Frank Weigmann

Entwurf und Implementierung eines *symfony*-Plugins zur Synchronisierung von Daten zwischen *OpenERP* und einer Webanwendung

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA (FH)

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informationstechnologie & Elektrotechnik

Mittweida, Juli 2009

Frank Weigmann

Entwurf und Implementierung eines *symfony*-Plugins zur Synchronisierung von Daten zwischen *OpenERP* und einer Webanwendung

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informationstechnologie & Elektrotechnik

Mittweida, Juli 2009

Erstprüfer: Prof. Dr.-Ing. Frank Zimmer
Zweitprüfer: Dipl.-Wirtsch.-Inf. Tim Heil

Vorgelegte Arbeit wurde verteidigt am:

Bibliografische Beschreibung:

Frank Weigmann:
Entwurf und Implementierung eines *symfony*-Plugins zur Synchronisierung von Daten zwischen *OpenERP* und einer Webanwendung 2009. - 85 S.
Mittweida, Hochschule Mittweida (FH) - University of Applied Sciences, Fachbereich Informationstechnologie & Elektrotechnik, Diplomarbeit, 2009

Referat:

Ziel der Diplomarbeit ist es, eine Möglichkeit zum Datenaustausch zwischen der Webanwendung *Jobmixer.com* und dem ERP-System *OpenERP* zu entwerfen und zu implementieren. Dabei wird auf vorhandene Komponenten der beiden Systeme zurückgegriffen. Die Synchronisation wird in Form eines *symfony*-Plugins zur Verfügung gestellt, welches direkt mit *Jobmixer.com* verbunden ist. Der Zugriff auf die Daten von *OpenERP* geschieht über dessen *XML-RPC*-Schnittstelle. Zum Speichern und Auslesen der Daten des Plugins und von *Jobmixer.com* wird *Doctrine* verwendet.

Desweiteren wird auf die Unterschiede in der Datenstruktur von *Jobmixer.com* und *OpenERP* und deren Ausgleich eingegangen. Zu diesem Zweck wird eine Struktur zum Mapping der Daten entwickelt und außerdem aufgezeigt, wie die Synchronisation konfiguriert und durchgeführt wird.

Danksagung

Für die Unterstützung bei meiner Diplomarbeit möchte ich mich bei folgenden Personen und Institutionen herzlich bedanken:

Prof. Dr.-Ing. Frank Zimmer von der Hochschule Mittweida, für die Betreuung meiner Diplomarbeit.

Meinem Betreuer von der *IN AUDITO GmbH* Dipl.-Wirtsch.-Inf. Tim Heil, M. Sc. in Comp. Sc. Thomas Kappel, Dipl.-Ing. (FH) Sebastian Schulze und M. A. Conny Ledwig für deren Hinweise, Anregungen, Kritiken und für das Korrekturlesen dieser Arbeit.

Der *IN AUDITO GmbH*, für die Möglichkeit, diese Diplomarbeit zu schreiben und die Unterstützung, die sie mir gegeben hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous Javascript and XML
CRUD	Create Retrieve Update Delete
DBAL	Database-Abstraction-Layer (Datenbankabstraktionsschicht)
ERM	Entity-Relationship-Model
ERP	Enterprise-Resource-Planning
GPL	GNU General Public License
HTTP	Hypertext Transfer Protocol
MVC	Model-View-Control
ORM	Object-Relational Mapper
PDO	PHP-Data-Objects (PHP-Datenobjekte)
PEAR	PHP Extension and Application Repository
PHP	PHP: Hypertext Preprocessor
RPC	Remote-Procedure-Call
SQL	Structured Query Language
SSH	Secure Shell
SVN	Subversion
URL	Uniform Ressource Locator
XLIFF	XML Localization Interchange File Format
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Kapitel 1

Einleitung

Heutige Geschäftsprozesse werden immer umfangreicher und komplexer. Um den daraus entstehenden Anforderungen an die Personalverwaltung, die Verkaufsabwicklung oder die Buchhaltung gerecht zu werden, gibt es eine vielschichtige Auswahl verschiedener ERP[1]-Systeme. Diese helfen beim Erledigen der Geschäftsprozesse oder übernehmen sie komplett. Bei der Erfüllung dieser Aufgaben kann es notwendig sein, mehr als ein einziges Produkt zu nutzen. Aus diesem Grund befinden sich häufig heterogene Software-Landschaften und somit eine Mischung verschiedener Systeme im Einsatz. Damit eine Zusammenarbeit dieser verschiedenen Systeme funktioniert, ist eine Kommunikation zwischen ihnen erforderlich.

Die Ausgangssituation der *IN AUDITO GmbH* besteht darin, dass ein ERP-System eine Webanwendung bei der Abwicklung des Verkaufsprozesses unterstützen sollt. Das ERP-System und die Webanwendung stehen vollständig implementiert zur Verfügung. Es existiert jedoch keine Möglichkeit, Daten gezielt zwischen den beiden Systemen auszutauschen. Das Ziel dieser Diplomarbeit wird daher sein, eine Möglichkeit zum Datenaustausch zwischen dem ERP-System und der Webanwendung zu entwerfen und diese zu implementieren.

Um die Anforderungen für den Entwurf der Synchronisation zu erhalten, wird im Kapitel „Anforderungen“ der Verkaufsprozess analysiert. Für diese Anforderungen wird im Kapitel „Entwurf“ eine optimale Umsetzung gesucht. Dazu wird auf die verschiedenen Besonderheiten und Unterschiede der beiden Systeme und deren mögliche Kombination genauer eingegangen. Die Umsetzung des Entwurfs wird in „Implementierung“ behandelt. Dabei wird im Detail darauf eingegangen, wie die einzelnen Komponenten der bei-

[1] Enterprise Resource Planning (ERP)

den Systeme zusammenarbeiten und genutzt werden. Zum Abschluss werden noch die Tests für die Synchronisierung im Kapitel „Test der Synchronisation“ betrachtet. Zu diesem Zweck wird die Testumgebung erläutert und Beispiele für die Umsetzung der Tests werden aufgezeigt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Themen behandelt, die für das grundlegende Verständnis dieser Diplomarbeit notwendig sind. Es werden das ERP-System *OpenERP* und die Webanwendng *Jobmixer.com* genauer erläutert. Auf die Technologien und Schnittstellen die in dieser Diplomarbeit Verwendung finden, wird ebenfalls detaillierter eingegangen.

2.1 Das freie ERP-System *OpenERP*

OpenERP ist eine Open-Source ERP-Lösung. Entwickelt und herausgegeben wird es von der aus Belgien stammenden Firma Tiny sprl unter der GPL[2] v3.0[3] Lizenz. Die derzeit aktuelle *OpenERP* Version ist 5.0.1[4]. Zielgruppe für *OpenERP* sind kleine und mittelständische Unternehmen. *OpenERP* unterstützt diese in folgenden Aufgabenbereichen:

- Finanzbuchhaltung,
- Warenwirtschaft,
- Lagerhaltung,
- Produktion,
- Kundenbeziehungsmanagement,
- Projektmanagement.

[2] GNU General Public License (GPL)
[3] *OpenERP* Lizenz: http://doc.openerp.com/license.html#license-link
[4] Stand 28. 05. 2009

Als besondere Merkmale von *OpenERP* sind zu nennen:

- modularer Aufbau,
- Mehrbenutzerfähigkeit,
- durchgängige Mehrsprachigkeit.

OpenERP ist ein Client-Server-System. Das bedeutet, dass es einen zentralen *OpenERP*-Server gibt, der alle Aufgaben übernimmt, sowie einen Client, mit dem Benutzer auf *OpenERP* zugreifen können. Die Installation des Servers ist unter Linux und Windows möglich. Entsprechende Client-Software steht für Linux, Mac OS X und Windows bereit.

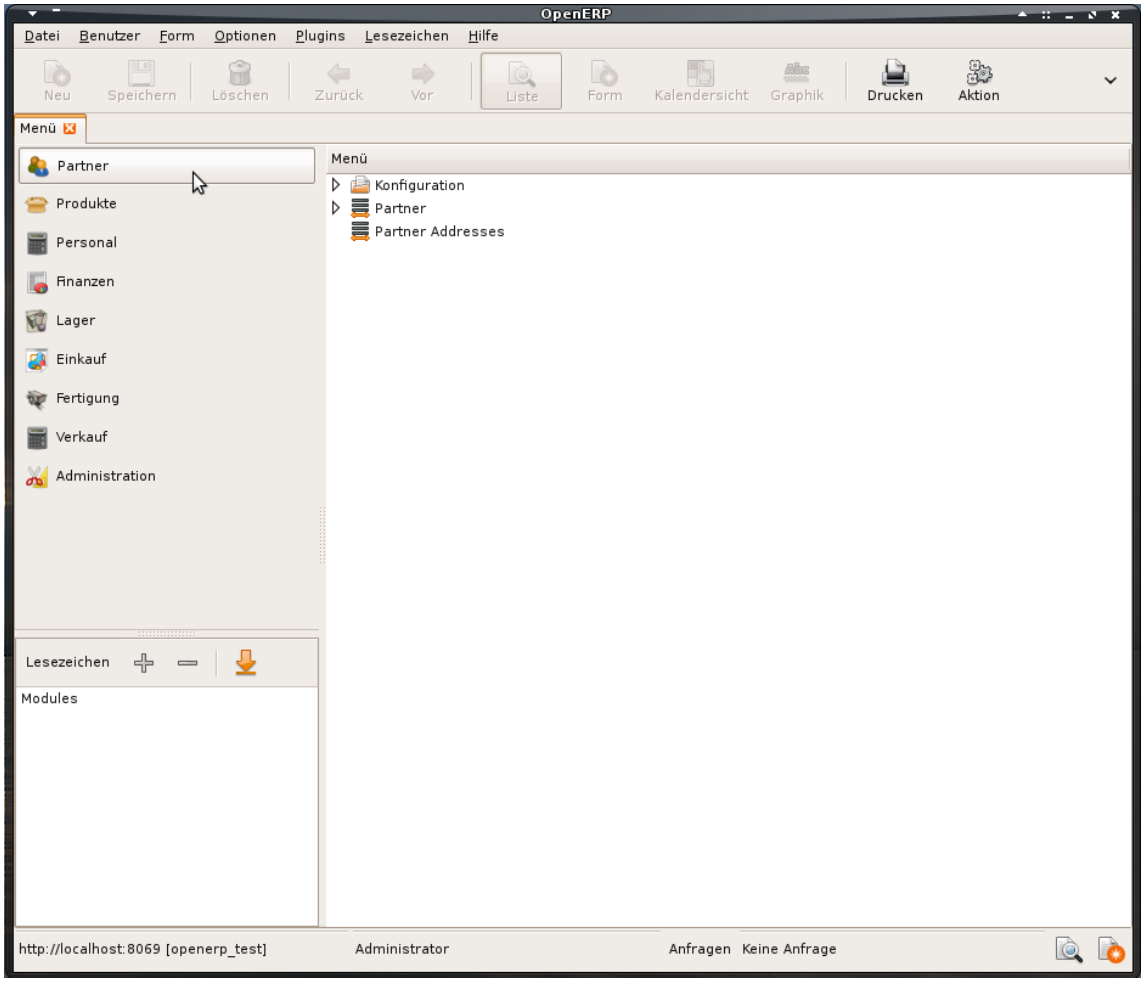

Abbildung 2.1: *OpenERP*-Client unter Linux

Der *OpenERP*-Server ist modular aufgebaut. Er setzt sich aus einem Basismodul und den für die verschiedenen Anforderungen benötigten Modulen zusammen. Das Basismodul

stellt die Grundfunktionen für *OpenERP* zur Verfügung, z. B. die Verwaltung der Kunden des Unternehmens. Weitere Module existieren für die auf Seite 3 bereits genannten Aufgabenbereiche.

Programmiert wurde *OpenERP* in der Programmiersprache Python 2.6, als Datenbank kommt PostgreSQL 8.3 zum Einsatz. Als Besonderheit ist zu erwähnen, dass die in Python erstellten Klassen gleichzeitig das Datenmodell bilden, das heißt, sie bilden die Struktur der Datenbank ab.

2.2 Die Webanwendung *Jobmixer.com*

Jobmixer.com[5] ist das Bewerberportal mit Unternehmensanbindung der *IN AUDITO GmbH*. Die Webanwendung bietet Dienste für private Nutzer an, z. B. die Erstellung individueller Bewerbungsunterlagen, mit denen sich die Nutzer direkt auf die auf der Plattformausgeschriebenen Stellen sowie auf jedes andere z. B. im Internet oder in Privatmedien inseriertes Jobangebot bewerben können. Zielgruppe der Plattform sind, angefangen bei Schülern und Studenten über Absolventen und Arbeitsuchende bis hin zu Fach- und Führungskräften alle Nutzer, die sich im Berufsleben befinden oder gerade in dieses starten. *Jobmixer.com* ist seit September 2006 mit der Version 2.0 online und wird ständig verbessert und um neue Funktionen erweitert. Die Anbindung von *OpenERP* soll mit Version 2.3 erfolgen.

Realisiert wurde *Jobmixer.com* mit dem PHP[6]-Framework *symfony*. Als ORM[7] kommt *Doctrine* zum Einsatz.

2.3 Das Webframework *symfony*

Symfony ist ein auf PHP basierendes Framework, das seit 2003 von Fabien Potencier für sensiolabs[8] entwickelt wird. Im Jahr 2005 wurde es unter der MIT-Lizenz[9] als Open-Source veröffentlicht. Da es zu diesem Zeitpunkt bereits seit 2 Jahren entwickelt wurde, war es schon sehr weit fortgeschritten und es fanden sich schnell aktive Nutzer.

[5] http://www.jobmixer.com
[6] PHP: Hypertext Preprocessor (PHP)
[7] Object-Relational Mapper (ORM)
[8] http://www.sensiolabs.com
[9] *symfony* Lizenz: http://www.symfony-project.org/license

Dies führte dazu, dass *symfony* auch bei großen Internetanwendungen eingesetzt wurde, wie z. B. `del.icio.us`[10] und `Yahoo Bookmarks`[11](Quelle [7]).

Der Ursprung von *symfony* liegt in einer Abspaltung vom *Mojavi-Framework*[12]. Die derzeit aktuelle Version von *symfony* ist 1.2.7[13].

Funktionalität von *symfony*

Symfony setzt auf das MVC[14] Entwurfsmuster und wurde von *Ruby on Rails*[15] inspiriert. Genau wie sein Vorbild unterstützt *symfony* den Entwickler mit verschiedenen Werkzeugen. Auf einige der Werkzeuge, auf die in dieser Diplomarbeit Bezug genommen wird, wird im Folgenden genauer eingegangen.

symfony-Tasks

Ein *symfony*-Task ist ein Werkzeug, das es ermöglicht, sich wiederholende Wartungsaufgaben, Datenbankoperationen oder andere Aufgaben auf der Kommandozeile auszuführen. Jeder Task hat einen Namensraum, einen Namen, eine Kurz- und eine Langbeschreibung. Wenn ein Task aufgerufen wird (Listing 2.1), ist es möglich, ihm Argumente und Optionen als Parameter zu übergeben.

```
1 php symfony <Namensraum>:<Name> <Argumente> <Optionen>
```

Listing 2.1: Aufruf eines *symfony*-Tasks

Der Namensraum dient zur Sortierung und Einordnung der Tasks. Durch Aufruf von `php symfony` auf der Kommandozeile werden alle in einem *symfony*-Projekt gefundenen Tasks nach Namensraum geordnet und sortiert angezeigt. Der Name ist der Bezeichner eines Tasks, mit dessen Hilfe jener aufgerufen werden kann (Listing 2.1).

```
1 php symfony help <Namensraum>:<Name>
```

Listing 2.2: Aufruf der Hilfe eines *symfony*-Tasks

[10] http://delicious.com/
[11] http://bookmarks.yahoo.com/
[12] http://www.mojavi.org/
[13] Stand 28. 05. 2009
[14] Model View Control (MVC)
[15] http://rubyonrails.org/

Die Kurzbezeichnung wird ebenfalls bei dem Aufruf von `php symfony` angezeigt. Wenn die Hilfe für einen Task, wie in Listing 2.2 dargestellt, aufgerufen wird, bekommt der Benutzer die Langbeschreibung angezeigt.

