Inhaltsverzeichnis

Einführung

Konfiguration

Einrichtung von PostgreSQL 8.4 unter Ubuntu 10.10

Pakete installieren:

$ apt-get install postgresql-8.4 postgresql-client-8.4

Dann zum User "postgres" wechseln und den "swa"-User anlegen:

$ sudo -u postgres sh

$ createuser swa -P

Enter password for new role: swa11

Enter it again: swa11

Shall the new role be a superuser? (y/n) y

Dann die Datenbank anlegen:

$ createdb swa

(dann exit, d.h. die subshell vom postgres-User verlassen)

in /etc/postgresql/8.4/main/pg_hba.conf "ident" auf md5 ändern (2x)

Den Datenbankserver neu starten:
$ sudo service postgresql restart

Danach sollte man sich mit:

$ psql swa swa

einloggen können (Passwort = swa11 wie in der Angabe!)

Generieren der Model-Klassen + DAOs + Unit-Tests

Das Projekt unter Eclipse öffnen.

Die Datei src/workflow/generator.mwe auswählen und mit "Run as MWE Workflow" ausführen -> Dateien werden erstellt:

- Entities in src-gen/
- DAOs in src-gen/
- Unit-Tests in src-gen/
- drop-tables.sql im Hauptverzeichnis
- persistence.xml in bin/META_INF/

Danach kann man die Tests mit "Run as JUnit Test" ausführen - das erstellt beim ersten Benutzen die Tabellen.

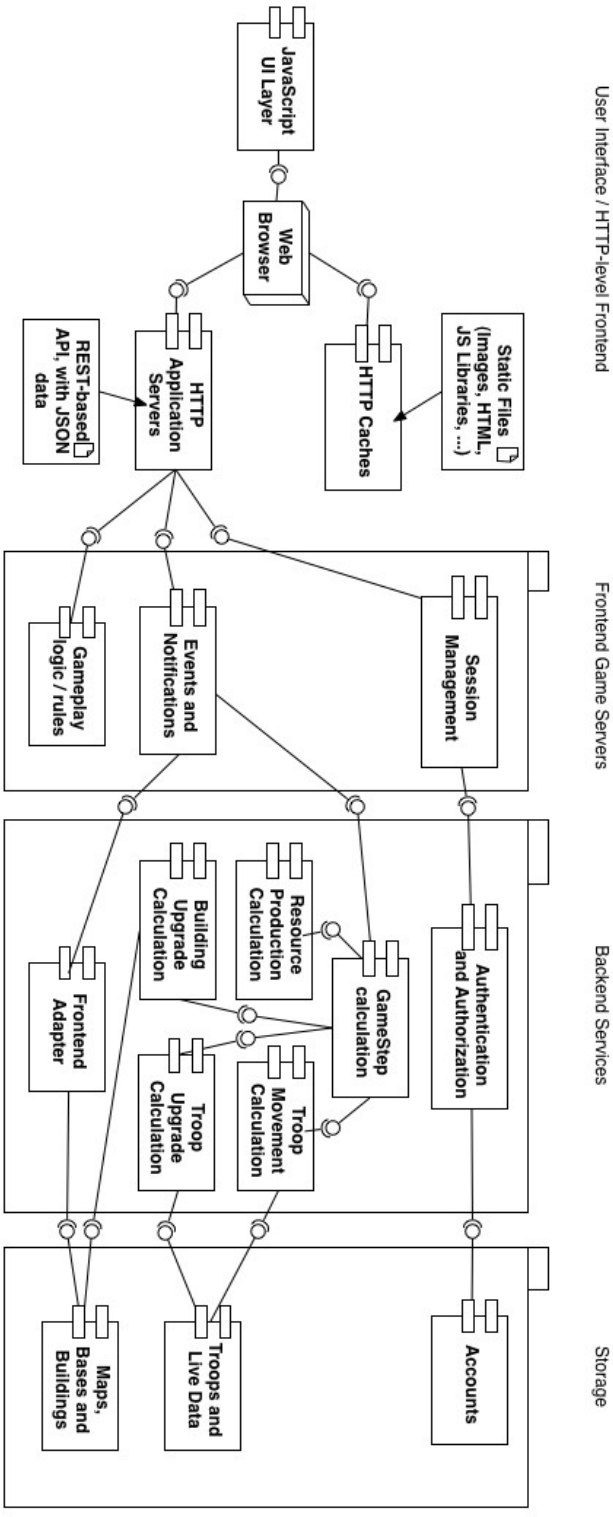
Tomcat-Server unter Eclipse for Java EE einrichten

