listendarstellung angehangen.

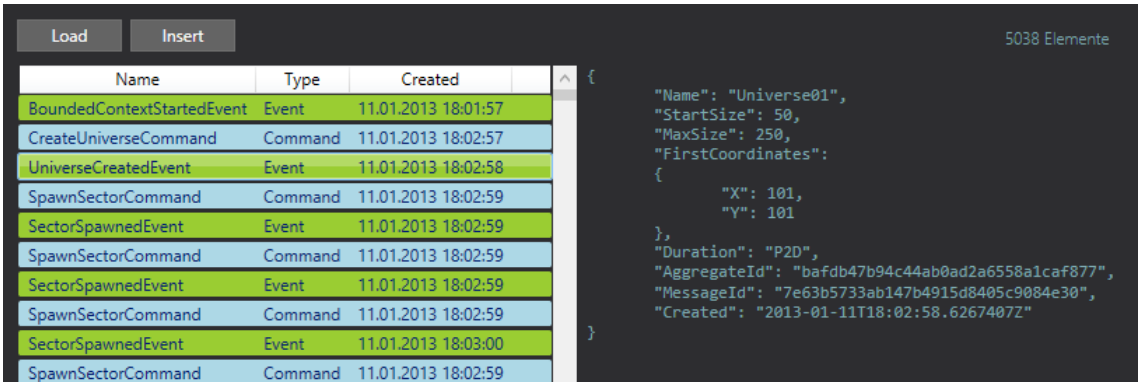


Abbildung 6.6.: Anzeige eines Event Streams

Nach einem Klick auf einen Eintrag der Liste wird das ausgewählte Command oder Event in JSON-/JSV-Format auf der rechten Seite des Fensters dargestellt. Dies bietet dem Benutzer die Möglichkeit, Werte auf deren Korrektheit zu überprüfen oder Werte nachzuschlagen.

Diese Funktion kann sowohl aus Wartungsgründen als auch für Prüfungen (z.B. seitens des Spielleiters) essenziell sein, denn hierüber sind alle Vorgänge des Systems ersichtlich.

6.5.3. Management der Warteschlangen

Ein erster Schritt bei der Diagnose und Wartung von *CombatZone* ist oftmals die Betrachtung der gegenwärtigen Warteschlangenlängen. Sind diese ungewöhnlich hoch, so ist dies ein Zeichen dafür, dass ein Prozess einer Worker-Instanz nicht ordnungsgemäß funktioniert. Handelt es sich jedoch um eine Poison-Warteschlange, so obliegt es dem Administrator oder Entwickler, sich dessen Inhalt anzuschauen, den Grund für die fehlgeschlagene Bearbeitung zu evaluieren und nach Lösung des Problems diese Nachricht wieder in die korrekte Warteschlange zu verschieben.

Es ist darüber hinaus auch über diese Oberfläche möglich, manuell Commands in das System einzuschleusen. Dies kann sinnvoll sein, wenn die eigentliche Benutzeroberfläche umgangen werden muss, eine Funktionalität mit Absicht nicht direkt zugänglich ist (Wartungsoperationen) oder ein spezifischer Test durchgeführt werden soll.

Die Benutzeroberfläche hierzu ist in Abbildung 6.7 dargestellt.

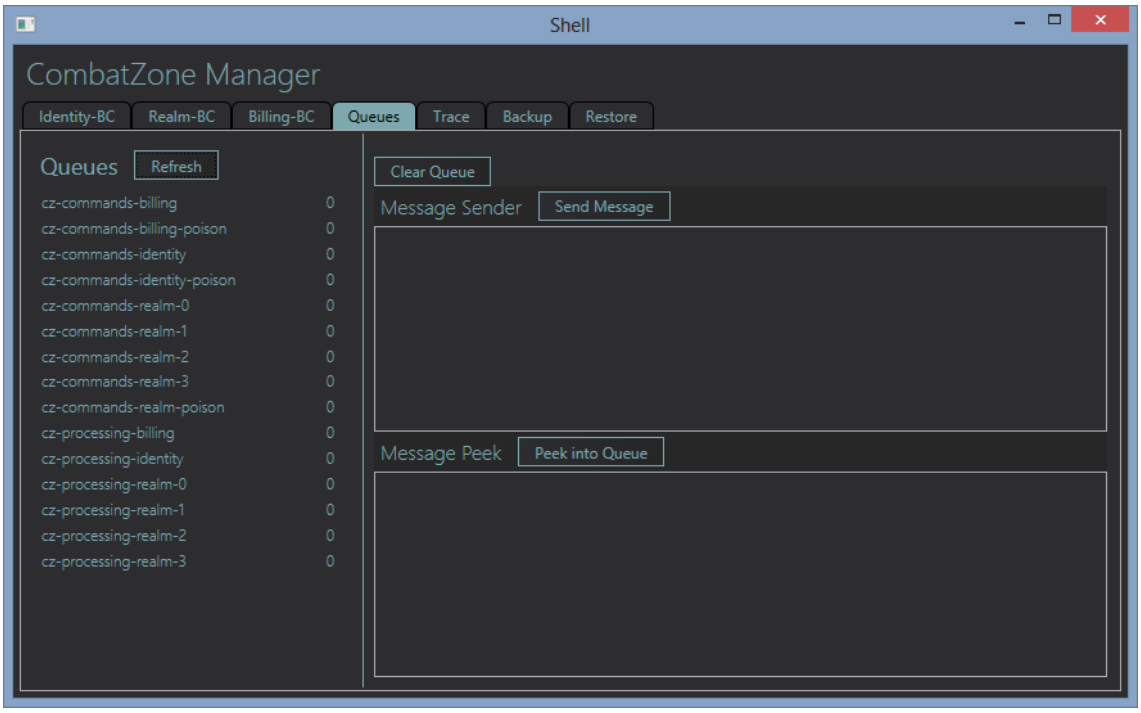


Abbildung 6.7.: Dialog zum verwalten der Warteschlangen

6.5.4. Datensicherung

Nach Auswahl des zu sichernden Bounded Context kann *CZ.Manager* beginnen, den jeweiligen Event Store auf die lokale Festplatte zu replizieren. Dies geschieht durch den Download der Blobs, deren Konsistenz dabei erhalten bleibt, da ein potenziell schreibender Prozess nur Daten anhängen kann.

Wahlweise kann diese Sicherung auch inkrementell erfolgen, um die benötigte Zeit oder das zu übertragene Datenvolumen zu verringern.

Über einen weiteren Bereich von *CZ.Manager* können die Sicherungen wieder eingespielt werden.

