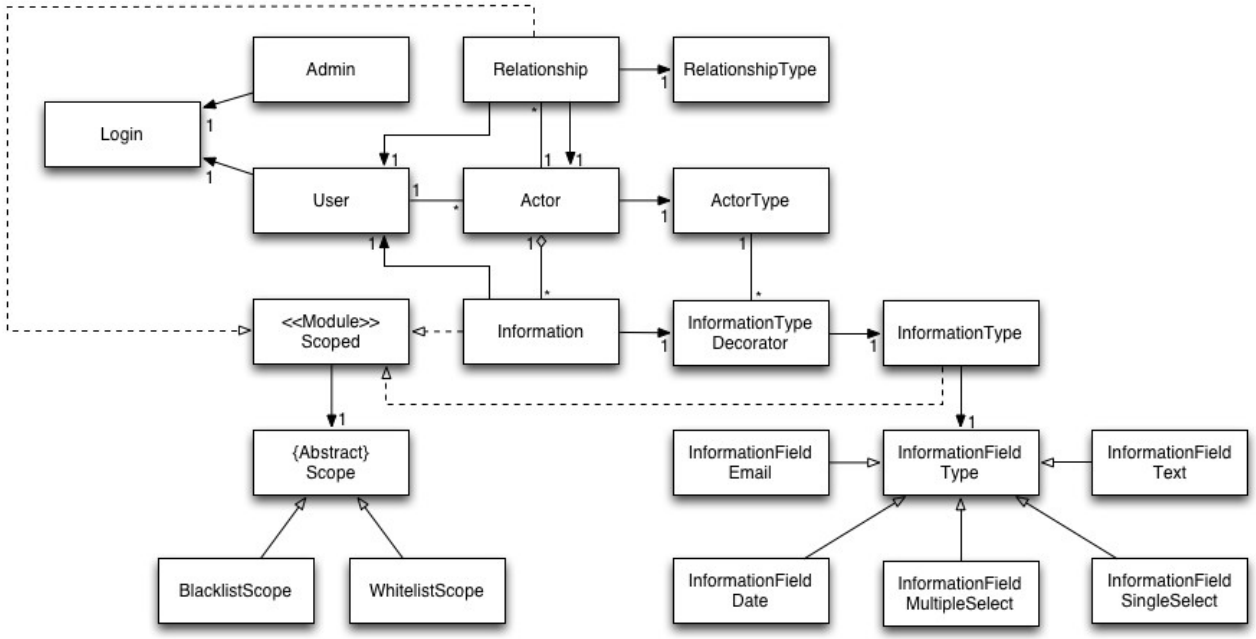
Dokumentation

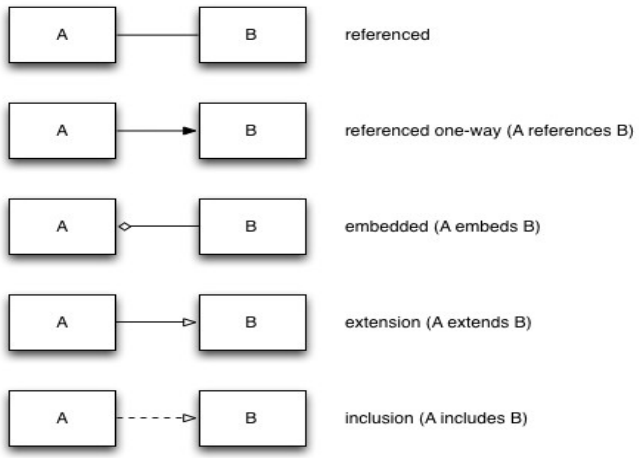
Im Rahmen des PSE-Praktikums musste nicht nur das Produkt zur Beurteilung abgegeben werden, sondern zusätzlich noch eine ganze Reihe an Deliverables. Die meisten Deliverables betreffen den Kunden nicht, da sie sich eher mit dem Arbeitsprozess als mit der Benutzung des Produkts auseinandersetzten. Eine Ausnahme dazu bilden die Design- und Quellcode-Dokumentation sowie das Benutzerhandbuch (Installationsanleitung).