Argumente und Optionen können Standardwerte besitzen, die als `optional` oder `required` gekennzeichnet werden Für jene kann ein Hilfetext angegeben werden. Wie in Listing 2.1 gezeigt, werden die Argumente vor den Optionen angegeben.

```
1  class einSymfonyTask extends sfBaseTask
2  {
3    protected function configure()
4    {
5      $this->namespace        = 'openerp';                /* Angabe des Namensraum */
6      $this->name             = 'synchronisiere';         /* Angabe des Namen */
7      $this->briefDescription = 'die Kurzbeschreibung';   /* Angabe der
           Kurzbeschreibung */
8
9      /* Angabe der Langbeschreibung */
10     $this->detailedDescription = <<<EOF
11 die Langbeschreibung.
12 EOF;
13
14     /* Angabe eines Argumentes */
15     $this->addArgument('argument', sfCommandArgument::OPTIONAL, 'Argument', 'hello');
16
17     /* Angabe einer Option */
18     $this->addOption('option', null, sfCommandOption::PARAMETER_OPTIONAL, 'Option',
19                      'world');
20
21     /* Einbindung der Datenbank */
22     $this->addOption('env', null, sfCommandOption::PARAMETER_REQUIRED,
23                      'The environment', 'dev');
24     $this->addOption('connection', null, sfCommandOption::PARAMETER_REQUIRED,
25                      'The connection name', 'doctrine');
26   }
27
28   protected function execute($arguments = array(), $options = array())
29   {
30     /**
31      * <Programmcode fuer den Task>
32      */
33   }
34 }
```

Listing 2.3: Aufbau eines *symfony*-Tasks

Listing 2.3 zeigt den Aufbau eines Tasks. Alle Angaben, die vorher besprochen wurden, können in der `configure()` Methode angegeben werden. Der auszuführende Programmcode wird in die `execute()` Methode geschrieben.

Zur Einbindung einer Datenbankverbindung müssen die Umgebung (`env`) und die Verbindung (`connection`) als Optionen übergeben werden.

symfony-Plugins

Ein *symfony*-Plugin dient dazu, den *symfony*-Kern zu erweitern und diesen dann in verschiedenen *symfony*-Anwendungen zu verwenden. Dabei können alle Funktionen, die in einem *symfony*-Projekt bereit stehen, in einem Plugin erzeugt werden, z. B. Templates, Datenbankmodels usw. Die Nutzung eines Plugins kann anwendungsweise aktiviert oder deaktiviert werden. Um ein Plugin zu installieren, existieren zwei Möglichkeiten:

Die erste Möglichkeit ist die Installation unter Verwendung von PEAR[16]. Das hat den Vorteil, dass Plugins auf einfachem Weg installiert, aktualisiert und wieder entfernt werden können. Die Installation wird dabei mit Hilfe eines *symfony*-Tasks durchgeführt, der die Pakete aus dem *symfony*-PEAR-Kanal[17] bezieht (Listing 2.4).

```
1 php symfony plugin:install <pluginName>
```

Listing 2.4: Installation eines Plugins aus dem *symfony* PEAR-Kanal

Durch die Übergabe der entsprechenden Parameter an den Task kann jener Plugins auch aus einem heruntergeladenen PEAR-Paket (Listing 2.5) oder aus einem anderem PEAR-Kanal (Listing 2.6) installieren.

```
1 php symfony plugin:install </Pfad/zum/heruntergeladenem/Paket/pluginName.tgz>
```

Listing 2.5: Installation eines Plugins aus einem PEAR-Paket

```
1 php symfony plugin:add-channel channel.symfony.pear.example.com
2 php symfony plugin:install <pluginName>
```

Listing 2.6: Installation eines Plugins aus einem anderen PEAR-Kanal

Die zweite Möglichkeit, ein *symfony*-Plugin zu installieren, ist, das Plugin direkt in das `plugin`-Verzeichnis des *symfony*-Projektes zu kopieren. Dazu kann es entweder aus einem Archiv entpackt oder aus einem Versionskontrollsystem, z. B. SVN[18], ausgecheckt werden. Wenn ein Plugin auf diesem Weg installiert wird, muss auch das Aktualisieren und Löschen manuell vom Benutzer erledigt werden.

[16] PHP Extension and Application Repository (PEAR)
[17] Der *symfony* PEAR-Kanal: plugins.symfony-project.org
[18] Subversion (SVN)

Für beide Möglichkeiten der Installation müssen für das Plugin individuell notwendige Anpassungen an den *symfony* Konfigurationsdateien vorgenommen werden.

Der Aufbau der Ordnerstruktur eines Plugins orientiert sich an der eines *symfony*-Projektes. Die einzige Ausnahme im Vergleich zu einem *symfony*-Projekt-Verzeichnis ist der `apps` Ordner.

Konfiguration mit YAML-Dateien

Die Konfigurationsdateien von *symfony* sind in der YAML[19]- Syntax geschrieben. YAML ist eine Sprache zur Datenserialisierung, die allerdings keine Tags besitzt. Dadurch lassen sich die Konfigurationsdateien sehr einfach strukturieren und sind für Menschen sehr gut lesbar. Der folgende Abschnitt zeigt, was beim Erstellen einer in YAML geschriebenen Datei zu beachten ist.

```
1 Oberpunkt:
2   Unterpunkt: <Inhalt des Unterpunktes>
```

Listing 2.7: Zuordnung von Ober- und Unterpunkten in einer YAML-Datei

Die Zuordnung der einzelnen Elemente funktioniert mithilfe von Doppelpunkten und Einrückungen. In Listing 2.7 ist die korrekte Zuordnung von Werten zum Unterpunkt und die Zuordnung vom Unter- zum Oberpunkt dargestellt. Bei der Formatierung sind keine Tabs erlaubt. Die Einrückungen dürfen nur mit Leerzeichen gemacht werden.

```
1 # ein Kommentar
2
3 ein_YAML_array_beispiel:
4   # Angabe eines normalen Literals
5   punkt_1:  Literal
6   # Literal mit zwei extra Leerzeichen zu Beginn
7   punkt_2:  '  Literal'
8   wahrheitswert_true:   true  #[ on, 1, true, yes ]
9   wahrheitswert_false:  false #[ off, 0, false, no ]
```

Listing 2.8: Notation von Werten

Die Angabe von Werten erfolgt in Literalen, als Wahrheitswert oder als Array. Ein normales Literal wird einfach hinter den Unterpunkt geschrieben. Wenn es eine spezielle Formatierung besitzt, z. B. ein oder mehrere Leerzeichen vor Beginn des Literals, wird

[19] YAML Ain't Markup Language (YAML)

es in einfachen Hochkommas geschrieben (Listing 2.8). Wahrheitswerte können wie in Listing 2.8 beschrieben angegeben werden. Dabei ist es dem Entwickler frei gestellt, in welcher der drei Möglichen er einen Wahrheitswert angibt.
Das YAML-Format unterstützt einfache und assoziative Arrays. Wie diese notiert werden, zeigt Listing 2.9.

```
1 ein_YAML_array_beispiel:
2   array_syntax_1:  [Wert1, Wert2. Wert3]
3   array_syntax_2:
4     - Wert1
5     - Wert2
6     - Wert3
7  assozatives_array:  [Schluessel1: Wert1, Schluessel2: Wert2, Schluessel3: Wert3]
```

Listing 2.9: Notation von einfachen und assoziativen Arrays

Automatisches Laden von Klassen

Die Autoload[20]-Funktion von *symfony* lädt automatisch die benötigten Klassen zur richtigen Zeit. Das bedeutet für den Entwickler, dass er den Ordner, in dem sich die Datei mit der entsprechenden Klasse befindet, nur in die `autoload.yml` Datei des *symfony*-Projektes eintragen muss. Listing 2.10 zeigt den Aufbau der in YAML Syntax verfassten Datei `autoload.yml`.

```
 1 autoload:
 2   # Label
 3   autoload_label:
 4     # Name
 5     name:           plugins lib
 6     # absoluter Pfad
 7     # Pfad Konstanten muessen mit "%" Beginnen und Enden
 8     path:           %SF_PLUGINS_DIR%/*/lib
 9     # Einstellung ob Unterverzeichnisse mit durchsucht werden sollen
10     recursive:      on
11     # Welche Verzeichnisse sollen ausgelassen werden
12     exclude:        [model, symfony]
13     # hiermit kann angegeben werden das der Autolaoder auf Dateien
14     # mit anderen Endungen laden soll
15     ext:          .inc
```

Listing 2.10: Notation der Einträge in der autoload.yml

Beim ersten Aufruf des Autoloaders geht dieser alle in der Datei `autoload.yml` eingetragenen Verzeichnisse durch und trägt die passenden Klassen z. B. mit der Endung `.php`,

[20] Übersetzt: Automatisches Laden.

in eine interne Liste ein. Die Liste befindet sich in der Datei `config/config_autoload.yml` welche sich im Zwischenspeicher befindet. Wenn während der Laufzeit eine Klasse aus einer der Dateien benötigt wird, fügt der Autoloader diese automatisch ein. Dazu wird das PHP-Autoloading verwendet, welches durch die `sfAutoload`-Klasse von *symfony* gekapselt wird.

Unit Tests von *symfony*

Unit Tests testen einzelne Methoden und überprüfen für eine Eingabe, ob die korrekte Ausgabe zurückgegeben wird. Da immer nur eine Eingabe überprüft werden kann, aber es verschiedene Fälle gibt, z. B. einen falschen Übergabewert, muss eine Methode mehrfach mit verschieden Parametern getestet werden. Nur so kann die korrekte Funktionalität sichergestellt werden.

Das Test-Framework von *symfony* wird `lime` genannt. Es besteht aus einer einzigen PHP-Datei, die keine Abhängigkeiten besitzt. Das Test-Framework kann daher auch außerhalb von *symfony* verwendet werden. Die Dateien in denen die Tests geschrieben sind, werden in das `/test/unit`-Verzeichnis des *symfony*-Projektes abgelegt.

Listing 2.11 zeigt den Aufbau einer Testdatei. Zu Beginn wird die Datei `unit.php` geladen. In dieser Datei werden alle Vorbereitungen für die Tests getroffen. Es wird z. B. Lime geladen und eine Datenbankverbindung zur Verfügung gestellt. Danach wird eine Instanz von Lime erzeugt. Dieser muss die Anzahl der Tests die durchgeführt werden sollen und eine Instanz der Klasse `lime_output_color` übergeben werden. Danach können die einzelnen Tests geschrieben werden.

```
1 <?php
2 include(dirname(__FILE__).'/../bootstrap/unit.php');
3
4 $t = new lime_test(<Anzahl der durchzufuehrenden Tests>, new lime_output_color());
5 /**
6  *<Programmcode fuer die Tests>
7  */
```

Listing 2.11: Aufbau eines Unit Tests

Weitere Werkzeuge

Weitere Werkzeuge, die *symfony* zur Unterstützung des Entwicklers anbietet, sind:

- Hilfe beim Realisieren von AJAX[21] gestützten Anwendungen unter Verwendung der Javascript-Bibliotheken *Prototype*[22] und *script.aculo.us*[23],
- Scaffolding zum schnellen Erstellen von Masken, die CRUD[24]-Funktionen bereitstellen,
- Internationalisierung über *XLIFF*[25] oder *Gettext*.

2.4 *Doctrine*

Doctrine ist ein ORM, der für PHP 5.2.3 und höher entwickelt wurde und unter der LGPL[26] Lizenz herausgegeben wird. Die Aufgabe von *Doctrine* ist es, die Tabellen einer Datenbank als Objekte abzubilden. Wenn neue Objekte erzeugt, bestehende geändert oder gelöscht werden, dann soll *Doctrine* diese Änderungen auf die entsprechende Datenbanktabelle übertragen.

Aufbau von *Doctrine*

Abbildung 2.2 zeigt die Zusammenhänge zwischen den verschiedenen Schichten, die für *Doctrine* eine Rolle spielen. Als Abstraktionsschicht für den Datenbankzugriff werden PDO[27] eingesetzt. Dadurch steht *Doctrine* die Grundlage für den Zugriff auf verschiedene Datenbanksysteme zur Verfügung, da es direkt darauf aufsetzt. *Doctrine* selbst besteht aus zwei Ebenen, einer DBAL[28], welche die PDO erweitert, und dem ORM. Beim Laden der ORM-Schicht wird der DBAL ebenfalls geladen.

[21] Asynchronous Javascript and XML (AJAX)
[22] http://www.prototypejs.org/
[23] http://script.aculo.us/
[24] Create Retrieve Update Delete (CRUD)
[25] XML Localization Interchange File Format (XLIFF)
[26] http://www.gnu.org/copyleft/lesser.html
[27] PHP-Data-Objects (PHP-Datenobjekte) (PDO)
[28] Database Abstraction Layer (Datenbankabstraktionsschicht) (DBAL)

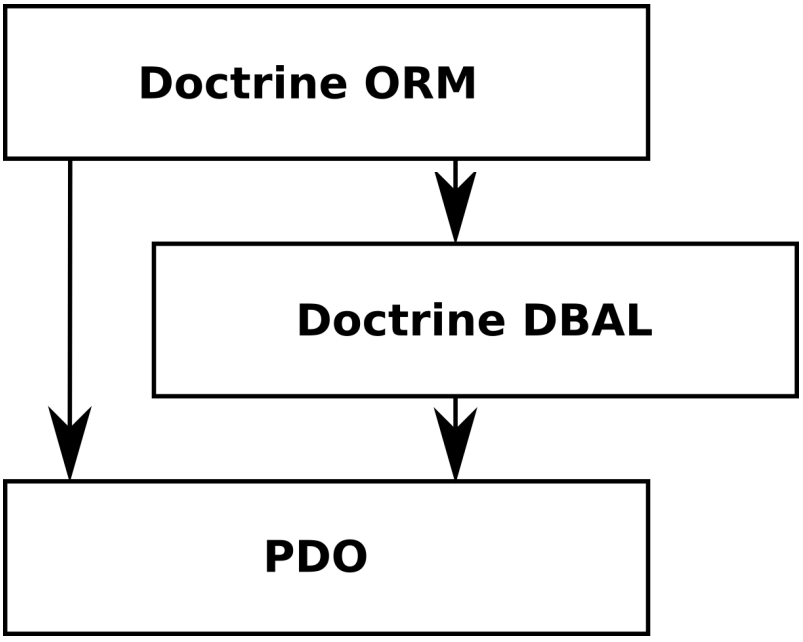

Abbildung 2.2: Die einzelnen Schichten von *Doctrine*. Quelle: [20]

Doctrine ORM
: Brücke zwischen der relationalen Datenbank und den Datenobjekten.