7. Test

7.1. Test der funktionalen Anforderungen

In diesem Abschnitt soll nun gezeigt werden, wie die in Abschnitt 4.1 definierten funktionalen Anforderungen auf deren korrekte Umsetzung hin getestet wurden.

7.1.1. Unit-Tests

Da durch die Verwendung von DDD die Kernfunktionalität innerhalb der Domainlogik-Klassen gebündelt ist, erschien es sinnvoll, die Unit Tests auf diesen Bereich der Software zu konzentrieren. Die Unit Tests sollen hierbei dazu dienen, die Funktionalität der Domänenlogik-Klassen auf deren Korrektheit zu überprüfen.

Da Aggregates ausschließlich den System-Zustand mittels der von ihnen erzeugten Events verändern können, war es die Primäraufgabe der Unit Tests, die Werte der erzeugten Events mit erwarteten Werten zu vergleichen.

Insgesamt wurden 24 Unit Tests für diesen Zweck implementiert. Eine Liste ist im Anhang unter dem Punkt A.5 zu finden.

7.1.2. Testlauf

Am 25.01.2013 wurde im Zeitraum von 9 Uhr morgens bis 13 Uhr ein Testlauf durchgeführt. Der Testlauf wurde von sieben Testern durchgeführt und sollte sowohl die funktionalen Anforderungen testen, als auch genügend Last auf Server-Seite erzeugen, um mittels dieser Werte die Last mit noch mehr Benutzern besser abschätzen zu können.

7.2. Test der nicht-funktionalen Anforderungen

Nach den funktionalen Anforderungen werden nun die nicht-funktionalen Anforderungen, welche in Abschnitt 4.2 erläutert wurden, auf ausreichende Beachtung geprüft.

7.2.1. Verarbeitung von Commands

Zum Test der Verarbeitungsgeschwindigkeit wurde wiederholt ein neues Universum erstellt, wobei mit der Anzahl der Command Processors experimentiert wurde. Die Erstellung eines Universums ist in Version 1.0 von *CombatZone* die teuerste Operation, welche ausgeführt werden kann. Da jeder der 2500 zu erstellenden Sektoren ein Aggregate ist, muss für jeden Sektor (sowie das Universum selbst) ein Command versendet werden, was summiert 2501 Commands (sowie 2501 Events) ergibt.

Dieser Test wurde mit 1, 2, 4 und 8 Command Processors im “Realm“ BC durchgeführt, die Ergebnisse sind in Abbildung 7.1 ersichtlich. Sie werden in Abschnitt 8.1 des nächsten Kapitels näher betrachtet.

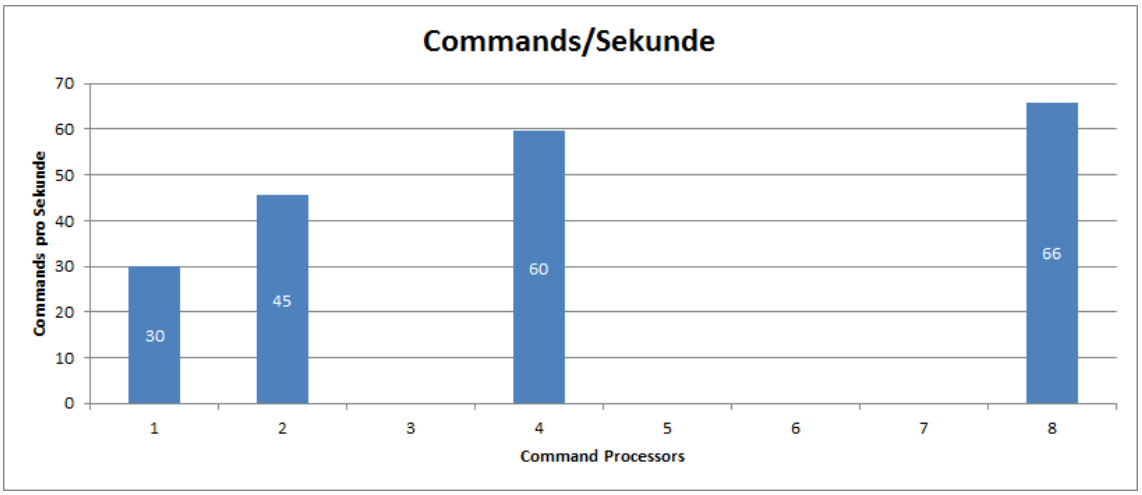


Abbildung 7.1.: Testergebnisse der Command-Verarbeitung

7.2.2. Erkennung ungültiger Commands

Während der Entwicklung kam es häufig vor, dass durch noch nicht implementierte Funktionen oder nicht korrigierte Fehler Commands als ungültig erkannt wurden. Ein explizites Testen dieser Erkennung ungültiger Commands war daher nicht notwendig.

7.2.3. Event Store

Während exzessivem Testen zeigte sich, dass die Performanz beim Lesen vom Event Store dramatisch abflachte, wenn viele gespeicherte Events bei einem Neustart der Applikation eingelesen werden mussten. Bei 200.000 gespeicherten Nachrichten ergab dies Ladezeiten von etwa 6 Minuten (bei Ausführung unter Windows Azure).

Nach intensivem Profiling und Optimierungen der Performanz-Engpässe konnte dieser Wert auf 14-20 Sekunden gesenkt werden. Bei der Zeit, welche Azure zur Bereitstellung einer virtuellen Maschine und Windows zum Hochfahren benötigt, sind diese 14-20 Sekunden vernachlässigbar.

7.2.4. Datensicherheit

Weil für die Speicherung der Anwendungsdaten eine Eigenentwicklung statt eines etablierten Speichersystems angewandt wurde, bedurfte es einer gesonderten Überprüfung der Datensicherheit.

Korruptionserkennung

In 5.7.2 wurde erwähnt, dass die vom Event Store gespeicherten Daten mit einem SHA-1 Hash persistiert werden. Während eines späteren Auslesens muss, damit eine Datenkorruption bemerkt werden kann, der SHA-1 Hash der gelesenen Daten mit dem hinterlegten Hash verglichen werden.

Um dies zu überprüfen, wurde mittels eines kleinen Programms ein Byte im BLOB-Speicher verändert und anschließend *CombatZone* gestartet. Beim Auslesen der BLOBs wurde korrekterweise eine Exception vom Event Store geworfen und die Applikation somit gestoppt.

Zwar ist in diesem Falle gezwungenermaßen manueller Administrator-Aufwand vonnöten, jedoch ist dies präferabel im Vergleich mit stiller Datenkorruption.

Der Quellcode für dieses Programm ist im Anhang A.3 zu finden.