verwendete tomcat version:
http://apache.webersiedlung.at/tomcat/tomcat-7/v7.0.14/bin/
einrichten des eclipse plugins:
http://www.mulesoft.com/tomcat-eclipse

Architektur

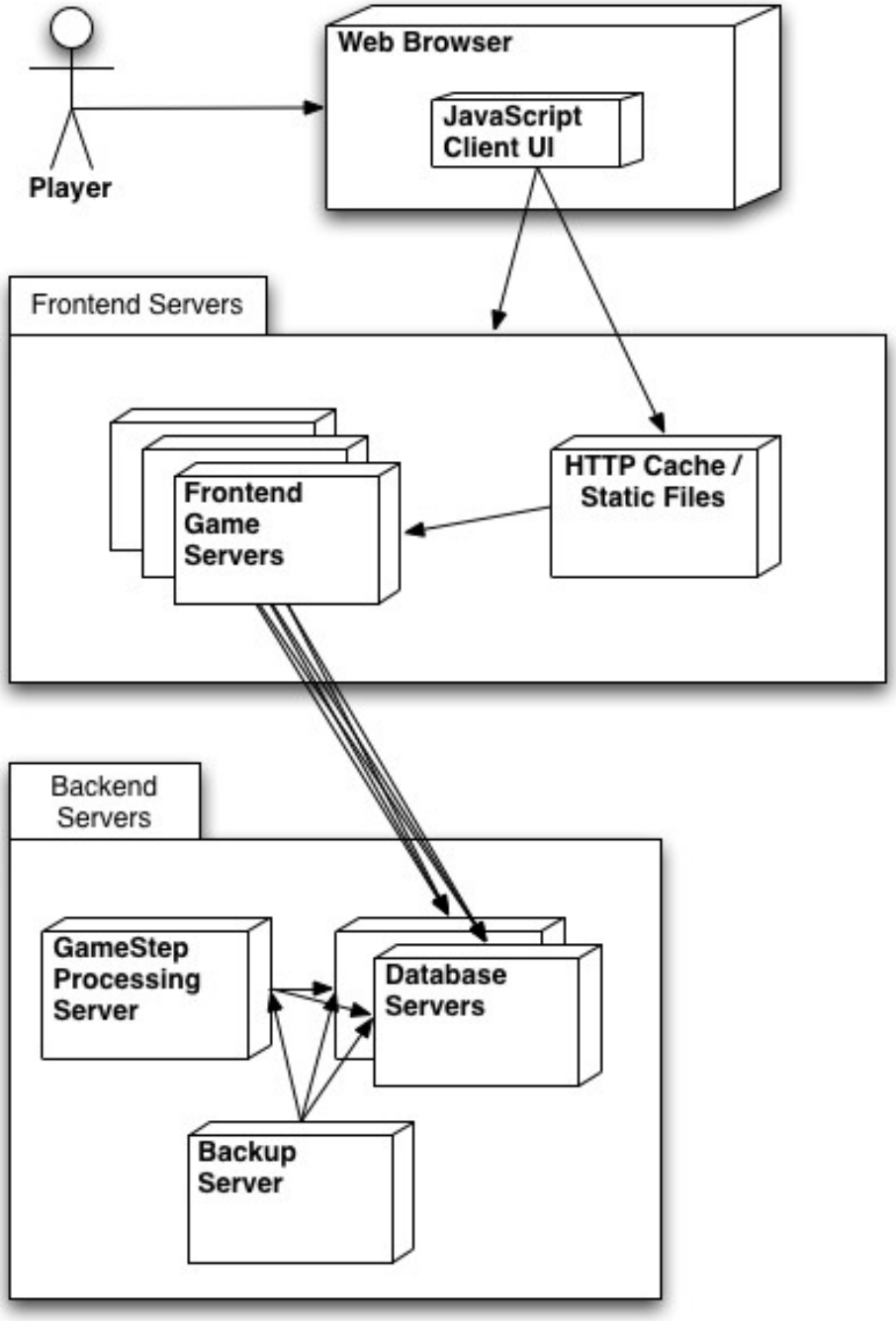
Komponentendiagramm

Dieses Diagramm beschreibt die Komponenten und Ebenen der Web-Anwendungsarchitektur. Außerdem werden des weiteren die Schnittstellen zwischen den einzelnen Komponenten beschrieben, sowie die Verbindungen zum formen des Systems aufgezeigt.