Doctrine DBAL
: Die Datenbankabstraktionsschicht von Doctrine.

PDO
: Die Abstraktionsschicht für den Datenbankzugriff.

Die Entwurfsmuster die bei der Entwicklung von *Doctrine* zum Einsatz kommen sind:

- Active Record,
- Data Mapper,
- Metadata Mapping.

Merkmale von *Doctrine*

Durch die Verwendung von PDO werden von *Doctrine* alle Datenbanksysteme unterstützt, für die PDO-Treiber[29] existieren.

[29] Liste mit allen derzeit verfügbaren PDO-Treibern: http://de.php.net/manual/de/pdo.drivers.php

Datenbankabfragen können mit *Doctrine* in einem objektorientierten SQL-Dialekt geschrieben werden. Dieser Dialekt heißt DQL und ist inspiriert von Hibernates[30] HQL[31]. Weitere Merkmale sind unter anderem[32]:

- Datenbanken können über PHP- oder YAML-Dateien konfiguriert werden,
- Einträge in eine Datenbank können validiert werden,
- *Doctrine* unterstützt die Vererbung von Tabellen.

2.5 *XML-RPC*

XML-RPC (Extensible Markup Language Remote Procedure Call) ist eine Schnittstelle, mit der es möglich ist, Methodenaufrufe zwischen verschiedenen Betriebssystemen und Umgebungen auszuführen. Inspiriert wurde *XML-RPC* von Remote Procedure Call (RPC) und einem frühen Entwurf von SOAP. Entwickelt wurde *XML-RPC* von Dave Winer in Zusammenarbeit mit Microsoft. Implementierungen existieren für alle gängigen Programmiersprachen.

Aufbau von *XML-RPC*

Als Protokoll für den Transport wird HTTP[33] verwendet. Für die Darstellung der übertragenen Daten wird die XML[34] verwendet.

Listing 2.12 zeigt eine *XML-RPC*-Anfrage. Zuerst wird die Version des XML-Dokumentes festgelegt. Als nächstes wird die Art der Übertragung gesetzt. Da in Listing 2.12 eine Anfrage gestellt wird, muss hier der Typ mit `<methodCall>` angegeben werden. Die Sektion `<methodName>` verlangt die Angabe des Namens der aufzurufenden Methode. Wenn der Methode Parameter übergeben werden müssen, können diese mit Datentyp im Abschnitt `<params>` angegeben werden. Bei den in Listing 2.12 angegebenen Parameteren handelt es sich beim ersten um den Typ `string` und beim zweiten um den Typ `int`.

[30] http://www.hibernate.org/
[31] http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html
[32] Liste der *Doctrine*-Merkmale: http://swik.net/Doctrine
[33] Hypertext Transfer Protocol (HTTP)
[34] Extensible Markup Language (XML)

```
<?xml version="1.0"?>
  <methodCall>
    <methodName>Klassenname.Methodenname</methodName>
    <params>
      <param>
        <value>
          <string>Parameter 1</string>
        </value>
      </param>
      <param>
        <value>
          <int>Parameter 2</int>
        </value>
      </param>
    </params>
  </methodCall>
```

Listing 2.12: Aufbau einer *XML-RPC* Anfrage

Bei einer *XML-RPC*-Antwort wird `<methodCall>` durch `<methodResponse>` ersetzt. Der Aufbau des übrigen des XML-Dokumentes bleibt unverändert. In der Sektion `<params>` werden die zu übermittelnden Daten mit Datentyp angegeben. Eine Übersicht über die verfügbaren Datentypen gibt Tabelle 2.1.

Tabelle 2.1: Die verschiedenen Datentypen von *XML-RPC*

Datentyp	Beschreibung
`int`	32 Bit Integer
`string`	eine ASCII Zeichenkette, die auch NULL Bytes enthalten kann
`boolean`	Wahrheitswert True (1) oder False (0)
`double`	Eine Fließkommazahl mit doppelter Genauigkeit.
`dateTime.iso8601`	Datum und Zeit
`base64`	Unverarbeitete Binärdaten verschiedener Längen. Base64 kodiert.
`array`	Ein eindimensionales Array von Werten. Einzelne Felder können jeden anderen Datentyp besitzen.
`struct`	Eine Sammlung von Schlüssel-Wert-Paaren. Die Schlüssel sind vom Datentyp String, die Werte können jeden anderen Datentyp besitzen.

2.6 Konventionen für die Umsetzung

Um den Quellcode für andere Entwickler lesbar zu gestalten, existieren Vorgaben zu dessen Formatierung. Dazu orientieren sich die bei dieser Diplomarbeit verwendeten Konventionen an den Vorgaben für den Quellcode der *IN AUDITO GmbH*.

Wie diese im Detail aussehen und welche Konventionen es noch gibt, z. B. in Bezug auf die Datenbanktabellen, wird in diesem Abschnitt erläutert.

Konventionen für die Namensgebung

Die Namenskonventionen geben vor, wie Methoden und Variablen zu benennen sind. Der gewählte Name einer Methode oder Variable ist kontextbezogen. Das bedeutet, dass bereits Metainformationen im Namen enthalten sind. Dabei muss beachtet werden, dass Informationen die z. B. durch den Klassennamen bereits übermittelt wurden, nicht mehr mit einbezogen werden.

Das Vergeben von Klassennamen, Methodennamen und Variablennamen geschieht im Quellcode von *Jobmixer.com* vollständig in englischer Sprache. Das Gleiche gilt auch für den Quellcode in dieser Diplomarbeit, da eine einheitliche Sprache bei der Namensgebung für die Lesbarkeit wichtig ist.

Konventionen für Methodennamen

Methodennamen bestehen aus einem Verb als Präfix, das zeigt wie die Methode verwendet werden soll[35], und aus einem oder mehreren Wörtern die genaueren Aufschluss über die Verwendung der Methode geben. Die Angabe des vollständigen Methodennamens erfolgt in der *CamelCase*-Notation.

An dieser Stelle werden einige Beispiele für Methodennamen und der Bezug zu ihrer Funktionalität aufgezeigt. Methoden, die zum Holen von Daten benutzt werden, haben `get` als Präfix, während Methoden, welche Daten speichern mit `set` beginnen. Wenn eine Methode das Präfix `is` besitzt, hat sie als Rückgabewert einen booleschen Wert. Beginnt der Methodenname mit `has`, überprüft die Methode ob ein Wert gesetzt wurde.

Konventionen für Variablennamen

Die Schreibweise von Variablennamen ist abhängig vom Typ der Variable. Objekte und Arrays werden in der *CamelCase*-Notation geschrieben, während Literale mit der *Underscore*-Notation angegeben werden.

[35] z. B. `get`: Es werden Informationen mithilfe der Methode beschafft und zurückgegeben.

Konstanten werden komplett in Großbuchstaben geschrieben und mehrere Wörtern durch einen Unterstrich getrennt.

Konventionen für Datenbanktabellen

Die Bezeichnung von Datenbanktabellen und Spaltennamen erfolgt in der *Underscore*-Notation. Tabellen, die für das Plugin genutzt werden, bekommen – für eine einfachere Zuordnung – den Präfix `oep`. Spaltennamen, die einen Fremdschlüssel darstellen, tragen den Namen derjenigen Tabelle zu der die Relation besteht, und das Suffix `id`.

Kapitel 3

Anforderungen

Dieses Kapitel behandelt die Anforderungen, die an diese Diplomarbeit gestellt werden. Der Schwerpunkt der Aufgabenstellung besteht darin, das ERP-System *OpenERP* in den Verkaufsprozess einzubinden. Daher wird dieses Kapitel einen Überblick darüber geben, welche Anforderungen erfüllt sein müssen, damit die Anbindung von *Jobmixer.com* an *OpenERP* funktioniert.

3.1 Analyse des Verkaufsprozesses

Um detaillierte Anforderungen für den Austausch der Daten zu erhalten, ist es erforderlich, den Ablauf für den Verkauf eines Produktes genauer zu analysieren. Zu diesem Zweck wird der Verkauf eines Firmenprofils[36] genauer betrachtet. Für den Verkaufsprozess wird vorrausgesetzt, dass der Kunde ein *Jobmixer.com*-Benutzerkonto besitzt und sich mit diesem angemeldet hat. Damit die Produkte und Produktkategorien für den Verkaufsprozess zur Verfügung stehen, müssen diese in *OpenERP* angelegt und zu *Jobmixer.com* übertragen werden. Nachdem dies geschehen ist, können die Produkte auf *Jobmixer.com* nach Produktkategorien sortiert, angezeigt und bestellt werden.

[36] Als Firmenprofil wird die Vorstellung eines Unternehmens auf *Jobmixer.com* bezeichnet. Beispielsweise das Firmenprofil der *IN AUDITO GmbH*: http://www.jobmixer.com/de/firmenprofil/in-audito

3.1.1 Ablauf des Verkaufsprozesses

Zu Beginn des Verkaufsprozesses muss der Kunde ein Produkt auf der Übersichtsseite auswählen. Im hier dargestellten Beispiel wählt der Kunde ein Firmenprofil und füllt es aus. Wenn das Firmenprofil akzeptiert ist[37], wird es im Warenkorb abgelegt und der Kunde erhält eine Benachrichtigung, dass der Bezahlvorgang für das Firmenprofil freigeschaltet wurde. Der Kunde hat jetzt die Möglichkeit, weitere Produkte zu seinem Warenkorb hinzuzufügen, oder die Bestellung abzuschließen.

Durch das Bestätigen des Warenkorbes kann der Kunde den Abschluss der Bestellung einleiten. Dazu wird er aufgefordert, seine Rechnungsdaten anzugeben und die gewünschte Zahlungsart auszuwählen. Sobald dieser Vorgang abgeschlossen ist, wird ihm eine komplette Übersicht über seine Bestellung angezeigt. Nachdem der Kunde diese bestätigt hat, ist die Bestellung abgeschlossen.

Den Status seiner Bestellung und die dazugehörigen Rechnungen werden dem Kunden auf einer separaten Übersichtsseite angezeigt.

Aus dem eben beschriebenen Ablauf lassen sich Anforderungen für die Diplomarbeit ableiten. Für einen besseren Überblick werden diese in zwei Gruppen aufgeteilt. Zum Einen sind das Anforderungen, die *OpenERP* erfüllen muss, damit es die an sich gestellten Aufgaben erfüllen kann. Zum Anderen sind das die Anforderungen an den Austausch der Daten zwischen *OpenERP* und *Jobmixer.com*.

3.1.2 Anforderungen an *OpenERP*

Die erste Anforderung ergibt sich aus der Notwendigkeit, die Produkte und Produktkategorien zu verwalten. Wie oben erwähnt, soll deren Verwaltung mithilfe von *OpenERP* geschehen und danach mit *Jobmixer.com* synchronisiert werden. Das bedeutet für *OpenERP*, dass es die Möglichkeit der Verwaltung bereitstellen muss.

[37] Eine Ablehnung des Firmenprofils kann erfolgen, wenn die Angaben im Firmenprofil gegen die Allgemeinen Geschäftsbedingungen der *IN AUDITO GmbH* verstoßen. In diesem Fall bekommt der Kunde die Möglichkeit, seine Angaben zu korrigieren.

Die zweite Anforderung betrifft das Annehmen und Verarbeiten von Bestellungen. Nachdem der Kunde auf *Jobmixer.com* seine Bestellung bestätigt hat und sie dann an *OpenERP* gesendet wurde, muss die Bestellung aufgenommen und in *OpenERP* angezeigt werden. Wenn die Bestellung in *OpenERP* abgeschlossen ist, muss die dazu passende Rechnung erzeugt werden.

Die dritte Anforderung besteht darin, dass die Kunden in *OpenERP* vorhanden sein müssen, damit ihnen die Bestellungen und Rechnungen zugeordnet werden können.

Zusammengefasst ergeben sich folgende Anforderungen an *OpenERP*. Es muss in der Lage sein:

- Produkte anzulegen und zu verwalten,
- eine Bestellung anzunehmen und diese zu verarbeiten können,
- für jede Bestellung eine Rechnung zur Verfügung zu stellen,
- die Kunden von *Jobmixer.com* aufzunehmen und sie Bestellungen und Rechnungen zuzuordnen.

3.1.3 Anforderungen an den Datenaustausch

Für den Datenaustausch ergeben sich aus dem oben beschriebenen Ablauf und den Anforderungen an *OpenERP* weitere Anforderungen.

Alle Produkte und Produktkategorien die in *OpenERP* angelegt wurden, sollen auch auf *Jobmixer.com* erscheinen. Damit das möglich ist, müssen sie zu *Jobmixer.com* übertragen werden. Dabei darf die Zuordnung von Produkten zu Produktkategorien nicht verloren gehen.

Auf *Jobmixer.com* abgeschlossene Bestellungen müssen zur weiteren Bearbeitung zu *OpenERP* übertragen werden. Wenn die Bestellung in *OpenERP* bearbeitet wurde, soll der Status der Bestellung zurück zu *Jobmixer.com* übermittelt werden. Wie aus den Anforderungen an *OpenERP* hervorgeht, hat jede abgeschlossene Bestellung eine dazugehörige Rechnung. Diese muss ebenfalls *Jobmixer.com* zur Verfügung stehen.

Für die Zuordnung der Bestellungen und Rechnungen zu einem Kunden muss dieser auch in *OpenERP* vorhanden sein. Damit das gewährleistet ist, müssen die Benutzer von *Jobmixer.com* zu *OpenERP* übertragen werden.

Nachfolgende Liste fasst alle Anforderungen für den Datenaustausch zwischen *OpenERP* und *Jobmixer.com* zusammen. Der Datenaustausch muss folgende Daten einbeziehen:

- alle Produkte und Produktkategorien,
- Bestellungen,
- Änderungen des Bestellstatus,
- die erzeugten Rechnungen,
- die Benutzer von *Jobmixer.com* und ihre Daten.

3.2 Umsetzung des Datenaustausches

Die Anforderungen aus Abschnitt 3.1 geben Aufschluss darüber, was notwendig ist, um Daten übertragen und verarbeiten zu können. In diesem Abschnitt werden die Anforderungen für die Umsetzung des Datenaustauschs definiert. Die Anforderungen sollen helfen, den Datenaustausch einfach und schnell zu nutzen, ohne z. B. bei einer Änderung der Datenstruktur umfangreiche Änderungen am Quellcode vornehmen zu müssen.

Um zu verhindern, dass bei jeder Änderung der Datenstruktur für die Synchronisation der Quellcode verändert werden muss, ist eine Konfigurationsdatei erforderlich. Darin werden alle Einstellungen, die zum Anpassen der Synchronisation notwendig sind, abgelegt und dem System bereitgestellt.

Die nächste Anforderung betrifft den Unterschied der Datenstrukturen, die Bezeichnungen und Schlüssel von *Jobmixer.com* und *OpenERP*. Damit es beim Datenaustausch nicht zur Zuordnung von falschen Daten kommt, z. B. eine falsche Zuordnung des Kunden und der dazugehörigen Rechnung, ist ein Mapping der Daten erforderlich. Das heißt, es muss möglich sein, mithilfe eines *Jobmixer.com*-Objektes und dessen Schlüssel, den passenden *OpenERP*-Datensatz herauszufinden. Außerdem müssen für beide Objekte alle Felder für den Datenaustausch bekannt sein.