Datensicherung und -Wiederherstellung

Um die Datensicherung und -Wiederherstellung zu testen, wurde zunächst via *CZ.Manager* der Event Store Inhalt des BCs "Realm" auf die lokale Festplatte gesichert. Daraufhin wurde manuell der Inhalt des BLOB-Containers geleert, um einen kompletten Datenverlust zu simulieren. Anschließend wurde – erneut mithilfe vom *CZ.Manager* – die Sicherung in den BLOB-Container wiederhergestellt.

Nach dem Abschluss der Wiederherstellung wurde die Anzahl der wiederhergestellten BLOBs, deren Größe sowie – stichprobenhaft – deren Inhalte verglichen. Da sich hierbei keine Unterschiede ergaben, wurden daraufhin die beiden Anwendungen von *CombatZone* gestartet, welche weiterhin fehlerfrei funktionierten.

Versendung von Warnmeldungen

Wie im Abschnitt 7.2.2, so musste auch das Versenden von Warnmeldungen bei Fehlschlagen einer Command-Verarbeitung nicht explizit auf Funktionalität geprüft werden, da solche Warnmeldungen während der Entwicklung hinreichend zur Anwendung kamen.

8. Ergebnisbewertung

8.1. Skalierbarkeit und Performanz

Während der Testdurchläufe in Abschnitt 7.2.1 zeigte sich, dass ab einer Command Processor Anzahl von vier eine weitere Erhöhung nur geringe Server-seitige Geschwindigkeitsvorteile bot. Jeder Command Processor verringert die Auswirkung, welche die Latenz jeder einzelnen Nachricht auf die Gesamtverarbeitungsgeschwindigkeit hat. Bei einer Verwendung von vier oder mehr Command Processors begrenzt jedoch die Latenz zum BLOB-Speicher, wie sie der Event Store einsetzt. Möglichkeiten, diese Begrenzung zu umgehen, werden in Abschnitt 10.3 gezeigt, jedoch ist die Performanz ausreichend, um das in Abschnitt 4.2.3 deklarierte Skalierbarkeitsziel zu erreichen.

Ein genauer Test der Client-seitigen Performance würde den Rahmen dieser Arbeit überschreiten, während des im Abschnitt 7.1.2 erwähnten Testlaufs sind jedoch keine Performanz-Probleme des User Interface aufgefallen.

8.2. Verfügbarkeit

Beim Betrachten der Verfügbarkeit ist es sinnvoll, zwischen den beiden Anwendungen wie folgt zu differenzieren:

8.2.1. Web-Applikation

Für die Web-Applikation von *CombatZone* ist es sehr einfach, eine hohe Verfügbarkeit zu ermöglichen. Da diese zustandslos implementiert ist, sowie direkt beim Start ihre eigene lokale Kopie des Datenbestandes erzeugt, lässt sich die Verfügbarkeit durch ein simples Erhöhen der Instanz-Anzahl verbessern.

Sollte die Funktionstüchtigkeit einer Instanz durch ein Problem seitens Windows Azure beeinträchtigt oder unterbrochen werden (bspw. durch Hardware-Ausfall), so wird automatisch eine neue Instanz von Azure bereitgestellt und gestartet. Sind zu diesem Zeitpunkt mehrere Instanzen aktiv, so dürfte ein Benutzer keine Auswirkungen von diesem Ausfall bemerken, da jedwede Anfragen durch Azure's Lastverteilung automatisch auf die aktiven Instanzen verteilt werden.

8.2.2. Worker-Anwendung

Wie ein Ansatz für eine hochverfügbare Variante aufgebaut sein könnte, wird im Abschnitt 10.4 beschrieben.

Der Worker ist mittels des in 5.4.1 gewählten Partitionierungsschematas nicht hochverfügbar konzipiert, wodurch jeder Bounded Context ein Single Point of Failure ist. Bei einem Ausfall der gesamten Instanz dauert es wenige Minuten, bis diese wieder erreichbar ist. Dies geschieht jedoch automatisch (durch Windows Azure) und würde die Bearbeitung der in der Zwischenzeit angefallenen Commands sofort nachholen.

Eine Möglichkeit, auch für die Worker-Anwendung eine Hochverfügbarkeit zu ermöglichen, hätte den Rahmen dieser Arbeit überstiegen. Der Abschnitt 10.4 im Kapitel *Ausblick* bietet einen kurzen Überblick, wie dies umgesetzt werden könnte.

8.2.3. Fazit

Letztendlich wurden die an die Verfügbarkeit gestellten Anforderungen erreicht, da die Web-Anwendung hochverfügbar gemacht werden kann und ein temporärer Ausfall einer Worker-Instanz (wie z.B. durch die Einspielung eines Updates) keinen großen Einfluss auf die User Experience besitzt.

8.3. Datensicherheit

Wie in 5.6.2 erläutert, werden fehlgeschlagene Commands nicht gelöscht, sondern einem Administrator zur Diagnose präsentiert, welcher nach Lösung des Problems den Vorgang wiederholen lassen kann. In RPC-basierten Systemen gehen fehlgeschlagene Operationen oftmals verloren. Durch diese zusätzliche Absicherung der Commands – welche die Absicht des Benutzers verkörpern – bietet *CombatZone* eine hohe Datensicherheit für Benutzeraktionen.

Die gespeicherten Event Streams werden durch das Hinterlegen von SHA-1 Hashs auf deren Datenkonsistenz prüfbar gespeichert. Zwar schützt dies nicht gegen Datenkorruption, es erlaubt jedoch – wie in 7.2.4 getestet – das Erkennen selbiger. Durch diese Erkennung alarmiert, können Administratoren eine Reparatur bzw. das Einspielen einer Sicherung vornehmen.

Um solche Backups anfertigen zu können, bietet das Wartungsprogramm eine Oberfläche an, um den aktuellen Stand eines Event Stores auf die lokale Festplatte zu replizieren. Dies lässt sich mittels der Ausführung über die Kommandozeile automatisieren und kann entweder als Komplett-Backup oder inkrementelles Backup geschehen.

Der technische Hintergrund für solche Backups ist geradezu trivial, denn für ein komplettes Backup müssen einfach nur alle BLOBs innerhalb des BLOB-Containers des jeweiligen Bounded Context heruntergeladen und gespeichert werden. Ein inkrementelles Backup hingegen lädt die letzte Backup-Datei des vorherigen Backups neu herunter (da sich diese verändert haben könnte) und ergänzt alle danach erstellten Dateien.

Wünschenswert wäre es – um den Umgang mit Sicherungen zu vereinfachen und die Zuverlässigkeit zu steigern –, wenn die Dateien in ein Dateiarchiv (z.B. ZIP-basiert) und erneut mit einem Hash gesichert werden würden. Dies könnte leicht nachträglich hinzugefügt werden.