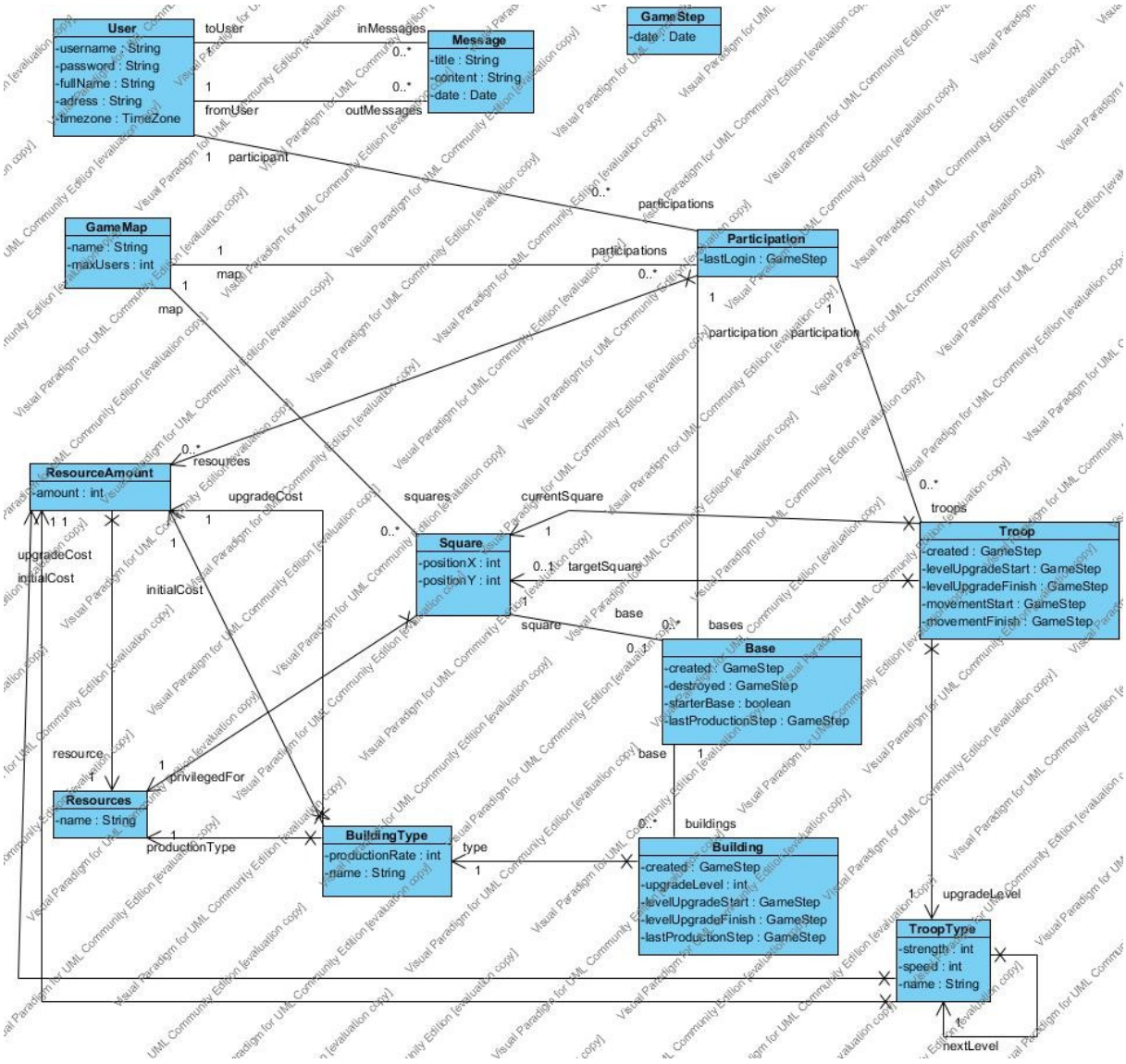
Deployment-diagramm

Der Service stellt sich in geteilter Form als Frontend und Backend dar. Das Frontend verarbeitet die Eingaben und Ausgaben. Das Backend beinhaltet den eigentlichen GameServer und den Datenbank Server. Backups werden von der Datenbank erstellt. Um die Ausfallsicherheit zu erhöhen kann das Backend mit mehreren Forntends kommunizieren. Das Backend kann wiederum durch Primary-Backup vor dem Ausfall abgesichert werden.



Das Frontend stellt sich als zustandslos dar. Das Backend hingegen verwaltet den aktuellen Gamestep.

Datenmodell



User: The Timezone will also be stored, as all time data in the system is UTC-based (Unix Timestamp) or based on the artificial „GameStep" internal clock (also used for synchronization).

Message: The message simply describes a message from a user to another user. The timestamp is saved, and based on the last message ID, the client can request new messages. When a user is deleted, all the messages that she has sent/received will also be deleted.

GameMap: The map has a name (used for displaying in the UI) and the maximum number of active users allowed. This can be used to load balance between different maps (e.g. don't show full maps, etc..).

Participation: The participation describes the user participating in a map. Using the „since" GameStep value, the system can determine when a new user has entered the game. The participation object also holds the resources that the user has available on that

map (here „map" stands for the container type map that maps an int value to each Resource enum).

Resource: This is an enumeration type describing the different types of resources supported by the system, for example gold, wood or food.

Square: A square is a single field in the game onto which bases can be placed
and on which troops can stand. It is part of a larger map, and it can be privileged for a specific resource, which makes harvesting this resource easier/more lucrative.