Um nach erfolgten Anpassungen am Quellcode Fehler bei der Synchronisierung auszuschließen, müssen Tests geschrieben und ausgeführt werden. Diese haben die Aufgabe, die Funktionalität der Synchronisation sicherzustellen.

Zusammengefasst werden an die Umsetzung des Datenaustausches folgende Anforderungen gestellt:

- Einstellungen sollen in einer Konfigurationsdatei änderbar sein,
- Objekte beider Seiten mit allen Feldern müssen einander zuordbar sein,
- das Übertragen der Daten muss auch zeitversetzt möglich sein,
- es müssen Tests für den Datenaustausch bereitgestellt werden.

3.3 Anforderungen an die Sicherheit

Da es sich bei den in *OpenERP* und *Jobmixer.com* gespeicherten Informationen um vertrauliche Daten handelt, ist es notwendig, Sicherheitsvorkehrungen zu treffen, um die Daten vor unbefugtem Zugriff zu schützen. Dies gilt für alle an der Synchronisation beteiligten Komponenten, angefangen bei der Datenbank von *OpenERP*, über den *OpenERP*-Server bis hin zu *Jobmixer.com* und der Kommunikation zwischen den einzelnen Systemen (Abbildung 3.1). Die Sicherheit von *Jobmixer.com* und des *OpenERP*-Servers soll keine Rolle in dieser Diplomarbeit spielen und wird daher als gegeben angesehen. Dieser Abschnitt wird sich ausschließlich mit den Sicherheitsanforderungen an die *OpenERP*-Datenbank und die Kommunikation zwischen den Systemen beschäftigen.

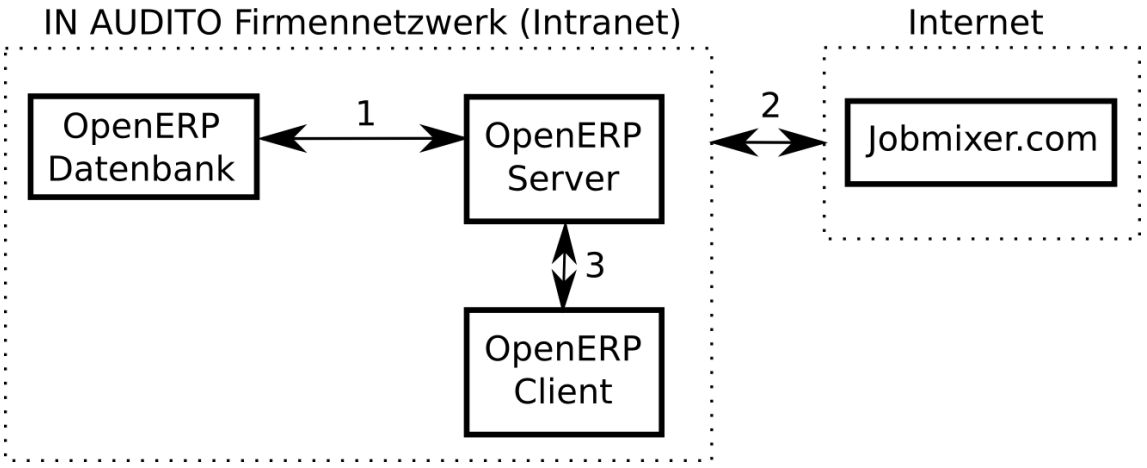

Abbildung 3.1: Überblick über die Kommunikationspartner der Synchronisation

1. Kommunikation zwischen *OpenERP*-Datenbank und *OpenERP*-Server
2. Kommunikation zwischen *OpenERP*-Server und *Jobmixer.com*
3. netzwerkinterne Kommunikation zwischen *OpenERP*-Server und *OpenERP*-Client

3.3.1 Sicherheit der Kommunikation

Die Kommunikation zwischen den Kommunikationspartnern der Synchronisation findet über ungeschützte Verbindungen statt und kann somit abgefangen und ohne große Probleme ausgewertet werden. Um dies zu verhindern, ist eine Absicherung des Datenaustausches notwendig. Besondere Aufmerksamkeit muss dabei der Kommunikation zwischen *Jobmixer.com* und *OpenERP* gewidmet werden, da diese über das Internet geführt wird. Die Kommunikation zwischen *OpenERP* und der *OpenERP*-Datenbank findet nur im geschützen Firmennetzwerk der *IN AUDITO GmbH* statt und kann daher als relativ sicher eingeschätzt werden.

Kapitel 4

Entwurf

Dieses Kapitel beschäftigt sich mit dem Entwurf der Datensynchronisation zwischen *OpenERP* und *Jobmixer.com* sowie mit der Sicherung der Datenübertragung. Ziel ist es, für alle im Kapitel „Anforderungen“ formulierten Anforderungen eine optimale Umsetzung zu finden und diese zu erläutern.

4.1 Vorbetrachtungen

Bevor mit dem Entwurf der Synchronisation begonnen wird, sind noch einige Vorbetrachtungen nötig. Diese dienen dem Zweck, frühzeitig Probleme, welche während der Implementierungsphase auftreten können, zu erkennen und diese schon im Entwurf zu vermeiden.

Die erste Vorraussetzung, die es hierbei zu beachten gilt, ist, dass für *Jobmixer.com* eine SOAP-Schnittstelle zur Verfügung steht. Um diese zu nutzen wären Änderungen am Quellcode von *OpenERP* notwendig, welche wiederum vermieden werden sollen, da dadurch der Wartungsaufwand (z. B. Aktualisierungen) für *OpenERP* steigen würde. *OpenERP* selbst hingegen stellt bereits eine vollständig implementierte *XML-RPC*-Schnittstelle zur Verfügung. Da sich diese Schnittstelle mit PHP nutzen lässt, wird die Synchronisation auf der Seite von *Jobmixer.com* realisiert.

Die zweite Vorraussetzung ist der Unterschied in den Datenstrukturen von *OpenERP* und *Jobmixer.com*. Das bedeutet dass die Tabellenstruktur und die Relationen voneinander abweichen und dass Felder unterschiedliche Namen haben. Des Weiteren können die Datentypen der Felder in *OpenERP* und *Jobmixer.com* unterschiedlich sein.

Das macht es notwendig, für jedes Objekt und seine Felder den Bezeichner von *OpenERP* und *Jobmixer.com* einander zugeordnet abzuspeichern.

4.2 Module für *OpenERP*

Um die Anforderungen, die an *OpenERP* gestellt werden, zu erfüllen, ist es erforderlich, dass die entsprechenden Module bereitstehen. Für jede in Abschnitt 3.1.2 definierte Anforderung an *OpenERP* existiert bereits ein *OpenERP*-Modul. Tabelle 4.1 zeigt, welches Modul für welche Anforderung zur Verfügung steht.

Tabelle 4.1: Die Anforderungen an *OpenERP* und das entsprechende Modul

Anforderung	**Modulname**
Produkte anlegen und verwalten	`product`
Bestellungen annehmen und verwalten	`sale`
Rechnungen aus Bestellungen erzeugen	`account`
Kunden aufnehmen	`base`

4.3 Datenstruktur der relevanten Daten in *OpenERP*

Dieser Abschnitt soll zum Verständnis der Datenstruktur von *OpenERP* beitragen. Er soll aufgezeigen, welche *OpenERP*-Objekte für den Verkaufsprozess relevant sind und wie diese untereinander zusammenhängen. Dabei wird sich die Betrachtung auf die bei der Synchronisation von *Jobmixer.com* und *OpenERP* wichtigen Elemente beschränken.

Abbildung 4.1 zeigt die entsprechenden Datenbanktabellen der für die Synchronisation wichtigen Objekte. Sie stellt alle für die Synchronisation von *Jobmixer.com* und *OpenERP* wichtigen Relationen mit den dazugehörigen Fremdschlüsseln dar.

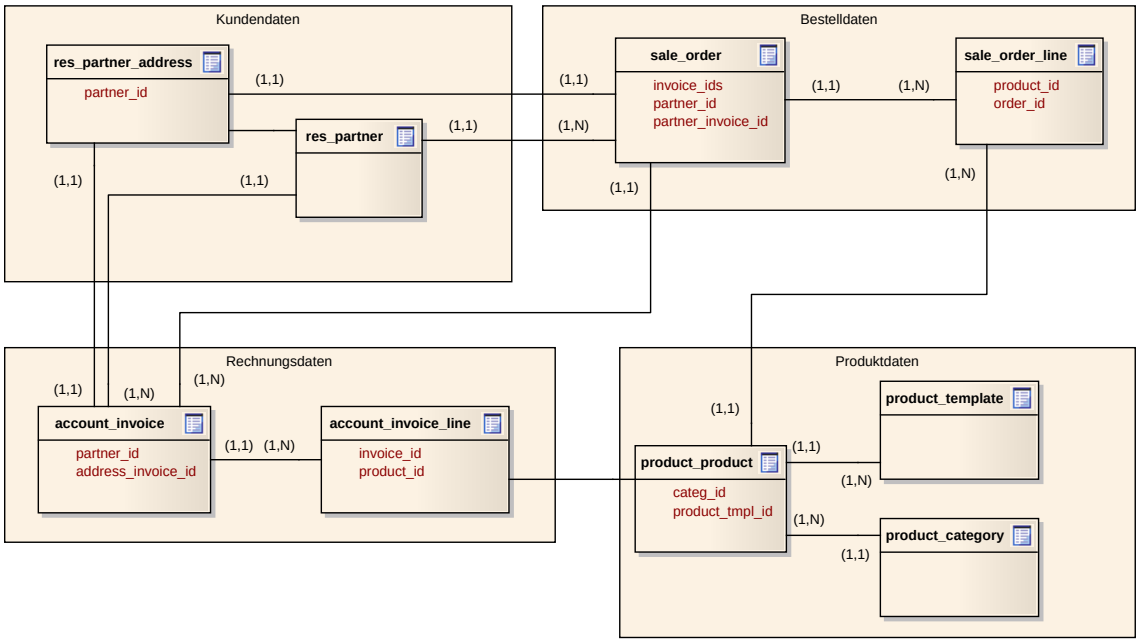

Abbildung 4.1: Datenstruktur der für die Synchronisation relevanten Tabellen in *OpenERP*

4.3.1 Kundendaten

Die Kundendaten sind in Grunddaten und Adressdaten des Kunden aufgeteilt. Die Grunddaten werden in `res_partner` gespeichert, während die Adressdaten in `res_partner_address` abgelegt werden. Durch diese Aufteilung ist es möglich, mehrere Adressen für einen Kunden anzulegen und diese für unterschiedliche Zwecke zu nutzen, z. B. als Rechnungsadresse und Lieferadresse.

4.3.2 Bestelldaten

Eine Bestellung ist in Bestelldaten und in Bestellposten aufgeteilt. In der Tabelle `sale_order` sind alle allgemeinen Daten für die Bestellung gespeichert, z. B. welchem Kunden sie zugeordnet ist und die dazugehörigen Rechnungen. Die Daten für die einzelnen Bestellposten werden in der Tabelle `sale_order_line` abgelegt. Eine Bestellung kann mehrere Bestellposten beihnhalten.

4.3.3 Rechnungsdaten

Die Struktur der Rechnungsdaten gliedert sich wie diejenige der Bestelldaten in Rechnung und Rechnungsposten. Es gibt die Rechnung und es gibt die Rechnungsposten. Alle allgemeinen, die Rechnung betreffenden Daten sind in `account_invoice` gespeichert, während die einzelnen Rechnungsposten in `account_invoice_line` hinterlegt sind.

4.3.4 Produktdaten

Ein Produkt wird in zwei Tabellen abgelegt, der Tabelle `product_product` und der Tabelle `product_template`. Beide Tabellen werden beim Auslesen der Daten aus *OpenERP* automatisch zusammengefasst. Das bedeutet, dass in diesem Fall nur *eine* Anfrage an *OpenERP* gestellt werden muss, um *alle* Produktdaten zu erhalten. Da die Produkte innerhalb von *OpenERP* angelegt werden, ist eine genauere Betrachtung der beiden Tabellen nicht notwendig.

Die Produktkategorien sind in der Tabelle `product_category` gespeichert und werden über eine Relation einem Produkt zugeordnet.

4.4 Datenstruktur der relevanten Daten in *Jobmixer.com*

Die Datenbankstruktur in Abbildung 4.2 zeigt die bereits bestehenden Tabellen von *Jobmixer.com*, welche für die Synchronisation von *Jobmixer.com* und *OpenERP* eine Rolle spielen. Die Struktur ist der in Abschnitt 4.3 beschriebenen Datenstruktur von *OpenERP* sehr ähnlich. Es existieren jedoch auch einige Abweichungen auf die in diesem Abschnitt genauer eingegangen wird.

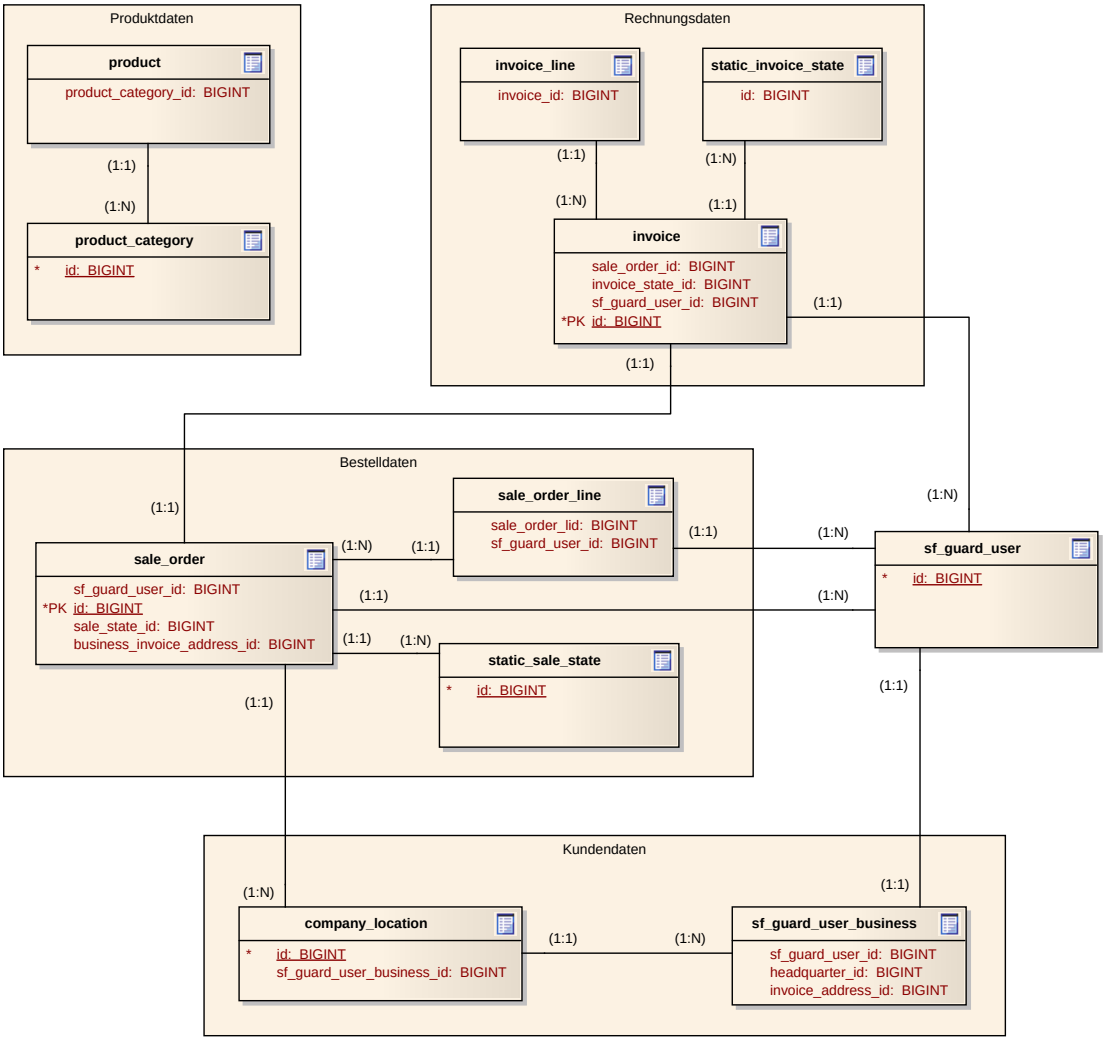

Abbildung 4.2: Datenstruktur der für die Synchronisation relevanten Tabellen in der *Jobmixer.com*-Datenbank

4.4.1 Kundendaten

Wie die Kundendaten in *OpenERP*, sind diese auch in der *Jobmixer.com*-Datenbank auf zwei Tabellen aufgeteilt. Die Tabelle `sf_guard_user_business` enthält die allgemeinen Kundendaten und die Tabelle `company_location` die Adressdaten der Kunden. Die Relation zu `sf_guard_user` wird verwendet, um den Kunden einem Benutzer zuzuordnen.