Zusammenfassend wurden die Anforderungen an die Datensicherheit, welche in Abschnitt 4.2.5 definiert wurden, komplett erfüllt.

Mit weiterem Aufwand könnte das Intervall-basierte Backup auch durch ein Live-Backup ergänzt werden, welches neue Events über die Publish-Subscribe-Nachrichteninfrastruktur erhält und diese sofort in das letzte Backup ergänzt. Eine weitere Möglichkeit zur Erhöhung der Datensicherheit wird im Abschnitt 10.4 skizziert.

8.4. Wartung und Überwachung

Für die Wartung von *CombatZone* stehen einem Administrator bzw. Entwickler das Azure Web Portal sowie der CZ.Manager zur Verfügung. Der CZ.Manager erlaubt es, die Event Streams und Views eines BCs auszugeben, Beobachtungen und Modifizierungen an den Warteschlangen und deren Inhalten durchzuführen, Server-Logs zu betrachten und Event Store Sicherungen zu veranlassen.

Hiermit deckt dieses Tool einen Großteil der anfallenden Wartungsarbeiten ab. Dies ist auch notwendig, da – anders als in SQL-zentrierten Anwendungen – nur sehr wenig Hilfsmittel für Anwen-

dungen existieren, welche auf CQRS und Event Sourcing basieren.

Für das manuelle Erstellen oder Modifizieren von Nachrichten innerhalb der Warteschlangen sind JSON-Kenntnisse erforderlich. Diese sind jedoch von einer für die Wartung beauftragen Person zu erwarten, wie auch SQL-Kenntnisse von einem Datenbank-Administrator vorausgesetzt werden.

Als zusätzliches Hilfsmittel erhalten im Server eingetragene E-Mail-Adressen automatisch eine Warnung (inkl. Details), wenn eine Nachricht in die Poison Queue (aus 5.6.2) verschoben und somit ein manueller Eingriff vonnöten ist.

Die aktuelle Auslastung des Systems ist einfach über die Länge der jeweiligen Warteschlangen erkennbar.

Die in Abschnitt 4.2.6 definierten Anforderungen sind somit hinreichend erfüllt.

9. Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung eines Internetspiels anhand Cloud-Technologien, Messaging sowie die Verwendung einer interaktiven und HTML5-basierenden Benutzeroberfläche. Die Infrastruktur sollte hierbei mittels geringen Veränderungen am Gesamt-System an höhere Bedürfnisse angepasst werden können.

Zum Anfang dieser Arbeit wurde hierfür ein Spielkonzept für ein Internetspiel namens *CombatZone* dargestellt, welches im Zuge dieser Arbeit prototypisch entworfen und implementiert wurde.

Während der Analyse des Projektziels stellte sich bereits heraus, dass solch ein Internetspiel hohe Anforderungen an Skalierbarkeit und Performanz stellt: Skalierbarkeit, da bei Erfolg – und somit steigender Spieler-Zahl – die Last rapide ansteigen kann, sowie Performanz, da die Reaktionsgeschwindigkeit eines Spiels eng verknüpft mit der Zufriedenheit des Spielers ist. Des Weiteren wurde eine effiziente und komfortable Möglichkeit zur Wartung des Spiels als wichtiges Kriterium für die Erhaltung eines ordnungsgemäßen Betriebs definiert.

Im Zuge des Entwurfs wurde als Anwendungsarchitektur Command-Query Responsibility Segregation (CQRS) gewählt, welche in Kombination mit Event Sourcing eine solide und erweiterbare Basis für ein logisch aufgebautes und leicht erweiterbares System versprach.

Parallel zu der Implementierung des Spiels wurde die Wartungsanwendung *CZ.Manager* entwickelt, welche den Einblick sowie Eingriff in die internen Komponenten von *CombatZone* erlaubt. Dies verhalf zu einer erleichterten Entwicklung und Wartung der Haupt-Applikation.

Die Performanz des *CombatZone*-Prototyps genügte für die gesetzten Anforderungen, jedoch war sowohl die Skalierbarkeit als auch die Performanz durch die Leistung der eigenen Event Store Implementation begrenzt. Hierzu werden im Ausblick Möglichkeiten aufgezeigt, wie bei sich ändernden Anforderungen diese Begrenzung gelöst werden kann.

Durchgehende Verwendung von Domain-Driven Design (DDD) und Event Sourcing machte es im Datenbestand später noch sehr gut ersichtlich, wie sich Werte ergaben und warum Änderungen geschehen sind. Dies machte das System als Ganzes leichter verständlich, sodass die Wartung und Überwachung vereinfacht wurde.

Die Sicherung und Wiederherstellung des Systems erwies sich als trivial, da der gesamte Datenbestand in einfachen Dateien gespeichert ist, welche mittels eines Wartungsprogramms oder Azure-kompatibler Software kopiert werden konnten.

Insgesamt wurden alle Anforderungen an das Projekt hinreichend erfüllt sodass es als Erfolg angesehen werden kann.

10. Ausblick

Während der Entwurfs- und Entwicklungsphase ergaben sich viele Themengebiete und Ideen, welche der Funktionalität und Erweiterbarkeit der Spiel-Infrastruktur zuträglich gewesen wären, jedoch nicht im Rahmen dieser Arbeit genauer betrachtet werden konnten. Diese Erweiterungen sind oft nur mit einer komplexeren Infrastruktur realisierbar. Die Modularität von CQRS ermöglicht es jedoch, die Architektur dahingehend nachträglich zu erweitern, dass diese Komplexität erst zu tragen kommt, wenn diese auch benötigt wird.

Innerhalb dieses Kapitels werden nun ausgewählte Thematiken angeschnitten.

10.1. Spiel-Elemente

Es gibt mehrere Spiel-Elemente, welche weiter ausgebaut werden könnten. Beispielsweise wäre es sinnvoll, wenn die unterschiedlichen Sektor-Arten auch spielerisch mehr Nutzen aufweisen würden. Derzeit sind nur Sonnensystem-Sektoren anders implementiert, der Rest unterscheidet sich nur in ästhetischen Sinne.

Wie bereits im früheren Kapitel *Spielkonzept* angedeutet, wäre es z.B. möglich, Kämpfe in solchen Sektoren zu beeinflussen, indem entweder der Angreifer oder der Verteidiger bevorteilt wird. Wurmlöcher könnten darüber hinaus untereinander Punkt-zu-Punkt Verbindungen besitzen und somit Flotten eine beschleunigte Fortbewegung ermöglichen.

10.2. Message-Routing

In der aktuellen Implementation benötigen alle Systemkomponenten, die Commands direkt übermitteln wollen, Kenntnis darüber, wie das Command Handling der Bounded Contexts partitioniert ist. Dies liegt daran, dass jede Command Partition exakt eine Warteschlange besitzt, und diese zur (direkten) Übermittlung des Commands angesprochen werden muss.