Base: This is a specific base that a user created (the user is identified by the participation). It saves the creation date (i.e. when the user has obtained the base) and the destruction date (i.e. when the user has quit or was defeated from the base). These dates are stored via the internal GameStep time, which is an abstract time used for one calculation step of the game world, and helps with synchronizing different clients and servers. The base sits on a specific square, and it's also known whether or not this base is the starter base for the user or not (there's only one starter base for each user on a map).

Building: Inside bases, there are several slots available for creating buildings. Each building has an ID, and a GameStep timestamp when it was created (this is used for synchronizing the building creation process). There are different building types, described by the enumeration type BuildingType. The upgrade_level specifies to which degree the building has been upgraded, in discrete levels (up to a specified maximum). When the building is being upgraded, the level_upgrade_start GameStep identified stores the GameStep position when the upgrade has started, and the level_update_finish time stores the GameStep position when the upgrade will be finished (so future invocations of the backend update procedure can increase the upgrade level and so that the client can calculate the percentage done based on the current GameStep value). The last_production_step refers to the last time the production of this building has been calculated. This allows to asynchronously update the resources of a user (e.g. there is no need to calculate the resources for users that are not logged in – the game backend can simply multiply the difference between current GameStep and last_production_step with the production_rate of the building
type to determine the resources produced when the user logs in (and during the game, obviously). It also allows more fine-grained granularity for updating the in-game data without having to do long-running calculations/locking.