4.4.2 Bestelldaten

Wie in *OpenERP* werden die Bestelldaten auch in der *Jobmixer.com*-Datenbank in Bestellung und die Bestellposten getrennt gespeichert. Anders verhält es sich dagegen mit dem

Speichern des Status in der `static_sale_state`-Tabelle. Diese Tabelle beinhaltet alle Status, die von *OpenERP* für eine Bestellung definiert wurden. Dadurch ist es möglich, den Status einer Bestellung von *OpenERP* direkt auf *Jobmixer.com* zu übertragen. Wie auch bei den Kundendaten, wird die Relation zu `sf_guard_user` benötigt, um eine Bestellung einem *Jobmixer.com*-Benutzer zuzuordnen.

4.4.3 Rechnungsdaten

Die Struktur der Rechnungsdaten in der *Jobmixer.com*-Datenbank ist mit deren Struktur in der *OpenERP*-Datenbank identisch. Es exisitiert, wie bei den Bestelldaten, eine Tabelle für den Status einer Rechnung (`static_invoice_state`). Diese enthält ebenfalls alle Status einer Rechnung in *OpenERP*. Die Relation von `invoice` zu `sf_guard_user` dient auch hier der Zuordnung der Rechnung zu einem *Jobmixer.com*-Benutzer.

4.4.4 Produktdaten

Die Produktdaten in der *Jobmixer.com*-Datenbank bestehen nur aus einer Tabellen. Die beiden *OpenERP*-Tabellen `product_product` und `product_template` werden in der `product`-Tabelle zusammengefasst. Die Zuordnung der Produktkategorien ist dagegen identisch.

Ein weiterer Unterschied zwischen *OpenERP* und der *Jobmixer.com*-Datenbank im Bereich der Produktdaten besteht darin, dass keine Relationen zu anderen Tabellen der *Jobmixer.com*-Datenbank vorhanden sind. Diese sind nicht notwendig, da die Produktdaten von *Jobmixer.com* in eine Bestellung oder einer Rechnung kopiert werden. Dadurch wird verhindert, das Bestellungen und Rechnungen, bei einer Änderungen an einem Produkt nachträglich geändert werden.

4.5 Schnittstellen und Techniken der Synchronisation

In diesem Abschnitt wird ein Überblick gegeben, welche Schnittstellen und Techniken für die Synchronisation eingesetzt werden und wie deren Aufgabenbereich aussieht.

4.5.1 *symfony*

Das *symfony*-Framework bildet die Basis von *Jobmixer.com*. Um die Synchronisation auf der Seite von *Jobmixer.com* zu implementieren, werden von *symfony* bereitgestellte Funktionen genutzt. So wird die Plugin-Schnittstelle für die Implementierung der Synchronisation in *Jobmixer.com* genutzt. Dadurch wird es möglich, das Anmelden von Daten für die Synchronisation in den Programmablauf von *Jobmixer.com* zu integrieren.

Desweiteren bietet *symfony* die Möglichkeit, eigene Tasks zu schreiben, mit deren Hilfe die Synchronisation durch einen einzigen Kommandozeilenbefehl ausgelöst werden kann.

4.5.2 *XML-RPC*

Die von *OpenERP* bereitgestellte *XML-RPC*-Schnittstelle wird für die Synchronisation genutzt. Dadurch können die von *OpenERP* bereitgestellten Methoden zum Erzeugen, Aktualisieren, Lesen und Löschen der *OpenERP*-Objekte genutzt werden. Das hat den Vorteil, dass die Objekte in *OpenERP* auf die von *OpenERP* vorgegebene Art und Weise behandelt werden. Dies betrifft z. B. das Erzeugen eines neuen Objektes in *OpenERP*. Dabei wird der Inhalt einiger Felder über eine im *OpenERP* definierte Methode automatisch erstellt, oder es werden Objekte beim Auslesen der Daten zu einer Ausgabe zusammen gefasst, z. B. die Tabellen `product_product` und `product_template` zu einer Tabelle.

4.5.3 *Doctrine*

Doctrine ist der ORM von *Jobmixer.com*. Er verwaltet das Speichern und Lesen von Informationen. Die Synchronisation wird *Doctrine* verwenden, um Daten für die Synchronisation aus der *Jobmixer.com*-Datenbank zu lesen und empfangene Daten zu speichern. Bei der Kontrolle des Abgleiches finden auch Zugriffe auf die *Jobmixer.com*-Datenbank statt. Diese werden ebenfalls von *Doctrine* durchgeführt.

4.6 Datenbankstruktur für die Synchronisation

Das in Abbildung 4.3 dargestellte ERM[38] zeigt die Struktur der Datenbanktabellen des Plugins. Die dargestellten Tabellen sind in unterschiedliche Aufgabenbereiche aufgeteilt.

[38] Entity-Relationship-Model (ERM)

Die Bereiche, welche zum Einsatz kommen sind der Controller für die Steuerung, das Datenmapping zum Zuordnen der Daten und *Jobmixer.com* um die Daten für die Synchronisation auszulesen bzw. zu speichern. Dies dient einem besseren Überblick über die Struktur.

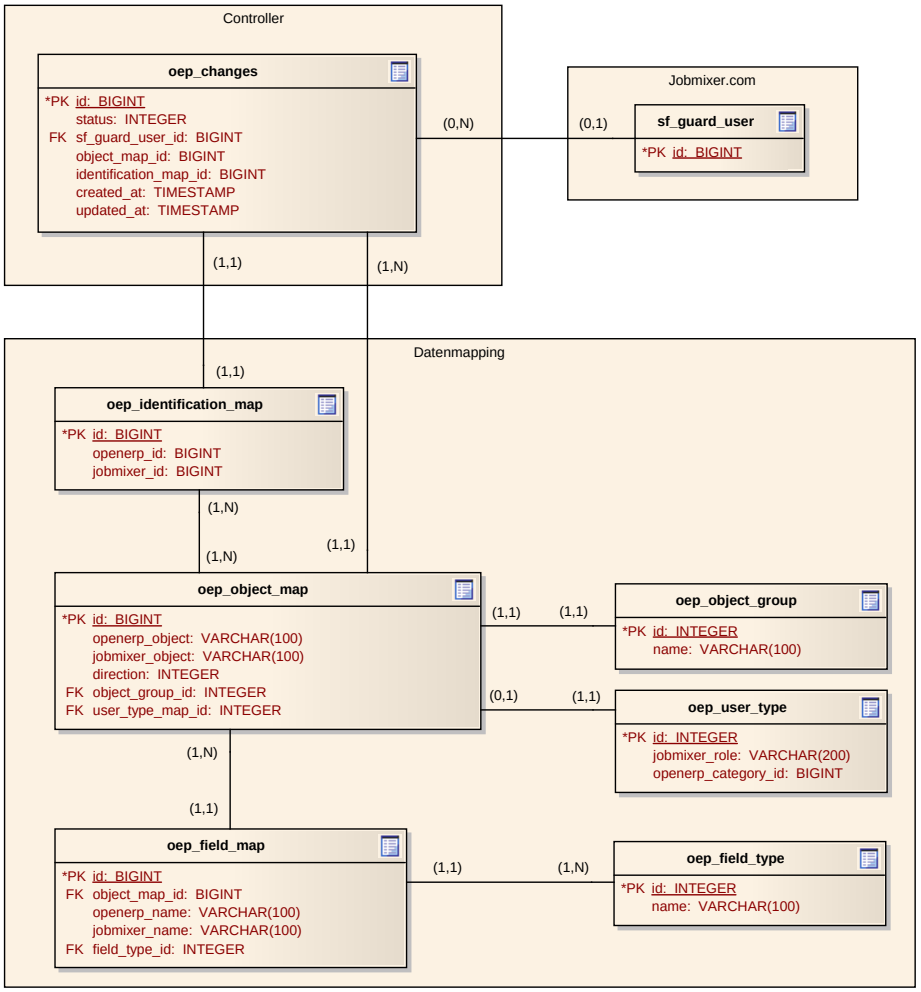

Abbildung 4.3: Datenstruktur von `iaOpenErpPlugin`

4.6.1 Steuerungstabelle der Synchronisation

Die im Abschnitt `Controller` existierende Tabelle ist dafür zuständig, die Daten für die Kontrolle der Synchronisation zu speichern und wieder zur Verfügung zu stellen. In ihr werden alle Objekte registriert, die synchronisiert werden sollen.

Die `status`-Spalte gibt an, wie das Objekt synchronisiert werden soll. Ist es z. B. noch nicht mit *OpenERP* synchronisiert, bekommt es den Status `neu`. Diese Zuordnung bewirkt, dass das Objekt bei der Synchronisation in *OpenERP* angelegt wird.

Wenn das Objekt bereits existiert, wird der Status `aktualisiert` vergeben und die Daten werden bei der Synchronisation aktualisiert. Einen kompletten Überblick über die möglichen Synchronisationsstatus gibt Tabelle 4.2.

Tabelle 4.2: Mögliche Status der Synchronisation

Status	Bezeichnung	Beschreibung
neu	`SYNC_STATE_NEW`	Das Objekt wurde neu hinzugefügt und noch nicht synchronisiert.
aktualisiert	`SYNC_STATE_UPDATE`	Das Objekt wurde bereits synchronisiert und wird jetzt aktualisiert.
synchronisiert	`SYNC_STATE_DONE`	Das Objekt wurde synchronisiert.
Richtung *OpenERP*	`SYNC_STATE_ONLY_UP`	Das Objekt wird nur von *Jobmixer.com* nach *OpenERP* synchronisiert.
Richtung *Jobmixer.com*	`SYNC_STATE_ONLY_DOWN`	Das Objekt wird nur von *OpenERP* nach *Jobmixer.com* synchronisiert.

4.6.2 Mapping der Daten

Die Tabellen aus dem Abschnitt `Datenmapping` dienen dazu, die Objekte und Felder von *Jobmixer.com* und *OpenERP* richtig einander zuzuordnen.

Zuordnung der Objekte

Für die Zuordnung von *Jobmixer.com*-Objekten zu *OpenERP*-Objekten ist die `oep_object_map`-Tabelle verantwortlich. Anhand dieser wird für jedes gegebene Objekt das passende Gegenstück gefunden und aufgerufen. Um die Richtung für die Synchronisation eines Objektes zu bestimmen, wird diese ebenfalls in der `oep_object_map`-Tabelle angegeben.

Zuordnung der Datenbankfelder

Die Zuordnung der Datenbankfelder von *Jobmixer.com* auf die Datenbankfelder von *OpenERP* ist in `oep_field_map` gespeichert. Zusammen mit den in `oep_field_type` gespeicherten Datentypen, die für die Synchronisation benötigt werden, wird der Inhalt der Datenbankfelder ausgelesen und für die Synchronisation nach *OpenERP* vorbereitet. Welchen Datentyp die Datenbankfelder von *Jobmixer.com* besitzen, ist von *Doctrine* erfasst.

Synchronisationsgruppen

Die Gruppen, in welche die Objekte für die Synchronisation eingeteilt werden, sind in `oep_object_group` gespeichert. Mithilfe der Gruppen werden die Objekte für die Synchronisation zusammengefasst. Das betrifft z. B. Produkte die zusammen mit den Produktkategorien synchronisiert werden.

Mapping von Benutzergruppen

Das Mapping der Benutzergruppen von *Jobmixer.com* und *OpenERP* geschieht in der Tabelle `oep_user_type`. Damit ist es möglich, für jede Benutzergruppe unterschiedliche Objekte anzugeben aus denen die Daten gelesen bzw. gespeichert werden. Dies ist z. B. bei den Grund- und Adressdaten der Benutzer notwendig.

Zuordnung der Daten

Die genaue Zuordnung der Daten wird mit Hilfe der `oep_identification_map`-Tabelle vorgenommen. Dazu werden die `Id` des *Jobmixer.com*-Objektes und die `Id` des *OpenERP*-Objektes einander zugeordnet und in der `oep_identification_map`-Tabelle gespeichert. Die Tabelle wird bei jeder Synchronisation aktualisiert und dabei ständig erweitert. In Verbindung mit der `oep_object_map` und der `oep_changes`-Tabelle, werden die Daten eines Objektes einem Kunden zugordnet.

4.7 Entwurf der Struktur

In diesem Abschnitt werden die verwendeten Klassen und ihre jeweilige Struktur (Abbildung 4.4) aufgezeigt. Die Klassen sind in Pakete aufgeteilt, welche verschiedene Aufgaben besitzen.

Das `Jobmixer.com`-Paket steht in der Abbildung für *Jobmixer.com* und wird als eigenständiges System betrachtet. Auf den Aufbau des *Jobmixer.com*-Pakets wird nicht eingegangen, da es zu umfangreich und der Entwurf nicht Teil dieser Diplomarbeit ist.

Das eigenständige `XML-RPC`-Paket stellt eine für die Nutzung der *OpenERP*-Schnittstelle mit PHP importierte Bibliothek dar und wird als Teilsystem betrachtet.

Auch der Aufbau der Bibliothek soll an dieser Stelle nicht weiter betrachtet werden, da dieser keine große Rolle für die Nutzung der Bibliothek darstellt.