Design-Dokumentation

UML-Klassendiagramm



- *UML-Klassendiagramm aller Model-Klassen*
- *Legende zum UML-Klassendiagramm*



Das Klassendiagramm enthält sämtliche Model-Klassen sowie die Beziehungen zwischen den Klassen. Das Diagramm entspricht also genau dem MongoDB-Datenmodell. Hinweis: Der Übersicht halber wurde auf die Darstellung der Klassenvariablen und -methoden verzichtet.

Einige Erläuterungen:

Beim Modellieren unseres MongoDB-Datenmodells war uns sehr wichtig, dass die bestmögliche Flexibilität des Systems gewährleistet wird. Unsere Applikation ist zwar eine Anwendung im Bereich Medizin, jedoch wollten wir ein System kreieren, welches auch in beliebig anderen Gebieten zum Einsatz kommen könnte. Wir haben deshalb von der Tatsache Gebrauch gemacht, dass MongoDB eine Dokument-basierte Datenbank ist. Wo immer möglich enthalten Objekte (bzw. Dokumente) in unserer Datenbank ausser ihrem Namen keine weiteren Stammdaten, sondern enthalten eine Liste an Referenzierungen an andere Objekte.

Die Ausnahme und damit quasi der “Einstieg” in den Datenmodell-Kreislauf bilden die Objekte der Klasse 'Information'. Diese enthalten jeweils genau eine wirkliche Information aus den Stammdaten sowie einen Verweis auf ein 'InformationType'-Objekt. Die 'Information'-Objekte sind zudem in einem 'Actor'-Objekt eingebettet und nicht referenziert, das heisst sie werden gleich innerhalb des 'Actor'-Objekts abgespeichert. Die 'Actor'-Objekte wiederum besitzen eine Referenz auf ein 'ActorType'-Objekt, welches dann eine Liste an Referenzen von 'InformationType'-Objekten (bzw. 'InformationTypeDecorator'-Objekte, siehe Abschnitt Design Patterns) hat. Dieser Kreislauf führt nun dazu, dass ein 'Actor'-Objekt jeweils genau solche 'Information'-Objekte besitzt, die auf ein 'InformationType'-Objekt zeigen auf welches auch das im 'Actor'-Objekt referenzierte 'ActorType'-Objekt zeigt.

Ähnlich wie die 'Actor'-Objekte sind die 'Relationship'-Objekte aufgebaut; sie enthalten jeweils eine Referenzierung auf ihr 'RelationshipType'-Objekt sowie auf genau zwei 'Actor'-Objekte. Mit ihnen lassen sich die Beziehungen zwischen verschiedenen 'Actor'-Objekten festhalten, deren Erhebung den eigentlichen Zweck der Applikation darstellt.
'Actor'- und 'Relationship'-Objekte sind jene zwei Objekt-Typen, welche in der Applikation dargestellt werden sollen. Damit die von den Benutzer eingegebenen Informationen aber nicht zwangsweise für alle Benutzer (oder einfach auch Besucher der Website) einsehbar sind, haben wir zudem sogenannte 'Scopes' (Sichtbarkeitsbereiche) implementiert.

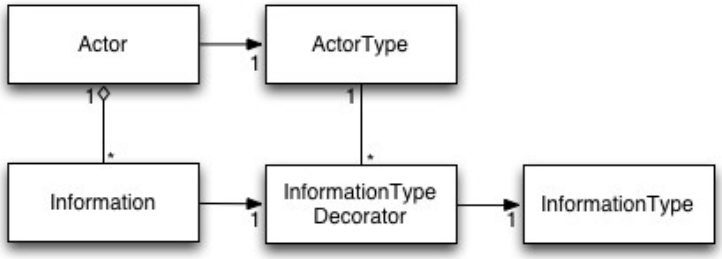
Wir haben uns für ein White-/Blacklist-Sytsem entschieden. Eine Whitelist enthält eine Liste von Benutzern, welche die Objekte, die diese Scope referenzieren, sehen dürfen (alle anderen Benutzer dürfen nicht); eine Blacklist enthält eine Liste von Benutzer die jenes Objekt nicht sehen dürfen (alle anderen Benutzer dürfen das Objekt sehen). Klassen, welche das Modul 'Scoped' beinhalten (siehe Abschnitt Design Patterns), haben einen definierbaren Sichtbarkeitsbereich. In unserem Modell sind das die Klassen 'Information' und 'Relationship'. Ein Benutzer kann also für jede Information eines 'Actor'-Objektes individuell entscheiden, wer alles darauf zugreifen kann. Gleiches gilt für die 'Relationship'-Objekte.