BuildingType: This class describes the metadata of a building. This includes the initial cost for creating the building (a Map from Resource to int, e.g. {Gold => 100, Wood => 50, Stone => 120}). The same for the upgrade costs (it's assumed that every upgrade costs the same after the initial building costs have been covered). The production_rate and production_type describe the amount of resources that this building type produces during a GameStep (for example, a gold mine could produce 5 amounts of gold every step).

GameStep: This is a simple numeric value that describes the current „step" in which the game is. When a certain amount of wall time has passed, this will be increased and updates will be carried out on the backend servers (i.e. buildings might be upgraded,

troops might be upgraded, troops might finish their movement, a fight between two troops might take place, etc..). This also allows clients to query what has changed since the last time their GameStep was updated.

Troop: This class describes one troop, belonging to a user (specified via „participation"). The GameStep value when it has been created is stored, as well as the current upgrade level, and (if a level upgrade is currently in progress) the start and (prospected) finish time of the upgrade operation. Also, the game must store the current square on which the troop resides, and the target square to which it wants to move (which is equal to the current square if the troop is stationary). The movement_start and movement_finish have similar semantics as the level_upgrade_start and level_upgrade_finish values, but for the movement from one square to another.

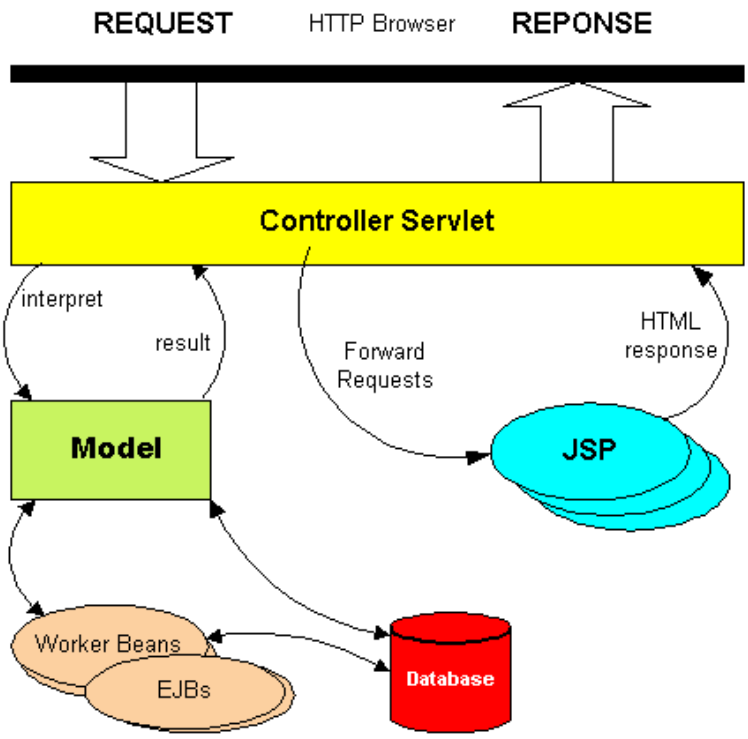
TroopType: This is the metadata class for troop that describes the troop type and the strength and speed. This is used for upgrading troops so they get better properties. The semantics of initial_cost and ugprade_cost are the same as for BuildingType.

CacheInvalidationEntry: This is a message that will be used to synchronize clients and servers, and servers with each other. The target_type describes the type of data which has been updated and the ID of the data which has been updated (target_id). Of course, „updated" here means created, updated and deleted equally. The client (or server) can either send the updated item with the invalidation entry, or ask for it specifically.

ItemType: An enumeration of the different data types used in the game (i.e.
User, Participation, Map, Troop, Base, Building, Square).