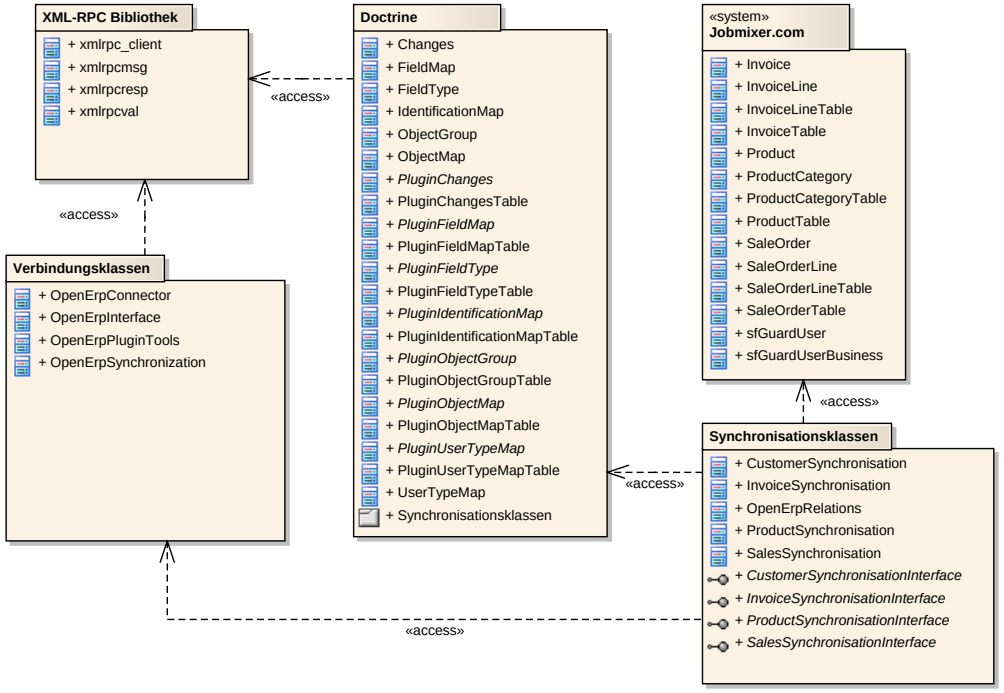

Abbildung 4.4: Übersicht über die Pakete, welche für die Synchronisation benötigt werden.

4.7.1 Die Datenbankklassen für die Synchronisation

Die Datenbankklassen für die Synchronisationsklassen enthalten die Datenbanklogik und Teile der Geschäftslogik. Alle Datenbankklassen (Abbildung 4.5) wurden anhand der Definition des Datenmodells in der Datei `schema.yml` des Plugins durch *Doctrine* erzeugt. Auf die Funktionalität der einzelnen Datenbanktabellen wurde bereits im Abschnitt 4.6 eingegangen.

Die Klassen mit dem Suffix `Table` enthalten die Datenbanklogik. Es werden ebenfalls die im Abschnitt „Datenbankstruktur für die Synchronisation“ beschriebenen Datenbanktabellen abgebildet und als Objekte zur Verfügung gestellt. Mithilfe dieser Objekte werden bestimmte Daten einer Tabelle oder die komplette Tabelle ausgelesen, ohne dass eine SQL[39]-Abfrage geschrieben werden muss.

[39] Structured Query Language (SQL)

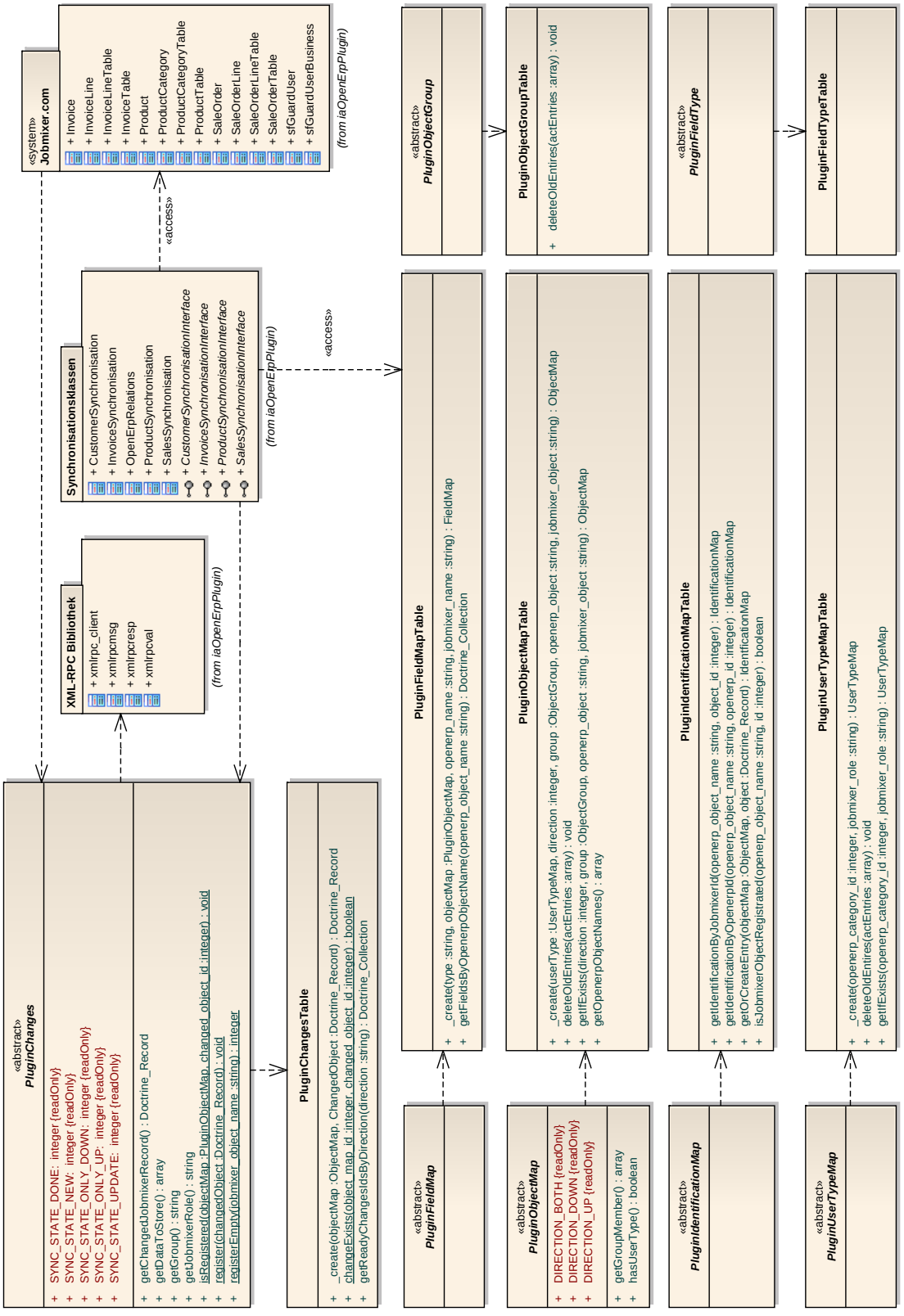

Abbildung 4.5: Übersicht über alle von *Doctrine* erzeugten Klassen dieses Plugins

In den als `abstract` gekennzeichneten Klassen in Abbildung 4.5 werde die Datenbanktabellen für die Steuerung der Synchronisation und für das Datenmapping als Objekte abgebildet. Das bedeutet, dass jeder Datensatz einer Tabelle als Objekt verfügbar ist und Änderungen an diesem Datensatz über das Objekt vorgenommen werden. Außerdem enthalten diese Klassen den Teil der Synchronisation, der für die Datenverarbeitung zuständig ist. Dies sind z. B. Methoden, um einen Datensatz speziell zu formatieren (`getDataToStore()` aus der `PluginChanges`-Klasse).

Abbildung 4.6 zeigt am Beispiel von `PluginObjectMap`, wie die abstrakten Klassen des Plugins erweitert werden. Dieser Vorgang wird über Klassen realisiert, die von *Doctrine* beim Erstellen der abstrakten Klassen automatisch mit erzeugt werden. Die automatisch erzeugten Klassen befinden sich im *symfony*-Verzeichnis unter `lib/model/doctrine/iaOpenErpPlugin` und damit außerhalb des Plugin-Verzeichnisses. Dadurch das die automatisch erzeugten Klassen von den Plugin Klassen erben und bei einer Aktualisierung nicht überschrieben werden, kann das Plugin erweitert oder angepasst werden, ohne den Plugin-Quellcode zu verändern.

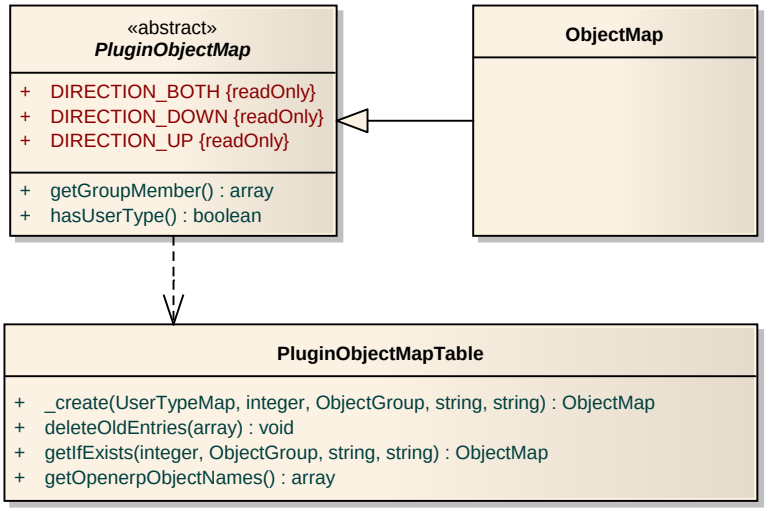

Abbildung 4.6: Beispiel für die Zusammenhänge zwischen den von *Doctrine* erstellten Klassen

4.7.2 Die Klassen zur Verwaltung der Synchronisation

Die Klassen zur Verwaltung der Synchronisation (Abbildung 4.7) ermöglichen die Synchronisation zwischen *OpenERP* und *Jobmixer.com*, da sie alle erforderlichen Funktionen zur Nutzung der *OpenERP*-Schnittstelle bereitstellen. Die folgenden Abschnitte gehen genauer auf den Verwendungszweck der einzelnen Klassen ein.

OpenErpConnector

Die `OpenErpConnector`-Klasse ist für den Verbindungsaufbau zwischen *Jobmixer.com* und *OpenERP* und dessen Autorisierung verantwortlich. Sie ist nach dem Singleton-Entwurfsmuster konzipiert. Damit nur eine Verbindung zu *OpenERP* gleichzeitig existiert. Um zu verhindern dass eine zweite Instanz erzeugt wird, wird der Konstruktor als `protected` deklariert.

OpenErpInterface

`OpenErpInterface` stellt vordefinierte Methoden zum Erzeugen, Aktualisieren, Lesen und Suchen von *OpenERP*-Objekten bereit. Um weitere Funktionen zu nutzen, kann die `execute()`-Methode verwendet werden, da sie in der Lage ist, alle Methoden eines *OpenERP*-Objektes über die *XML-RPC*-Schnittstelle zu verwenden.

Die `OpenErpInterface`-Klasse ist ebenfalls nach dem Singleton-Entwurfsmuster konzipiert, da auch hier maximal eine Instanz und damit auch nur ein Aufruf einer *OpenERP*-Methode über XML-RPC stattfinden soll. Der Konstruktor ist hier ebenfalls `protected`.

OpenErpSynchronization

`OpenErpSynchronization` stellt die Methode zum Starten der gesamten Synchronisation und die Methode für das Synchronisieren eines einzelnen Eintrages von `Changes` zur Verfügung.

Bei dieser Klasse wird ebenfalls auf das Singleton-Entwurfsmuster zurückgegriffen. So kann immer nur ein Synchronisationsvorgang gleichzeitig ablaufen.

OpenErpPluginTools

`OpenErpPluginTools` stellt Hilfsfunktionen für die verschiedenen Teile des Plugins bereit. Alle Methoden dieser Klasse sind statisch. Sie lassen sich aufrufen, ohne vorher eine Instanz der Klasse zu erzeugen. Die Klasse dient als Sammelklasse für die verschiedenen Hilfsfunktionen.

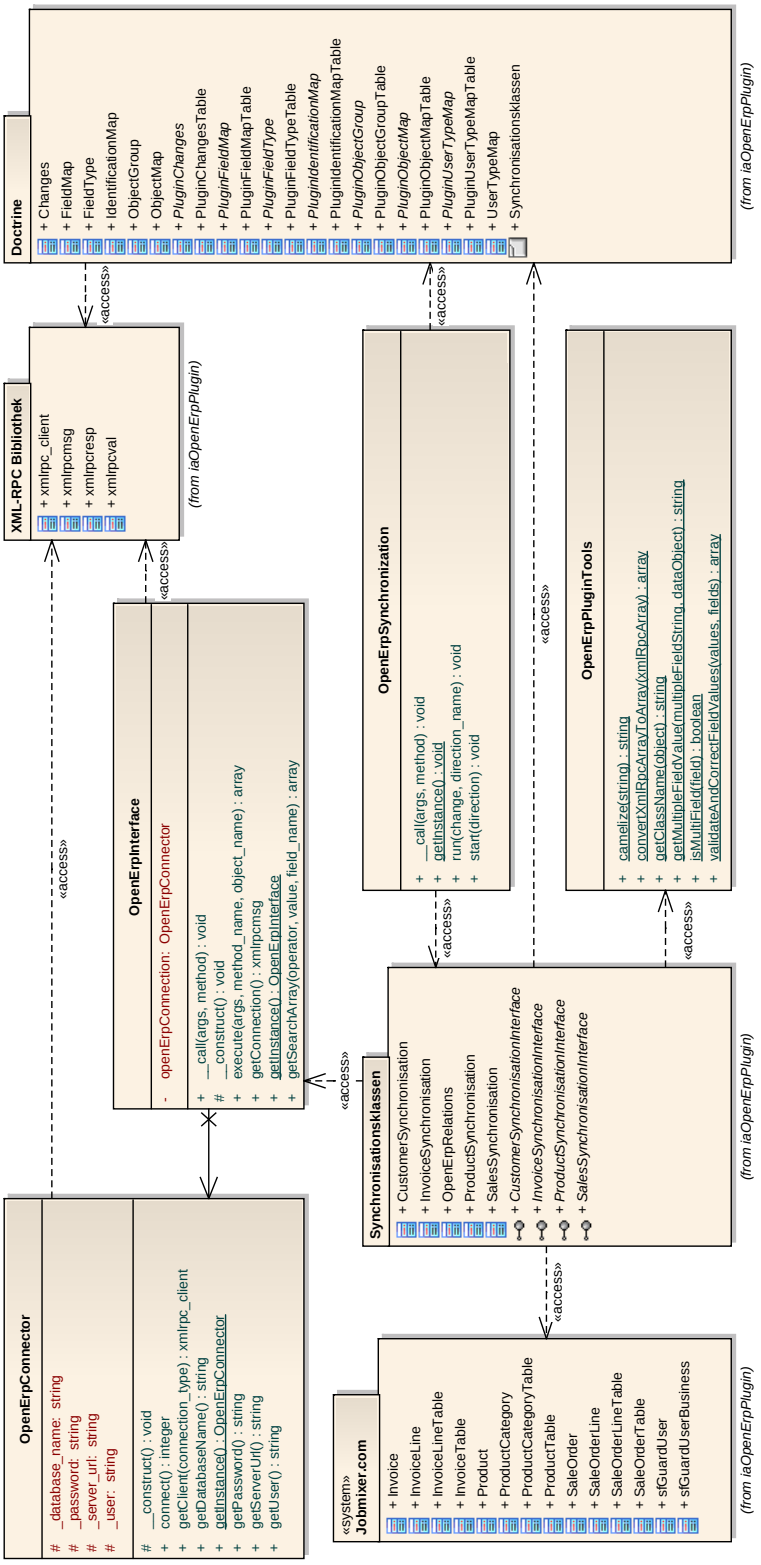

Abbildung 4.7: Übersicht über alle Klassen, welche für die Verwaltung der Synchronisation zwischen *Jobmixer.com* und *OpenERP* zuständig sind

4.7.3 Die Klassen für die Steuerung der Synchronisation

Die Klassen und Interfaces zur Steuerung der Synchronisation (Abbildung 4.8) werden durch einen Task aus den Konfigurationseinstellungen in der Datei `openerp.yml` erzeugt (vgl. Abschnitt 4.8). Die einzige Ausnahme ist die Klasse `OpenErpRelations`. Diese Klasse wird vom Plugin bereitgestellt. Ihre Aufgabe ist es, für die Synchronisationsklassen Methoden bereitzustellen, die den einzutragenden Wert für *OpenERP*-Relationen zurückgeben. Das hat den Grund, dass Relationen bei der Synchronisation durch die Angabe der `Id` des verbundenen *OpenERP*-Objektes angelegt werden.