Zu beachten ist ausserdem unsere Design-Entscheidung, dass 'User' und 'Actor' zwei

verschiedene Klassen sind. 'Actor'-Objekte gehören jeweils einem bestimmten 'User'-Objekt, welches als einziges Schreibrechte dafür besitzt. Ein 'Actor'-Objekt hat genau ein 'User'-Objekt als Besitzer, aber ein 'User'-Objekt kann mehrere 'Actor'-Objekte (oder auch keines) besitzen. Damit soll z.B. Spitälern ermöglicht werden, dass das Sekretariat nur einen User-Account (='User'-Objekt) hat, aber mit diesem mehrere z.B. Ärzte (='Actor'-Objekte) verwalten kann.

Design Patterns

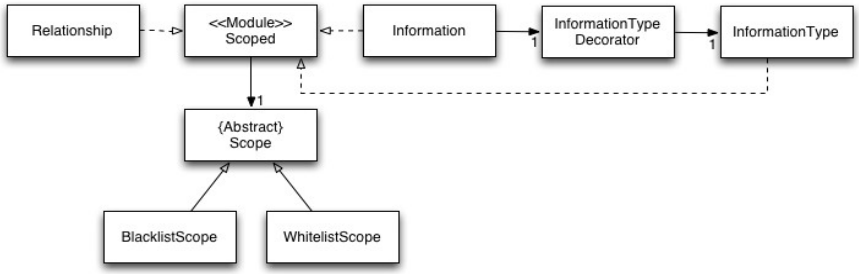
Decorator



- *UML-Klassendiagramm des Decorator-Patterns*

Erläuterung:
Bei der Implementierung unseres Datenmodells sind wir auf das Problem gestossen, dass 'ActorType'-Objekte zwar eine Liste an 'InformationType'-Objekten brauchen, zu jedem dieser Objekte jedoch im 'ActorType'-Objekt zusätzliche Daten (namentlich je ein Boolean ob das Ausfüllen des dazugehörigen 'Information0-objekts Pflicht ist und ob der Wert dieses Objekt durch die Suche gefunden werden kann) hätten speichern müssen. Wir mussten also als Abstraktion zwischen das 'ActorType'- und das 'InformationType'-Objekt ein weiteres Objekt der Decorator-Klasse 'InformationTypeDecorator' einbauen, welches von aussen angesprochen werden kann als wäre es ein Objekt der Klasse 'InformationType'.

Module



- *UML-Klassendiagramm des Module-Patterns*

Erläuterung:
Im Zuge der Implementation der Sichtbarkeitsbereiche (Scopes) für Objekte ergab sich folgendes Problem: Die Klassen 'Information', 'Relationship' sowie 'InformationType' (Sonderfall, denn 'Information'-Objekte sollen von ihrem 'InformationType'-Objekt den Sichtbarkeitsbereich erben falls dieser nicht spezifisch für sie deklariert wird) sollen alle dieselbe Funktionalität haben: Sie haben einen definierbaren Sichtbarkeitsbereich (Klasse 'Scope'). Eine Lösung dieses Problems wäre gewesen, eine abstrakte Klasse 'Scoped' zu machen und die anderen Klasse diese erweitern zu lassen. In Verbindung mit MongoDB hätte dies aber bedeutet, dass alle Dokumente von den Objekten der Klassen, welche die 'Scoped'-Klasse erweitern, innerhalb der Datenbank am selben Ort gespeichert worden wären. Dies wäre nicht ideal gewesen und praktischerweise liefert Ruby auch eine gute Alternative: die sogenannten Modules.
Modules lassen sich im Gegensatz zu Klassen nicht instanzieren, sie liefern viel mehr einfach eine Sammlung von Funktionen, Variablen und Konstanten, die in Klassen eingefügt (“include”) werden können. Durch Auslagern sämtlicher Funktionalitäten in ein Module names 'Scoped' konnten wir der Problematik bezüglich Speicherort in MongoDB elegant aus dem Weg gehen.