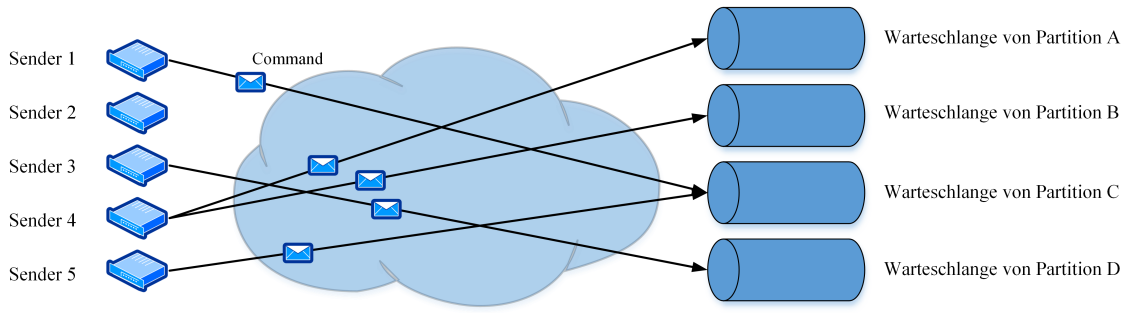


Abbildung 10.1.: Direkte Nachrichtenübermittlung

Dies hat zur Folge, dass jede Änderung der Partitionierung Anpassungen an allen weiteren Komponenten nach sich zieht. Da dies Aufwand und eine potenziell höhere Fehleranfälligkeit bedeutet, sollte bei häufigen Partitionsänderungen eine Möglichkeit gefunden werden, um die Command-sendenden Komponenten stärker von den Warteschlangen zu entkoppeln.

Eine oft genutzte Möglichkeit hierfür ist die Implementierung eines Message Routers. Alle versendeten Commands würden hierbei an den Router gesendet werden, welcher diese wiederum an die einzelnen Warteschlangen der Command Partitionen verteilen würde. Dies hätte den Vorteil, dass nur der Router über die aktuelle Partitionierung Kenntnis haben muss, was den Aufwand bei einer Änderung verringern würde.

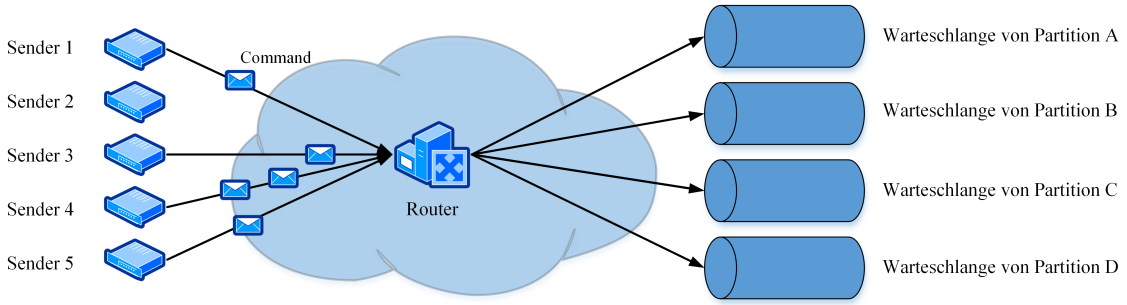


Abbildung 10.2.: Indirekte Nachrichtenübermittlung mittels Message Router

Bei einem solchen Aufbau muss jedoch Bedacht darauf gelegt werden, dass die Latenz des Command Handlings nicht zu stark leidet. Zwar besitzt der Router nur eine kleine Logikverarbeitung, er muss jedoch stark für einen hohen Nachrichtendurchsatz optimiert werden. Je nach erwarteten Nachrichtenaufkommen bedeutet das eine weitere Komponente, welche skaliert werden muss. Darüber hinaus muss der Command Router auch ausfallsicher konzipiert werden, da ein Ausfall des Routers jegliche Konnektivität zu den Command Processors unterbinden würde.

10.3. Erhöhung der Skalierbarkeit

Der Event Store ist die begrenzende Komponente in der zum Zeitpunkt dieser Arbeit aktuellen Version von *CombatZone*. Dies war zu erwarten, denn wie bereits während der Planung der Event Store Implementation (s. Abschnitt 5.7) erwähnt, handelt es sich hier um eine sehr minimalistische Version. Soll also dieser Aspekt der Architektur optimiert werden, so gibt es hierfür vier Varianten, welche nachfolgend (kurz) betrachtet werden sollen.

10.3.1. Verwendung eines vorhandenen Event Stores

Wie bereits im Abschnitt 5.7 angemerkt, gibt es bereits vorhandene Event Store Implementationen, sowohl unter Open Source Lizenzmodellen als auch kommerzieller Natur.

Durch die Verwendung eines bereits existierenden Event Stores kann – insbesondere bei erhöhten Anforderungen – eigener Entwicklungsaufwand eingespart werden. Die Lösung mit dem wahrscheinlich größten Potenzial ist der vom CQRS-Namensgeber Greg Young entworfene und unter [7] verfügbare *Event Store*. Kurz vor Finalisierung dieser Arbeit wurde hiervon die finale Version 1.0 herausgebracht.

10.3.2. Wechsel auf eine weitere Speicherungstechnik

Für synchrone und sequentielle Schreiboperationen wurde in der dieser Arbeit beiliegenden Implementation beinahe das Limit von Azures BLOB Storage erreicht.

Durch Verwendung einer alternativen Persistierungsoption könnte dieser Engpass umgangen werden. Eine Erklärung der Alternativen ist im Rahmen dieses Abschnittes nicht möglich, daher folgt nachfolgend ausschließlich eine Nennung der möglichen Alternativen:

- Azure Table Storage
- SQL Azure
- Ein- oder mehrere Redis-Server-Instanzen
- Datei-basierte Speicherung

10.3.3. Partitionierung des Event Stores

Durch das in [15] beschriebene und im Zuge dieser Arbeit implementierte Verfahren kann es in jeder Event Store Instanz ausschließlich eine aktive – und somit beschreibbare – Datei geben. Da im "Realm" BC alle Universen enthalten sind, muss diese eine Datei die Last aller aktiven Universen verkraften können.

Würde der "Realm" BC, vergleichbar einer Tenant-basierten Aufteilung (Mandantenfähigkeit), in weitere Segmente aufgeteilt werden, so würde dies eine weitere Skalierung ermöglichen. Diese Segmente könnten anhand der Universen erstellt werden. Jedes Universum würde somit eine eigene Event Store Instanz bekommen, wodurch sich die Leistung der Universen nicht gegenseitig beeinflussen könnte.