Die Aufgabe der Synchronisationsklassen (z. B.`CustomerSynchronisation`) besteht darin, die Synchronisation für jedes *OpenERP*-Objekt speziell angepasst durchzuführen. Das bedeutet, dass auf die Besonderheiten jedes *OpenERP*-Objektes eingegangen werden muss, z. B. auf Relationen zu anderen Objekten. Die Interfaces haben dabei die Aufgabe vorzugeben, welche Methoden diese Klassen dazu implementieren müssen.

Die Aufteilung der Interfaces und Synchronisationsklassen erfolgt nach den in der Konfigurationsdatei angegebenen Gruppen (vgl. Abschnitt 4.8). Dabei stellt der Gruppenname immer den ersten Teil des Namens eines Interfaces oder einer Synchronisationsklasse dar. In Abbildung 4.8 sind bereits die Klassen, die bei der Umsetzung dieser Diplomarbeit erzeugt werden sollen, enthalten.

Die Interfaces werden bei jedem Durchlauf des Tasks neu erstellt. Dadurch enthalten sie immer die aktuellen Vorgaben bezüglich der Methoden, welche für die Synchronisation der einzelnen Objekte implementiert werden müssen. Abgeleitet werden diese Vorgaben aus den Angaben der Datei `openerp.yml`. Die Synchronisationsklassen werden dagegen nur erstellt, sofern sie noch nicht existieren. Wenn sie bereits existieren, werden die Klassen und damit die Implementierung der Methoden für die Synchronisierung nicht überschrieben. Das bedeutet: wird ein neues Objekt für die Synchronisation in der Datei `openerp.yml` eingetragen, dann wird speziell für dieses Objekt eine Klasse zur Steuerung der Synchronisation sowie ein Interface mit den Vorgaben, zu den darin enthaltenen Methoden erstellt.

Die Synchronisationsklassen und die Interfaces werden nicht innerhalb des Pluginverzeichnisses erzeugt, sondern in `lib/model/doctrine/ /synchronisation` (Klassen zur Verwaltung der Synchronisation) und `lib/model/doctrine/iaOpenErpPlugin/generated/interface` (Interfaces).

Dadurch können die Klassen durch andere Entwickler erweitert werden, ohne das Änderungen am Plugin vorgenommen werden müssen.

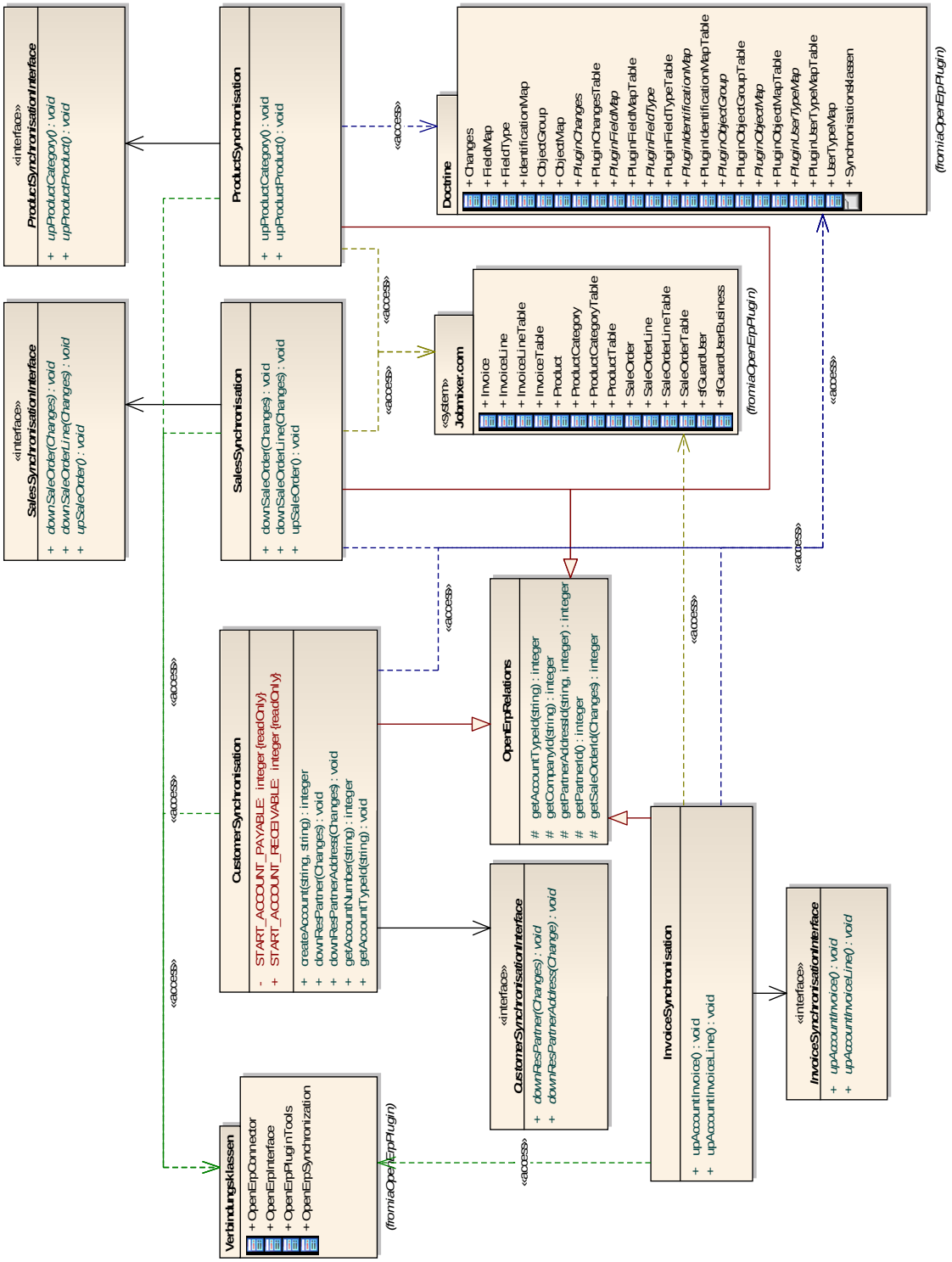

Abbildung 4.8: Die Klassen für die Steuerung der Synchronisation

4.8 Konfiguration der Synchronisation

Damit es möglich ist Objekte zur Synchronisation hinzuzufügen oder bestehende Objekte anzupassen, kann die Synchronisation konfiguriert werden. Zu diesem Zweck existiert eine zentrale Konfigurationsdatei, mit dem Namen `openerp.yml`. In dieser Datei werden folgende Einstellungen vorgenommen:

- Das Mapping der Objekte,
- das Mapping der Felder,
- die Angaben über Benutzertypen, Objektgruppen und Datentypen für die Synchronisation,
- die Einordnung der Objekte in Synchronisationsgruppen,
- eine Angabe, welchen Datentyp jedes einzelne Feld in *OpenERP* besitzt sowie
- die Angabe für die Synchronisationsrichtung der einzelnen Objekte.

Eine Ausnahme bilden die Verbindungsdaten für *OpenERP*. Diese werden in der Datei `app.yml` abgelegt, da es von *symfony* vordefinierte Zugriffsmethoden für diese Datei gibt und dadurch auf die Daten leichter zugegriffen werden kann.

Die Einstellungen der Konfigurationsdatei gelangen über einen *symfony*-Task in die entsprechenden Datenbanktabellen. Wenn der Task die Konfiguration abgeschlossen hat, kann das Eintragen von Änderungen und das Synchronisieren dieser Änderungen beginnen.

4.9 Steuerung der Synchronisation

Dieser Abschnitt zeigt, wie die Kontrolle der Synchronisation funktionieren wird, und gibt einen Überblick über die Aufgaben, die der Controller beherrscht.

Abbildung 4.9 zeigt eine komplette Übersicht aller an der Synchronisation beteiligten Komponenten. Wie zu erkennen ist, werden alle Aktionen der Synchronisation durch eine zentrale Stelle („Kontrolle der Synchronisation“) verwaltet.

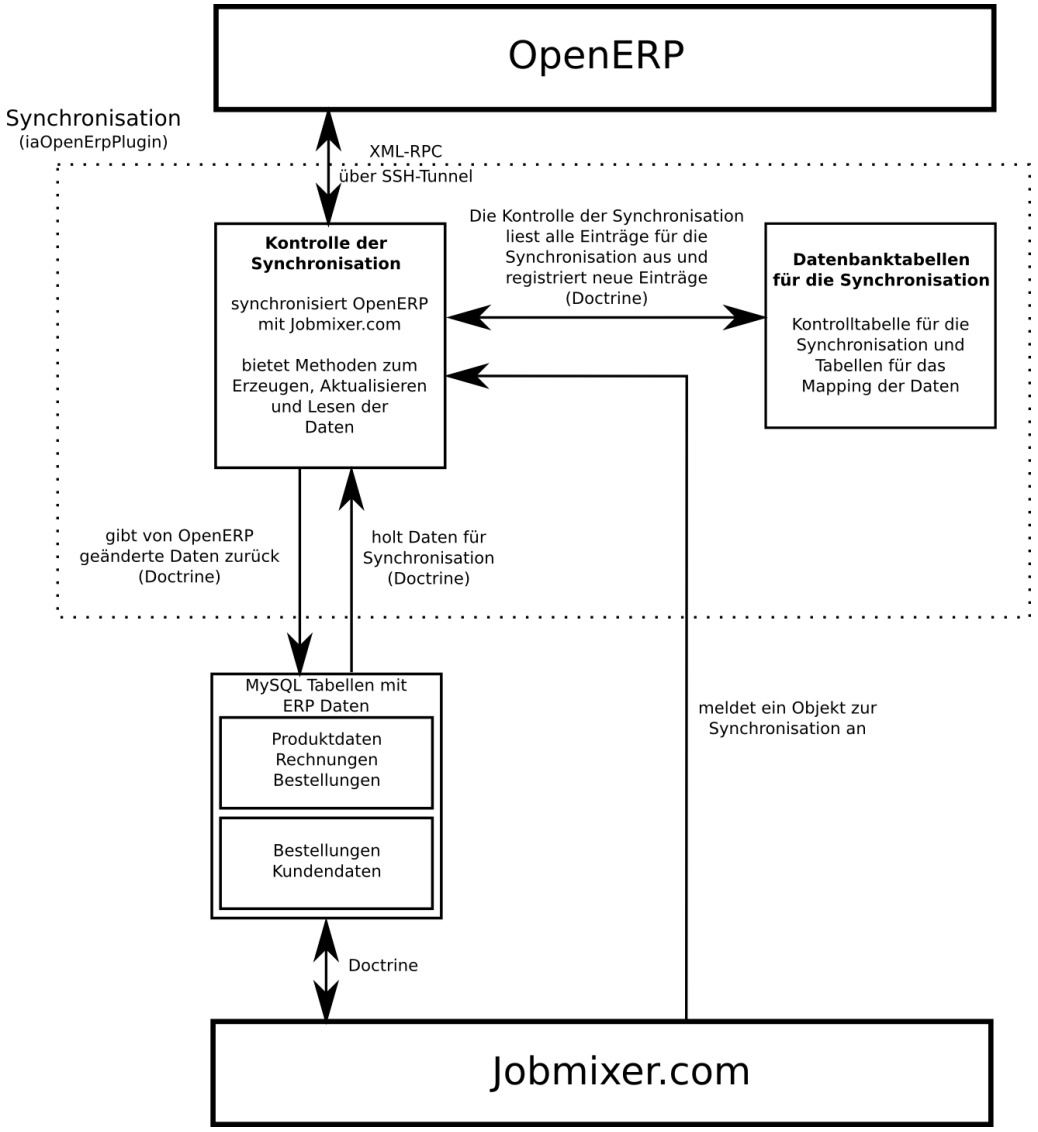

Abbildung 4.9: Übersicht über die Struktur zur Steuerung der Synchronisation

Werden Änderungen an für *OpenERP* relevanten *Jobmixer.com*-Objekten vorgenommen, wird dies dem Controller mitgeteilt. Dieser trägt die Änderung in die dafür zuständige Datenbanktabelle ein. Bekommt der Controller das Signal zum Starten der Synchronisation von *Jobmixer.com* nach *OpenERP*, liest er alle eingetragenen Änderungen aus und besorgt sich die Daten aus den entsprechenden *Jobmixer.com*-Objekten. Die ausgelesenen Daten werden über die *XML-RPC*-Schnittstelle von *OpenERP* übertragen und gespeichert.

Erhält der Controller das Signal zur Synchronisation der Daten von *OpenERP* nach *Jobmixer.com*, trägt er – abhängig vom Objekt – vor Beginn eigene Änderungen in der `oep_changes`-Tabelle ein. Danach wird diese ausgelesen, die Daten werden übertragen und in der Datenbank von *Jobmixer.com* gespeichert.

Der Controller beherrscht folgende Aufgaben:

- neue oder geänderte Objekte für die Synchronisation registrieren,
- feststellen, ob ein zu registrierendes Objekt neu anzulegen oder zu aktualisieren ist,
- Daten für die Synchronisation bereitstellen,
- passende Methode zur Synchronisation für jedes Objekt bereitstellen,
- Verwalten des Zugriffes auf die *XML-RPC*-Schnittstelle.

Zur Verwaltung der *XML-RPC*-Schnittstelle beherrscht der Controller:

- den Verbindungaufbau zu *OpenERP*,
- die Authentifizierung bei *OpenERP*,
- das Bereitstellen der Methoden zum Suchen, Lesen, Erzeugen und Aktualisieren von Objekten.

4.10 Ablauf der Synchronisation

In diesem Abschnitt wird ein Überblick über die Synchronisationsvorgänge gegeben, die zur Synchronisation zwischen *Jobmixer.com* und *OpenERP* notwendig sind. Zu diesem Zweck wird jeder dieser Vorgänge einzeln betrachtet und die Schrittfolge erläutert.