Model-View-Controller (MVC)

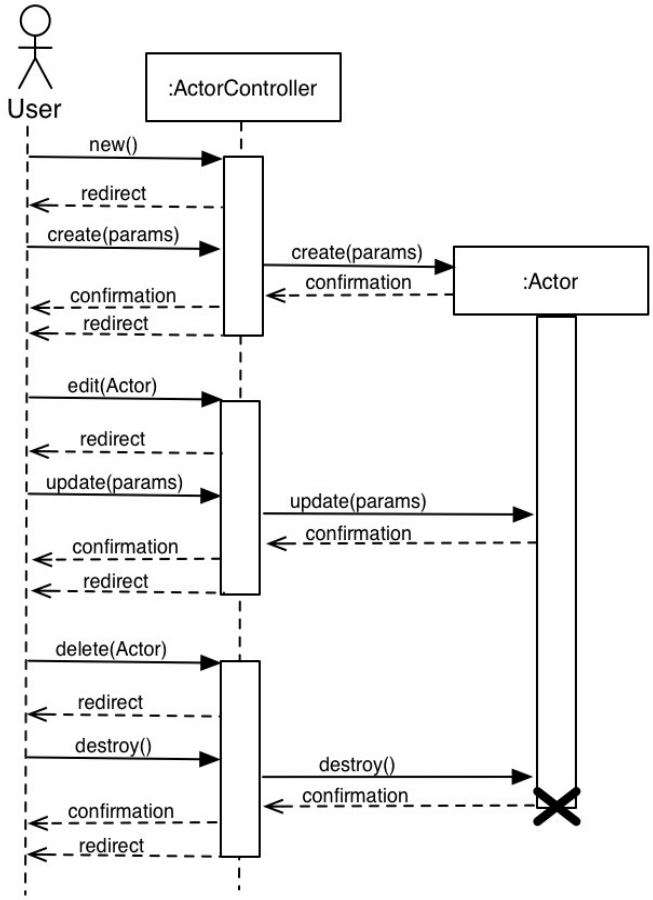
- *(kein UML-Diagramm)*

Wie so manches Web-Framework verwendet auch Ruby on Rails das Software-Architektur Pattern MVC. Wir haben uns bei der Implementierung strikt an dieses Pattern gehalten. Die Funktionsweise wird im Abschnitt Use Cases genauer erläutert.

Use Cases

Datenmanipulation

Bei den allermeisten Use Cases der Applikation handelt es sich jeweils um Manipulationen der Datenbank durch den Benutzer mit Operationen auf den Objekten der Klassen Benutzer (User; Hinzufügen, Editieren), Akteure (Actor; Hinzufügen, Editieren, Löschen) und Beziehungen (Relationship; Hinzufügen, Editieren, Löschen) handelt. Der Admin kann zudem für Objekte der Klassen Akteurtyp (ActorType), InformationTyp (InformationType) sowie Beziehungstyp (RelationshipType) dieselben Operationen ausführen. Sie funktionieren alle nach dem Schema:

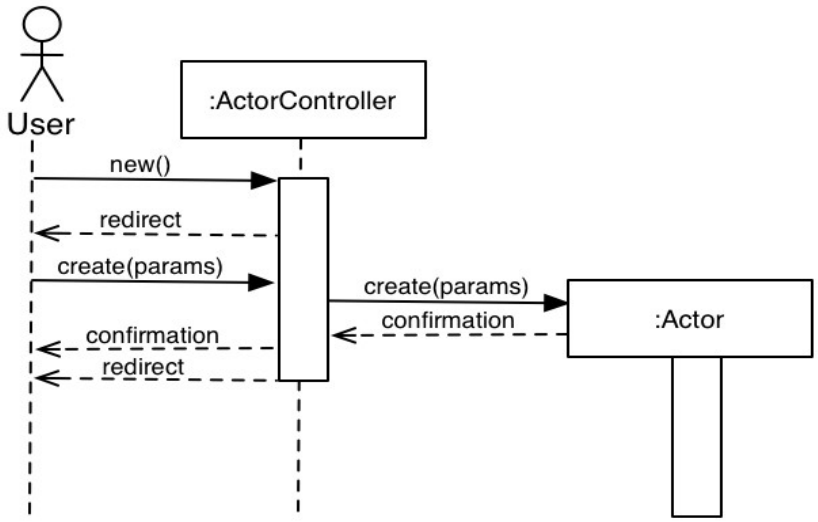


- *Sequenzdiagramm Datenmanipulation*

Erläuterung:
Jede Model-Klasse (bis auf die Klasse Information*) besitzt getreu dem MVC-Pattern eine Controller-Klasse, welcher als Bindeglied zwischen dem View (dem User-interface) und der Model-Klasse selbst agiert. Im Sequenzdiagramm werden die Operationen Hinzufügen (new()), Editieren (edit()) und Löschen (delete()) am Beispiel der Klasse Akteur (Actor) dargestellt.

Die Abläufe der einzelnen Operationen im Detail:

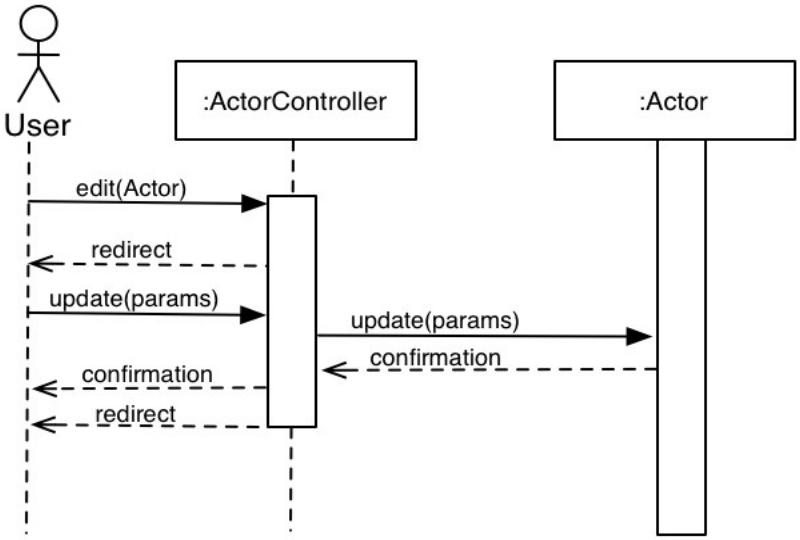
new():



- *Sequenzdiagramm: Hinzufügen eines Akteurs*

1. Der Benutzer ruft durch Klicken auf einen Link/Button die Operation new() auf, welche dann einen Aufruf im ActorController auslöst.
2. Der ActorController leitet den Benutzer (nach Verifizieren seiner Berechtigung) auf eine View mit einem Eingabeformular weiter.
3. Dort kann der Benutzer die benötigten Parameter eingeben und diese anschliessend an den Controller übermitteln.
4. Dieser erstellt nun ein neues Objekt der Model-Klasse 'Actor' gemäss den Parametern.
5. Der Controller erhält von der Klasse über den Erfolg der Manipulation eine Rückmeldung.
6. Das Erfolgen/Nichterfolgen der Manipulation wird dem Benutzer mitgeteilt
7. Abschliessend wird der Benutzer auf eine neutrale View weiterleitet.