Frontend

Das Frontend ist für die Eingabe der Benutzer sowie für die Ausgabe der Spielstände zuständig.



Die aktuelle Version verwendet zu diesem Zweck Java Server Pages zur Darstellung der Ergenisse, welches es sich direkt aus der Datenbank holt. Als Controller der einzelnen Schritte werden Servlets eingesetzt, welche auch die Eingabe der Benutzer auf ihre Gültigkeit kontrollieren. Die Kommunikation mit der Datenbank erfolgt teils über Beans und zum größeren Teil direkt.

Backend

Das Backend berechnet die einzelnen Schritte im Spiel und schreibt die Ergebnisse direkt in die Datenbank. Außerdem wird nach jedem Berechnungsschritt das Frontend davon in Kenntnis gesetzt. Die Kommunikation zwischen Frontend und Backend wird im Folgenden beschrieben.

Backend Services für das Frontend

Das Backend bietet ein paar Services für das Frontend (siehe

die Klasse notification.Backend):

 * registerClient() - um sich für Notifications zu registrieren
 * unregisterClient() - um die Notifications wieder abzubestellen

 * onMapCreated() - "kümmere dich um die Updates für eine bestimmte Map"
 * onMapDestroyed() - "die Map gibt's nicht mehr - keine Updates ausführen"

Die Frontends müssen (zB hardcodiert oder über eine Konfigurations-Datei) über die einzelnen Hosts bescheid wissen, auf denen Backend-Instanzen laufen.

Finden einer Backend-Instanz für einen bestimmten Host

Dafür gibt's die Helper-Klasse "backend.BackendLocator", die eine statische Funktion "getBackendByHost(String)" besitzt. Übergibt man den Hostnamen, und läuft auf dem entsprechenden Host eine Backend-Instanz, so bekommt man ein "notification.Backend"-Objekt als Resultat, welches die oben beschriebenen Methoden bereit stellt.

API, die das Backend vom Frontend erwartet

Das Frontend muss das Interface "notification.Frontend" implementieren. Dieses Interface hat eh nur 1 Methode:

 * onGameStepCreated() - Backend hat soeben einen neuen GameStep erstellt

Die implementierende Instanz muss beim Aufruf von registerClient() und beim Aufruf von unregisterClient() übergeben werden.

Typischer Ablauf (am Beispiel 1 Frontend, 1 Backend)

1.) Backend wird gestartet:

```
java backend.BackendServer
```

2.) Frontend wird gestartet, und macht (schematisch) folgende Aufrufe:

```
Backend backend = BackendLocator.getBackendByHost("backend1.local");
backend.registerClient(this);
// ... Mainloop ...
backend.unregisterClient(this);
```

3.) Wenn eine neue Map erstellt wird, ruft das Frontend folgendes auf:

```
long newMapId = /* ID der neuen Map */;
backend.onMapCreated(newMapId);
```