Im Zuge dieser Aufteilung könnte somit auch eine horizontale Skalierung erreicht werden, indem die aktiven Universen auf unterschiedlichen Worker-Instanzen gehostet werden. Ein vorgeschalteter Message Router (s. Abschnitt 10.2) würde sich hierbei um die korrekte Verteilung der Commands zu der korrekten Worker-Instanz kümmern. [17]

10.3.4. Asynchrone Speicherung

Die derzeitige Implementation erfordert es, dass die Schreiboperation der jeweiligen Command-Transaktion abgeschlossen ist, bevor der Command Processor (und somit dessen Thread) die Bearbeitung fortführt. Da hierdurch jede Schreiboperation einzeln ausgeführt wird, ist die Anzahl der ausgeführten Commands pro Sekunde begrenzt durch die Anzahl der Schreiboperationen, welche auf einen BLOB ausgeführt werden können.

Würde diese synchrone Vorgehensweise in eine asynchrone Alternative geändert werden, so dürfte dies eine bessere Verarbeitungsrate ergeben. Hierdurch könnten z.B. die Daten aller parallel verarbeitenden Command-Transaktionen in einer kombinierten Schreiboperation persistiert werden.

Die Effizienzsteigerung dürfte nach einer vertikalen Skalierung der Worker-Instanz umso größer sein, da durch das Hinzufügen von weiteren Prozessorkernen eine bessere Parallelisierung realisiert werden könnte. Dies hätte zur Folge, dass dadurch mehr Schreiboperationen kombiniert und somit eine höhere Bearbeitungsrate erzielt werden könnte.

10.4. Erhöhung der Verfügbarkeit

Während die Web-Anwendung von *CombatZone* als hochverfügbar angesehen werden kann, ist dies für die Worker-Anwendung nicht der Fall. Dementsprechend ist die Verfügbarkeit des Gesamtsystems eingeschränkt.

Nachfolgend werden zwei Ansätze dargestellt, welche die Verfügbarkeit des Systems weiter erhöhen würden.

10.4.1. Umgang mit Worker-Ausfall

Da die Web-Applikation nur sehr lose an den Worker-Prozess gekoppelt ist, ergeben sich Szenarien, in denen die Web-Applikation funktionsfähig ist, ohne dass ein Worker-Prozess verfügbar ist. Wird bspw. der einzige Worker-Prozess gerade neu-gestartet, so könnte die Web-Applikation dies durch das Empfangen von System-Events mitbekommen und dem User anzeigen.

Wenn der Web-Applikation bewusst ist, dass der Worker-Prozess nur wenige Sekunden nicht erreichbar ist (z.B. bei dem Einspielen von einem kleinen Patch), so könnte diese das stillschweigend hinnehmen und darauf vertrauen, dass der aktualisierte Worker-Prozess die sich füllende Warteschlange nach dessen Start zeitnah leert. Eine Möglichkeit zur Kommunikation dieses Szenarios wäre das System-Event "BoundedContextStoppedEvent", dies könnte entweder den Grund für das Stoppen des BCs beinhalten oder einen Zeitpunkt, zu dem erwartet wird, dass dieser wieder erreichbar ist.

10.4.2. Replikation & Fail-Over

In den Abschnitten 5.4.1 und 8.2 wurde bereits angedeutet, dass es noch weitere Möglichkeiten der Partitionierung gibt. Diese basierten jedoch ausschließlich auf Ansätzen, welche einen Bounded Context auf jeweils einer einzelnen Worker-Instanz beließen. Hierdurch kann ein Bounded Context nicht als hochverfügbar angesehen werden, denn jede Worker-Instanz ist somit ein Single-Point-of-Failure.

Eine Möglichkeit, dieses Problem zu lösen, wäre eine Aktiv/Passiv Cluster-Implementation. Hierbei würden stets zwei Instanzen eines Bounded Contexts sowie der Worker-Anwendung gleichzeitig aktiv sein. Die erste gestartete Instanz würde sofort die Arbeit aufnehmen und Commands verarbeiten. Die zweite gestartete Instanz würde das Dasein der ersten bemerken – beispielsweise durch eine Broadcast-Anfrage im Netzwerk – und eine Verbindung mit dieser aufbauen.

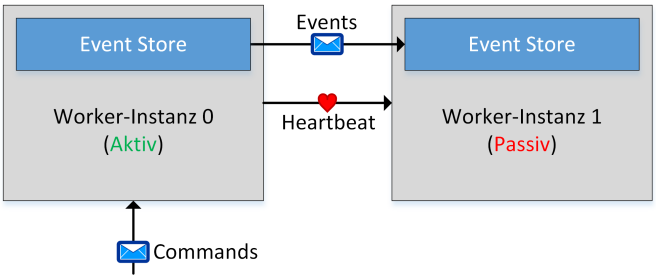


Abbildung 10.3.: Replizierender Fail-Over-Cluster

Durch diese offene Verbindung, über die in kurzen Intervallen ein Heartbeat-Signal ausgetauscht werden könnte, würde die zweite (passive) Instanz registrieren, wenn die erste ausfallen würde und könnte diese ersetzen.

Damit solch ein unverzügliches Fortsetzen der Arbeit möglich ist, würde es notwendig sein, dass der komplette Datenbestand in beiden Instanzen verfügbar ist. Um dies zu ermöglichen, würden hier alle neu hinzukommenden Events direkt an eine passive Instanz repliziert werden. Ob dies innerhalb oder außerhalb der Command-Transaktion erfolgen soll, muss anhand der benötigten Performanz und Datensicherheit abgewägt werden.

Dieser Aufbau würde darüber hinaus die Datensicherheit zusätzlich erhöhen, da der gesamte Datenbestand hiermit an zwei Orten gespeichert werden würden. Durch die Flexibilität von Windows Azure wäre es sogar möglich, dass sich diese zwei Server auf unterschiedlichen Rechenzentren und Kontinenten befinden.