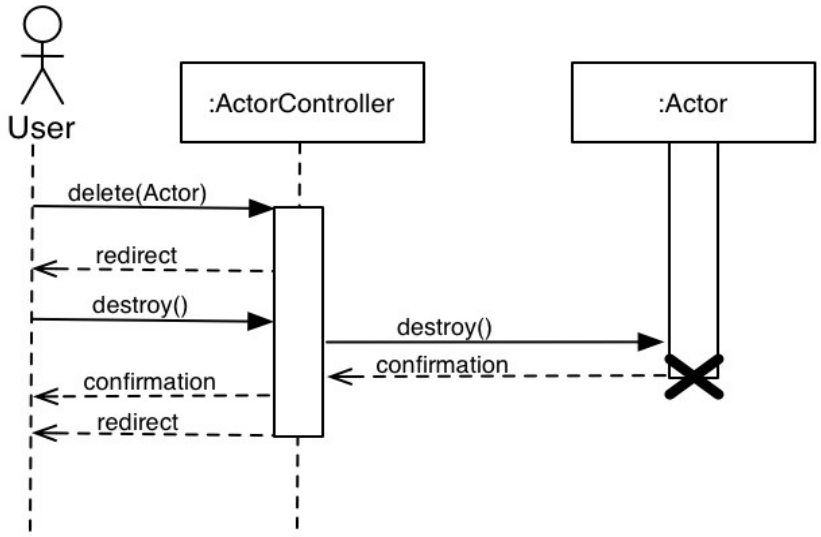
edit(Actor):



- *Sequenzdiagramm: Bearbeiten eines Akteurs*

1. Der Benutzer ruft durch Klicken auf einen Link/Button die Operation edit(Actor) auf, welche dann einen Aufruf im ActorController auslöst.
2. Der ActorController leitet den Benutzer (nach Verifizieren seiner Berechtigung) auf eine View mit einem Eingabeformular weiter.
3. Dort kann der Benutzer die benötigten Parameter eingeben und diese anschliessend an den Controller übermitteln.
4. Dieser aktualisiert nun das Objekt der Model-Klasse 'Actor' gemäss den Parametern.
5. Der Controller erhält von der Klasse über den Erfolg der Manipulation eine Rückmeldung.
6. Das Erfolgen/Nichterfolgen der Manipulation wird dem Benutzer mitgeteilt
7. Abschliessend wird der Benutzer auf eine neutrale View weiterleitet.

delete(Actor):