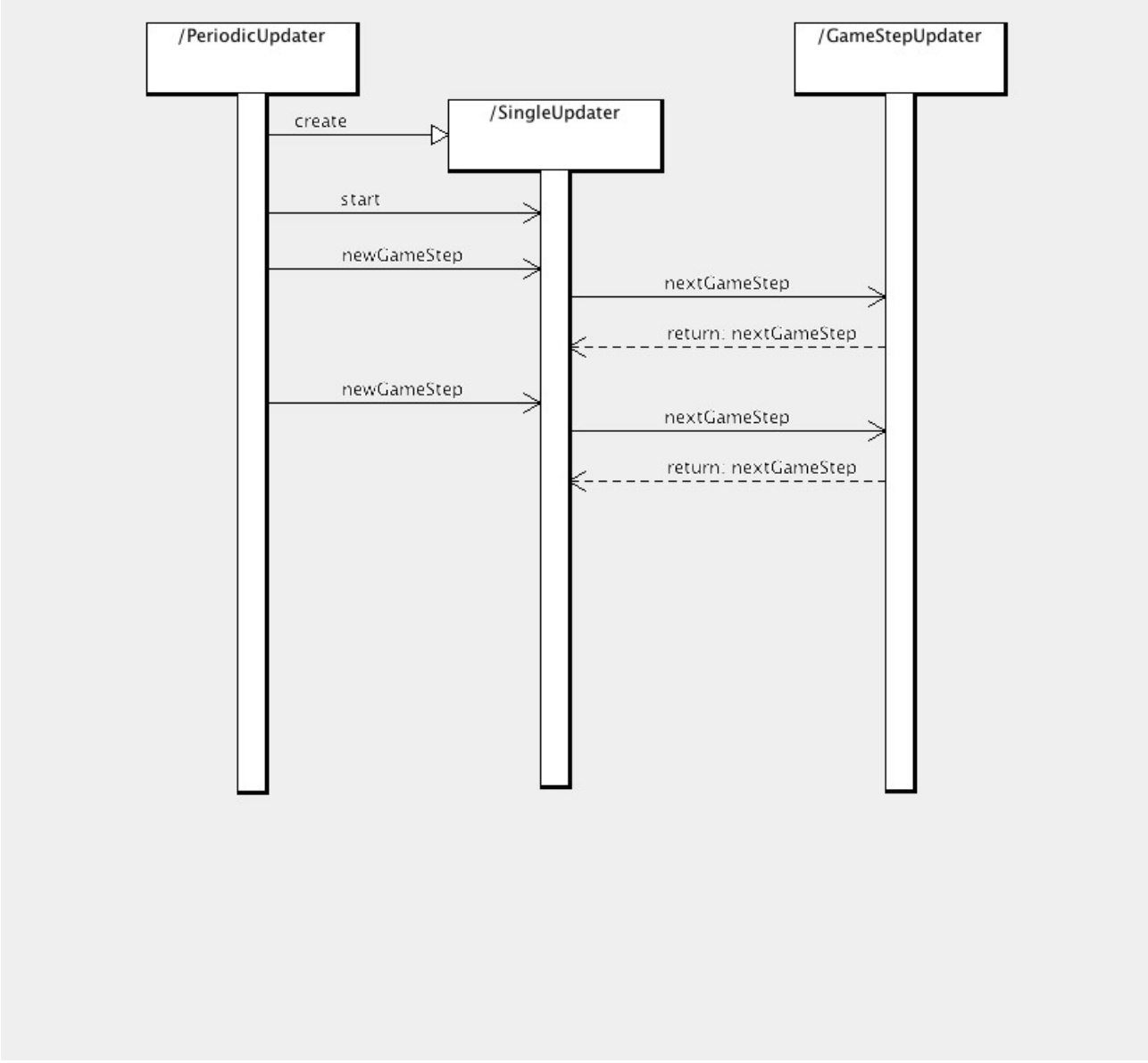
4.) Wenn eine Map beendet/geschlossen/entfernt wird, macht das Frontend:

```
long oldMapId = /* ID der gelöschten Map */;
backend.onMapDestroyed(oldMapId);
```

5.) Das Frontend implementiert die Methode onGameStepCreated(). wenn diese Funktion aufgerufen wird, dann muss sich das Frontend darum kümmern, die neuen Daten aus der Datenbank zu lesen, und diese den Clients anzuzeigen.

Berechnung der Spielschritte

Nach dem eine neue Map erstellt wurde, teilt dies das Frontend dem Backend mit. Dieses erstellt einen neuen GameStep und damit einen neuen periodischen Updater. Dieser periodische Updater ruft nach einer gewissen Zeit immer wieder den SingleUpdater auf. Dieser holt sich die aktulle Map aus der Datenbank, lässt alle Änderungen berechnen und schreibt die Ergebnisse in die Datenbank.



Ein Aufteilung der Berechnungsarbeit auf verschiedene Prozesse ist somit gegeben und kann leicht ausgelagert werden, wenn dies aus Performacegründen gewünscht ist.