Literaturverzeichnis

[1] Abdullin, Rinat: *Recent Lessons Learned in Lokad.CQRS.* – Blog – Zugegriffen am: 27.01.2013 - Verfügbar unter http://abdullin.com/journal/2012/7/5/recent-lessons-learned-in-lokadcqrs.html

[2] Ben Armstrong: *Time Synchronization in Hyper-V.* 2010. – Blog – Zugegriffen am: 21.01.2013 - Verfügbar unter http://blogs.msdn.com/b/virtual_pc_guy/archive/2010/11/19/time-synchronization-in-hyper-v.aspx

[3] Bill Wilder: *Azure FAQ: How frequently is the clock on my Windows Azure VM synchronized?* 2011. – Blog – Zugegriffen am: 21.01.2013 - Verfügbar unter http://blog.codingoutloud.com/2011/08/25/azure-faq-how-frequently-is-the-clock-on-my-windows-azure-vm-synchronized/

[4] Christopher M. Moyer: *Building Applications in the Cloud: Concepts, Patterns, and Projects.* Addison-Wesley Professional, 2011. – ISBN 978-0-321-72020-7

[5] deVadoss, John ; Lascelles, Francois ; Rischbeck, Thomas ; Wilhelmsen, Herbjörn ; Plunkett, Tom ; Little, Mark ; Assi, Anthony ; Liu, Anna ; Chappell, David ; Roy, Satadru ; Simon, Arnaud ; Erl, Thomas: *Service-Oriented Infrastructure: On-Premise and in the Cloud.* Prentice Hall, 2013. – ISBN 978-0-13-236028-9

[6] Evans, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003. – ISBN 978-0-321-12521-7

[7] Event Store LLP: *Event Store.* – Projekt-Homepage - Zugegriffen am: 23.01.2013 - Verfügbar unter http://geteventstore.com/

[8] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002. – ISBN 978-0-321-12742-6

[9] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003. – ISBN 978-0-321-20068-6

[10] Lokad: *Lokad.CQRS Sample Project.* – Projekt-Homepage – Zugegriffen am: 25.01.2013 – Verfügbar unter http://lokad.github.com/lokad-cqrs/

[11] Meyer, Bertrand: *Object-Oriented Software Construction.* Prentice Hall, 2000. – ISBN 978-0136291558

[12] Microsoft Corporation: *Microsoft Inductive User Interface Guidelines.* 2001. – Artikel – Zugegriffen am: 28.12.2012 – Verfügbar unter http://msdn.microsoft.com/en-us/library/ms997506.aspx

[13] Mizonov, Valery ; Manheim, Seth: *Windows Azure Queues and Windows Azure Service Bus Queues - Compared and Contrasted.* 2012. – Artikel – Zugegriffen am: 03.01.2013 – Verfügbar unter http://msdn.microsoft.com/en-us/library/hh767287%28VS.103%29.aspx

[14] ServiceStack: *JSON, CSV, JSV Text Serializers.* 2012. – Projekt-Homepage – Zugegriffen am: 04.01.2013 – Verfügbar unter http://www.servicestack.net/docs/text-serializers/json-csv-jsv-serializers

[15] Sheehy, Justin ; Smith, David: *Bitcask - A Log-Structured Hash Table for Fast Key/Value Data*. 2010. – Artikel – Zugegriffen am: 10.01.2013 – Verfügbar unter `http://downloads.basho.com/papers/bitcask-intro.pdf`

[16] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme : Prinzipien und Paradigmen*. München u.a : Pearson Studium, 2008. – ISBN 978-38-2737293-2

[17] Vernon, Vaughn: *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. – Rohfassung – ISBN 978-0-321-83457-7

[18] Young, Greg: *CQRS Documents*. 2010. – Artikel – Zugegriffen am: 18.01.2013 – Verfügbar unter `http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`

Glossar

autonom Miteinander verbundene, *autonome* Systeme oder Komponenten, funktionieren unabhängig voneinander und können auch einzeln betrieben werden.

Binary Large Object (BLOB) Binärdaten, welche als ein Element gruppiert gespeichert wurden.

Inhärenz Eine inhärente Eigenschaft ist eine unveränderliche, essenzielle Eigenschaft.

Minimap Als *Minimap* wird eine verkleinerte Version einer Karte dargestellt, wie sie in Spielen oder Anwendungen vorkommen kann.

Single Point of Failure Ein *Single Point of Failure* beschreibt eine Komponente eines Systems, deren Ausfall das gesamte System in einen nicht-betriebsfähigen Zustand bringen würde.

Technologieagnostisch Etwas *Technologieagnostisches* funktioniert identisch, unabhängig davon welche Technologie verwendet wurde.

User Experience Der Begriff *User Experience* beschreibt die Art, wie ein User die Benutzung einer Software erlebt. Dies schließt bspw. Ästhetik, Benutzerfreundlichkeit und Reaktionsfreudigkeit ein.

A. Anhang

A.1. Quelltext BLOB-Storage Latenz-Testprogramm

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using CombatZone.Domain.Identity.Account;
using CombatZone.MessageStores;
using Microsoft.WindowsAzure.Storage;
using ServiceStack.Text;

namespace BlobTapeLatencyTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var storageAccount = CloudStorageAccount.Parse(@"DefaultEndpointsProtocol=↩
                https;AccountName=...;AccountKey=...");

            var blobClient = storageAccount.CreateCloudBlobClient();

            var container = blobClient.GetContainerReference("blobtapelatencytest");
            container.DeleteIfExists();
            container.CreateIfNotExists();

            List<double> latencies = new List<double>();

            BlobTapeStore tape = new BlobTapeStore(container);

            DateTime dt;
            for (int i = 0; i < 500; i++)
            {
                dt = DateTime.UtcNow;

                RegisterAccountCommand cmd = new RegisterAccountCommand(Guid.NewGuid(),↩
                    "TestAccount", "...", "test2@test.de");
                AccountRegisteredEvent evt = new AccountRegisteredEvent(cmd.AggregateId↩
                    , cmd.AccountName, cmd.SaltedPasswordHash, cmd.EmailAddress);
                byte[] buffer;
                using (var ms = new MemoryStream())
                {
                    JsonSerializer.SerializeToStream(cmd, ms);
                    JsonSerializer.SerializeToStream(evt, ms);
                    buffer = ms.GetBuffer();
                }

                tape.Append("test", buffer, -1);
                latencies.Add((DateTime.UtcNow - dt).TotalMilliseconds);
            }