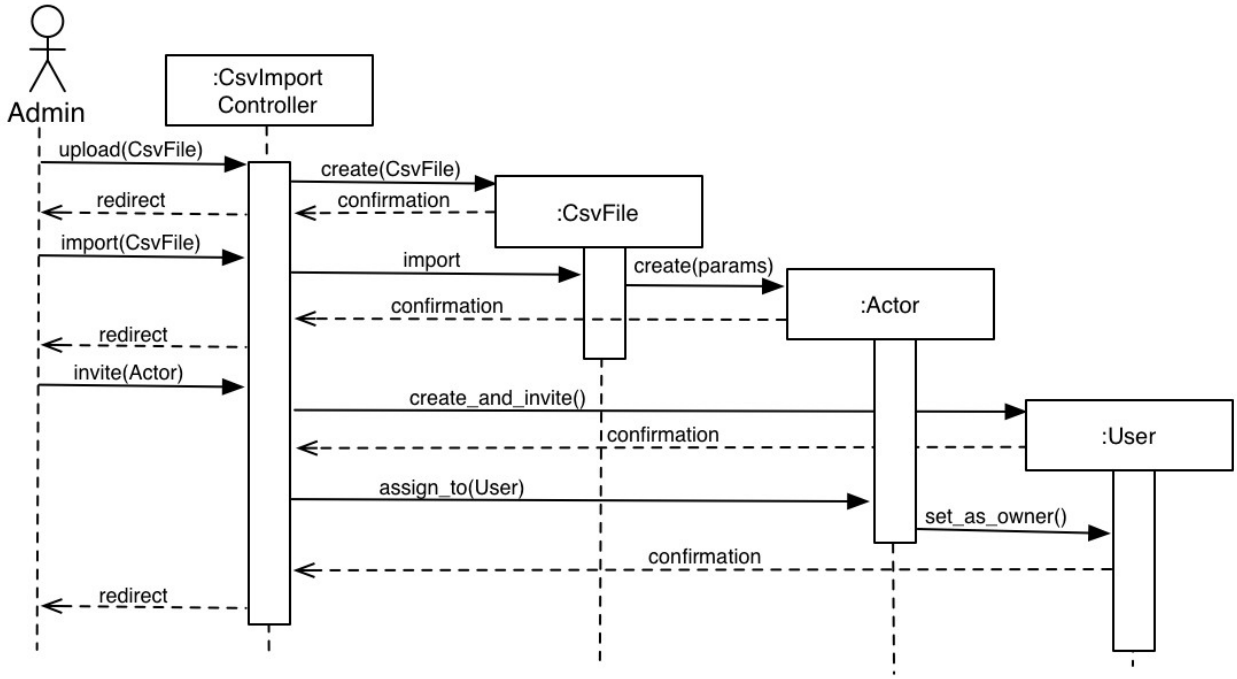


- *Sequenzdiagramm: Löschen eines Akteurs*

1. Der Benutzer ruft durch Klicken auf einen Link/Button die Operation delete(Actor) auf, welche dann einen Aufruf im ActorController auslöst.
2. Der ActorController leitet den Benutzer (nach Verifizieren seiner Berechtigung) auf eine View mit einem Bestätigungsdialog weiter.
3. Dort muss der Benutzer den Vorgang bestätigen und anschliessend an den Controller übermitteln.
4. Dieser zerstört nun das Objekt der Model-Klasse 'Actor'.
5. Der Controller erhält von der Klasse über den Erfolg der Manipulation eine Rückmeldung.
6. Das Erfolgen/Nichterfolgen der Manipulation wird dem Benutzer mitgeteilt
7. Abschliessend wird der Benutzer auf eine neutrale View weiterleitet.

Datenimport

Etwas komplizierter ist der Vorgang beim Import von CSV-Dateien. In diesem Use Case importiert ein Admin eine Liste mit z.B. Einträgen von Ärzten in Verbindung mit ihren E-Mailadressen. Für jeden dieser Eintrag-Einheiten (z.B. pro Arzt) wird dann ein User mit der E-Mailadresse und ein Akteur mit den restlichen Daten erstellt. An die E-Mailadresse wird ein Link geschickt, damit sich der User einloggen kann und die Angaben seines Akteurs überprüfen und erweitern kann. Im folgenden Schema wird der import für eine CSV-Datei der Einfachheit mit genau einem Akteur dargestellt:



- *Sequenzdiagramm Datenimport*

1. Der Admin lädt über den CsvImportController eine CSV-Datei hoch.
2. Der CsvImportController erstellt eine solche Datei (bzw. Objekt) in der Datenbank
3. Das CsvFile-Objekt bestätigt dem Controller das Erstellen.
4. Der Admin wird auf eine Seite weitergeleitet, wo die importierte Datei (das eben erstellte Objekt) angezeigt wird.
5. Der Admin klickt für jenes Objekt auf 'Importieren'.
6. Der Controller führt die Import-Operation durch.
7. Jeder Actor, der im CsvFile enthalten ist, wird erstellt.
8. Jeder Actor bestätigt dem Controller seine Erstellung.
9. Der Admin wird nun auf eine Seite weitergeleitet, wo alle durch den Import erstellten Akteure aufgelistet werden.
10. Der Admin klickt für den entsprechenden Actor auf 'Einladen'.
11. Der Controller erstellt nun zu jedem Actor einen neuen User.
12. Jeder neue User bestätigt dem Controller seine Erstellung.
13. Der Controller weist nun jedem Actor den User zu, welcher ihn besitzt.
14. Jeder Actor setzt sich seinen Besitzer (User).
15. Der Abschluss des Vorgangs wird dem Controller mitgeteilt.
16. Der Admin wird auf eine Seite, wo er über den Abschluss der Operation informiert wird, weitergeleitet.

Erläuterung: Die Schritte 7, 8, 10, 11, 12, 13, 14, 15 werden einfach jeweils entsprechend der Anzahl Akteure oft wiederholt.

Alle Grafiken, welche in diesem Dokument vorkommen, befinden sich im Verzeichnis diagrams/ in besser Qualität.

Quellcode-Dokumentation

Die Quellcode-Dokumentation befindet sich in den Quellcode-Dateien sowie im Verzeichnis doc/. Dort einfach auf irgend eine HTML-Datei klicken.

Benutzer-Handbuch

Das Benutzerhandbuch befindet sich in Form einer Installationsanleitung in der Datei INSTALL.md.