            Console.WriteLine("{0} tries, {1} ms latency on average", latencies.Count, ↩
                latencies.Average(x => x));
            Console.ReadKey();
        }
    }
}
```

A.2. Schätzkalkulation für Event-Aufkommen

$A = Anzahl\ registrierter\ Benutzer = 1500$
$B = Durchschn.\ Anzahl\ Sonnensystem - Sektoren = 8$
$C = Durchschn.\ Bauzeit$

$$C = \frac{Summe\ d.\ Bauzeiten}{Anzahl\ Schiffstypen} = \frac{1+3+5+7+10}{5} = 5{,}2$$

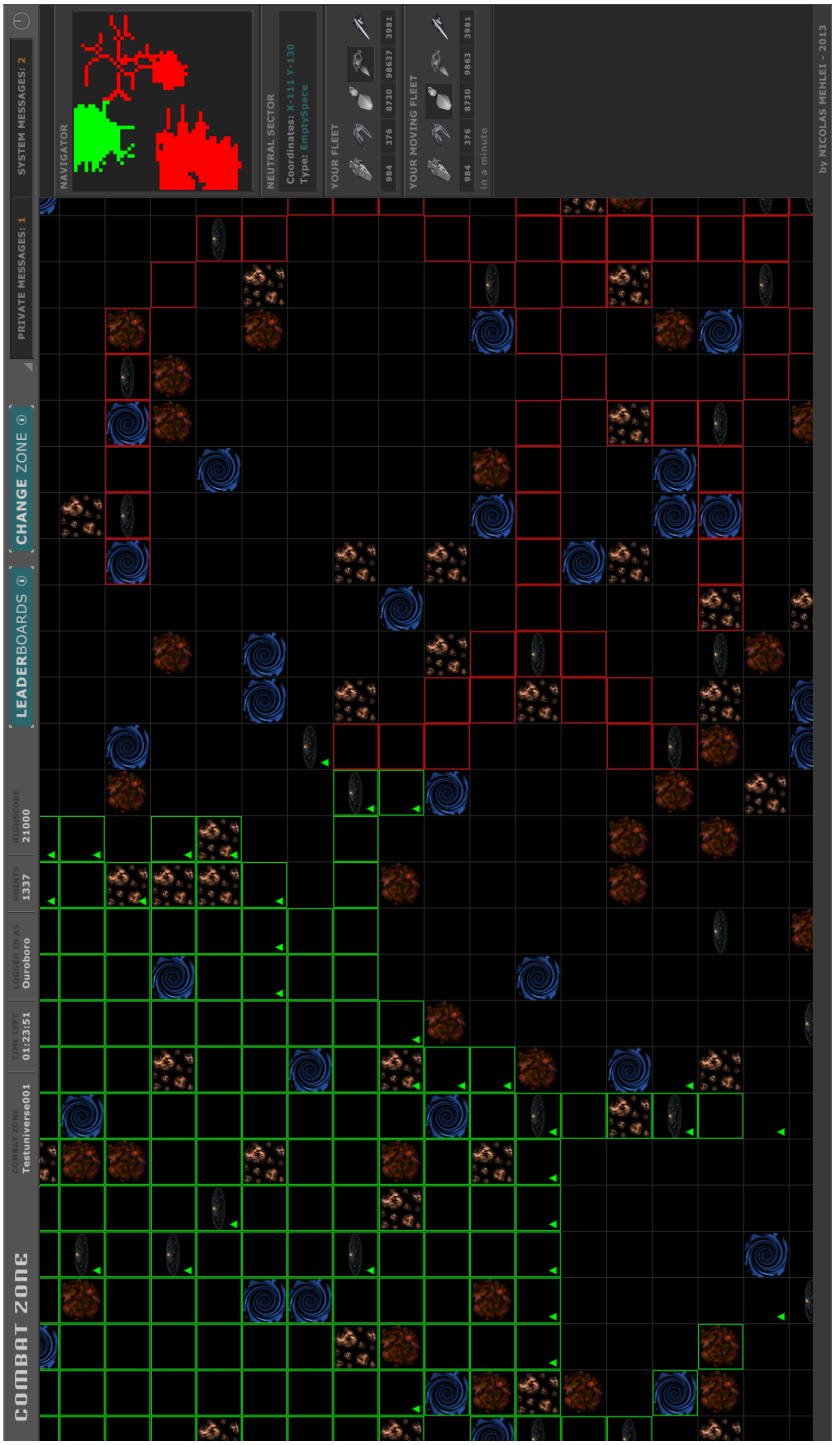
$$Events\ pro\ Minute = \frac{A * B}{C} = \frac{12000}{5{,}2} = 2307$$

$$Events\ pro\ Sekunde = \frac{2307}{60} = 38{,}5$$

A.3. Quelltext Datenkorruptionsprogramm

```
// connect to blob storage
var storageAccount = CloudStorageAccount.DevelopmentStorageAccount;
var blobClient = storageAccount.CreateCloudBlobClient();

// fetch blob reference
var container = blobClient.GetContainerReference("cz-log-realm");
var blob = container.GetPageBlobReference("00000016-2013-01-15-114935.dat");

var buffer = new byte[512*1024];

// read current blob content
var readStream = blob.OpenRead();
readStream.Read(buffer, 0, buffer.Length);
readStream.Close();
var stream = blob.OpenWrite(512*1024);

// "corrupt" data
buffer[127] = 65;

// write back "corrupted" data
stream.Write(buffer, 0, buffer.Length);
stream.Flush();
stream.Close();

Console.WriteLine("written");
Console.ReadLine();
```

A.4. Screenshot des User Interface



A.5. Liste der Unit-Tests

Klasse	Test-Bezeichnung
Aggregate “Universe“	Universe_Can_Be_Created Universe_Create_Calculates_Correct_Start_Coordinates Universe_Can_Be_Renamed Universe_Rename_Is_Discarded_If_Name_Is_Unchanged Universe_Can_Be_Started Universe_Can_Be_Ended
Aggregate “Sector“	Sector_Can_Be_Instantiated Sector_Can_Be_Spawned Sector_Cannot_Be_Spawned_Twice Sector_Spawn_Creates_SectorSpawnedEvent Sector_Spawn_Creates_Valid_SectorSpawnedEvent Start_Outpost_Can_Be_Established Start_Outpost_Cannot_Be_Established_On_Empty_Sector Start_Outpost_Cannot_Be_Established_On_Asteroid_Belt Start_Outpost_Cannot_Be_Established_On_Already_Taken_Sector Sector_Can_Finish_First_Ship Sector_Can_Finish_Second_Ship Sector_Ship_Count_Stays_Correct_After_Dispatched_Fleet
Aggregate “MovingFleet“	MovingFleet_Can_Be_Instantiated MovingFleet_Can_Be_Created MovingFleet_Can_Arrive
Aggregate “Player“	Player_Can_Be_Created Player_Can_Join_Universe
Value Type “ShipCollection“	Can_Add_ShipCollections

A.6. Inhalt der beiliegenden CD

- Verzeichnis **Anhänge**
  Alle im Anhang aufgeführten Quelltext-Fragmente
- Verzeichnis **Bachelorarbeit-PDF**
  Diese Arbeit in digitaler Version als PDF-Dokument.
- Verzeichnis **Quellen**
  Alle Online-Quellen, welche im Literaturverzeichnis aufgeführt sind.
- Verzeichnis **Source-Code**
  Der Source-Code von *CombatZone*.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum	Unterschrift