Bei jeder Synchronisation wird davon ausgegangen, dass alle Daten korrekt eingeben werden. Das Validieren der Daten und die Fehlerbehandlung bei einer inkorrekten Eingabe fallen in den Aufgabenbereich von *Jobmixer.com* und werden aus diesem Grund an dieser Stelle nicht betrachtet.

4.10.1 Synchronisation von Kundendaten

Synchronisation für einen neuen Benutzer

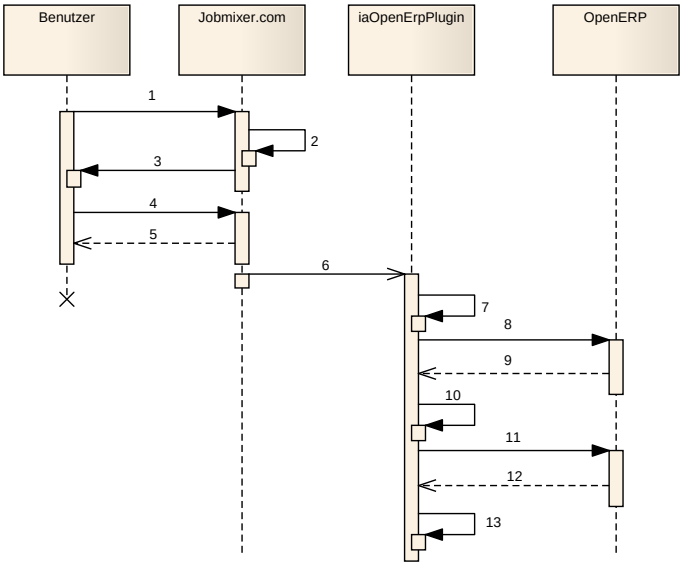

Abbildung 4.10: Ablauf zum Anlegen eines neuen Benutzers

1. Der Benutzer füllt das Registrationsformular aus und sendet die Daten an *Jobmixer.com*.
2. Die eingegebenen Daten werden überprüft und gespeichert; es wird ein neues Benutzerkonto angelegt.
3. Der Benutzer bekommt seine Registrierung per E-Mail bestätigt und wird aufgefordert, das Benutzerkonto zu aktivieren.
4. Das Benutzerkonto wird aktiviert.
5. Der Benutzer erhält die Bestätigung, dass die Registrierung abgeschlossen ist.
6. Der neue Benutzer wird zur Synchronisation angemeldet.
7. Es wird ein neuer Eintrag für die Synchronisation erstellt.
8. Der neue Kunde wird in *OpenERP* erzeugt.
9. *OpenERP* gibt die `Id` des neu angelegten Benutzers zurück.
10. Die zurückgegebene *OpenERP*-Kunden-`Id` und die *Jobmixer.com*-`Id` des Kunden werden einander zugeordnet.

11. Die Adresse des Kunden wird in *OpenERP* angelegt und dem Kunden zugeordnet.

12. *OpenERP* gibt die `Id` der neu angelegten Adressdaten zurück.

13. Die zurückgegebene Adressdaten-`Id` wird der entsprechenden *Jobmixer.com*-`Id` zugeordnet.

Synchronisation für einen vorhandenen Benutzer

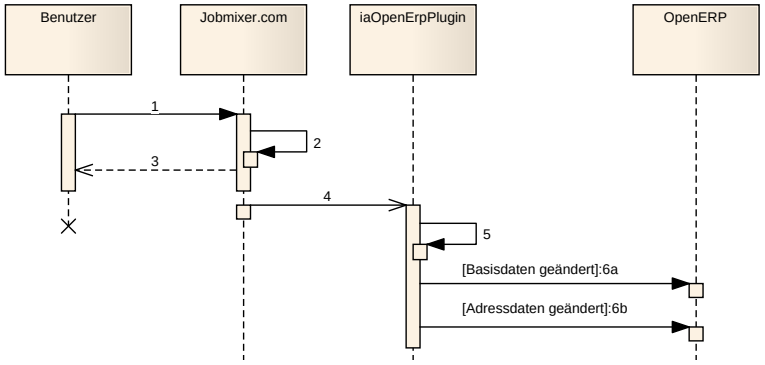

Abbildung 4.11: Aktualisieren der Daten eines bereits vorhanden Benutzers

1. Benutzer ändert seine Grunddaten, z. B. seine Adressdaten, und sendet diese an *Jobmixer.com*.

2. Änderungen werden überprüft und gespeichert.

3. Der Benutzer bekommt bei erfolgreicher Änderung die Rückmeldung, dass seine Daten geändert wurden.

4. Die Änderungen werden zur Synchronisation angemeldet.

5. Das Plugin bemerkt, dass die Daten schon einmal synchronisiert wurden und erzeugt lediglich einen Eintrag zum Aktualisieren der Daten.

6. (a) Grunddaten des Kunden werden aktualisiert.

   (b) Adressdaten des Kunden werden aktualisiert.

4.10.2 Synchronisation von Produktdaten

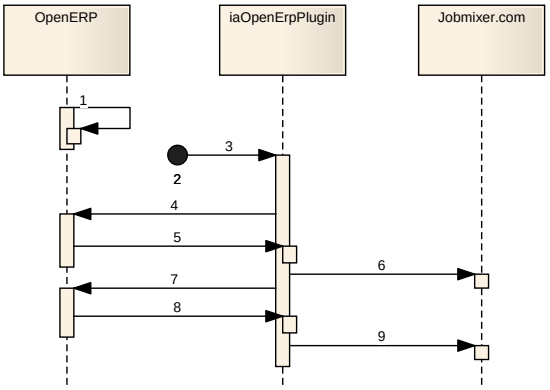

Abbildung 4.12: Ablauf, wenn Produkte hinzugefügt oder aktualisiert werden

1. Produkte werden in *OpenERP* angelegt und aktualisiert.
2. Produktsynchronisation wird z. B. durch ein zeitgesteuertes Skript ausgelöst.
3. Produkte und Produktkategorien werden zur Synchronisation angemeldet.
4. Produktkategorien werden aus *OpenERP* herausgesucht.
5. Produktkategorien werden von *OpenERP* zum `iaOpenErpPlugin` übertragen.
6. `iaOpenErpPlugin` überträgt die Produktkategorien zu *Jobmixer.com*.
7. Produkte werden von *OpenERP* zum `iaOpenErpPlugin` übertragen.
8. Das `iaOpenErpPlugin` überträgt die Produktkategorien zu *Jobmixer.com*.
9. Produkte werden zu *Jobmixer.com* übertragen.

4.10.3 Synchronisation von Bestelldaten

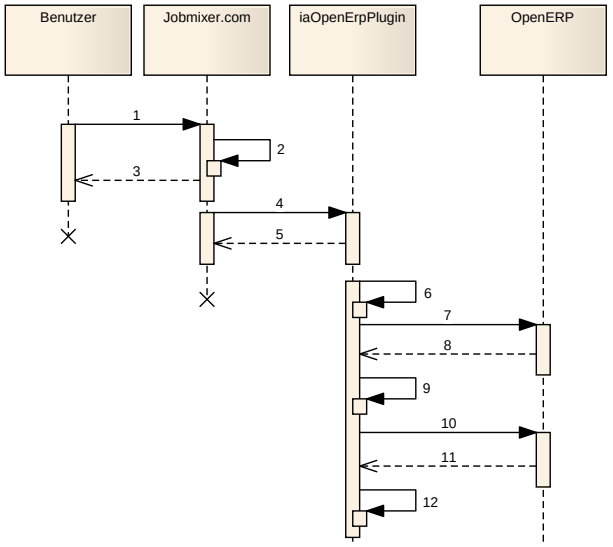

Abbildung 4.13: Ablauf der Synchronisation von Bestellungen

1. Der Kunde führt die Bestellung durch.
2. Die Bestellung wird gespeichert.
3. Der Kunde bekommt die Bestellbestätigung.
4. Die Bestellung wird zur Synchronisation angemeldet.
5. Der Status der Bestellung wird auf `In Bearbeitung` gesetzt.
6. Neue Einträge zur Synchronisation werden erstellt.
7. Die Bestellung (`SaleOrder`) wird synchronisiert.
8. *OpenERP* gibt die `Id` der neu angelegten Bestellung zurück.
9. Die zurückgegebene *OpenERP*-`Id` und die *Jobmixer.com*-`Id` der Bestellung werden einander zugeordnet.
10. Alle Bestellposten (`SaleOrderLine`) werden synchronisiert.
11. *OpenERP* gibt die `Id` des neu angelegten Bestellpostens zurück.
12. Die zurückgegebene *OpenERP*-`Id` des Bestellpostens wird der entsprechenden *Jobmixer.com*-`Id` zugeordnet.

4.10.4 Synchronisation zur Aktualisierung des Bestellstatus

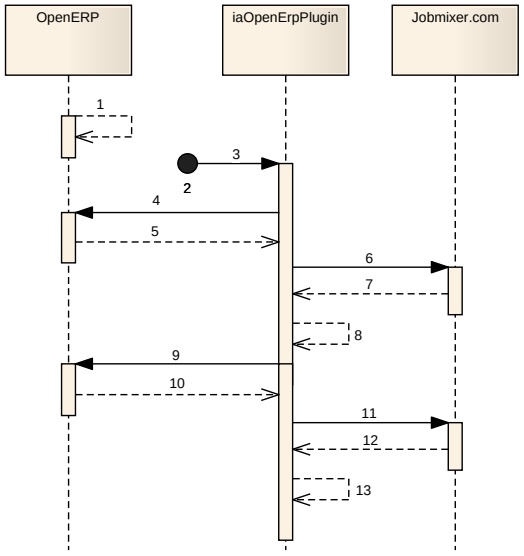

Abbildung 4.14: Ablauf für die Aktualisierung des Bestellstatus

1. Bestellung wird in *OpenERP* geändert.
2. Synchronisation der Bestellung wird in Richtung *Jobmixer.com* ausgelöst, z. B. durch einen Task.
3. Es werden alle Bestellungen mit dem Status `progress` herausgesucht.
4. Gefundene Bestellungen werden von *OpenERP* zu `iaOpenErpPlugin` übertragen.
5. Bestellungen, die zu *Jobmixer.com* gehören, werden aktualisiert.

4.10.5 Synchronisation von Rechnungsdaten

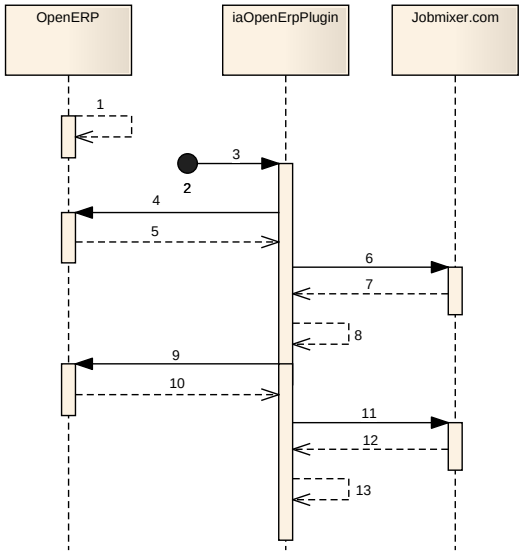

Abbildung 4.15: Ablauf der Synchronisation von Rechnungen

1. Rechnung wird in *OpenERP* erstellt.
2. Synchronisation der Rechnungen wird z. B. durch das *Jobmixer.com* Backend ausgelöst.
3. Rechnung und Rechnungsposten werden für die Synchronisation registriert.
4. Rechnung wird aus *OpenERP* herausgesucht.
5. Die Rechnungsdaten werden von *OpenERP* zum `iaOpenErpPlugin` übertragen.
6. Rechnungsdaten werden vom `iaOpenErpPlugin` zu *Jobmixer.com* übertragen und der dazugehörigen Bestellung zugeordnet.
7. *Jobmixer.com* gibt die `Id` der neu angelegten Rechnung zurück.
8. Synchronisationseintrag für Rechnung mit *OpenERP*-`Id` und *Jobmixer.com*-`Id` wird erstellt.
9. Rechnungsposten werden aus *OpenERP* herausgesucht.
10. Rechnungsposten werden von *OpenERP* zum `iaOpenErpPlugin` übertragen.
11. Rechnungsposten werden vom `iaOpenErpPlugin` zu *Jobmixer.com* übertragen und der Rechnung zugeordnet.

12. *Jobmixer.com* gibt die `Id`s der neu angelegten Rechnungsposten zurück.

13. Synchronisationseintrag für die Rechnungsposten mit *OpenERP*-`Id` und *Jobmixer.com*-`Id` wird erstellt.

4.11 Absicherung der Kommunikation

Die Kommunikation zwischen *Jobmixer.com* und *OpenERP* findet standardmäßig über eine ungesicherte Verbindung statt. Da bei der Synchronisation sensible Daten übertragen werden, muss die Kommunikation gesichert werden.

Um eine sichere Kommunikation zwischen *OpenERP* und *Jobmixer.com* zu gewährleisten, wird eine durch einen SSH[40]-Tunnel gesicherte Verbindung aufgebaut. Das bedeutet, dass bei jeder Synchronisation ein gesicherter Tunnel geöffnet werden muss und nach Abschluss der Synchronisation dieser wieder geschlossen wird.

[40] Secure Shell (SSH)

Kapitel 5

Implementierung

5.1 Implementierung der Datenbanktabellen

Um die Datenbanktabellen für die Synchronisation zu nutzen, ist es nicht notwendig eine neue Datenbank anzulegen. Die Tabellen werden der bereits existierenden Datenbankstruktur von *Jobmixer.com* hinzugefügt. Änderungen an den bereits bestehenden Tabellen werden bei diesem Schritt nicht durchgeführt.

5.1.1 Erstellen des Models

Zu Beginn werden die Tabellen definiert, das bedeutet, der Aufbau der Datenbanktabellen sowie die Relationen werden im YAML-Format angelegt. Die Datei, in der die Tabellendefinitionen gespeichert werden, heißt `schema.yml` und ist im Verzeichnis `config/doctrine` von `iaOpenErpPlugin` abgelegt. Auflistung 5.1 zeigt als Beispiel die Notierung für die Tabelle `oep_changes`. Auf die gleiche Art und Weise werden auch die anderen Tabellen definiert. Wenn die Angaben in der Datei `schema.yml` durch *Doctrine* verarbeitet sind, werden aus den beschriebenen Tabellen Klassen erzeugt, mit denen es möglich ist, die Datenbanktabellen wie Objekte zu behandeln.

```
Changes:
  tableName: oep_changes
  # Folgende Zeile gibt an, dass zu den hier definierten Feldern
  # noch die Felder "updated_at" und "created_at" dazukommen.
  # Auf diese Weise kann erfasst werden, wann ein Eintrag erzeugt
  # bzw. geaendert wurde.
  actAs: [Timestampable]
  columns:
    identification_map_id:
      type:         integer(11)
    object_map_id:
      type:         integer(11)
    state:
      type:         integer(4)
    sf_guard_user_id:
      type:         integer(11)
  relations:
    ChangesIdentificationMap:
      class:        IdentificationMap
      foreignType:  one
      foreignAlias: Change
    ChangesObject:
      class:        ObjectMap
    JobmixerUser:
      class:        sfGuardUser
```

Listing 5.1: Notierung der `Changes`-Tabelle in der Datei schema.yml

5.1.2 Migration der Datenbank

Durch das Erstellen des Models für die Tabelle kann diese als Objekt angesprochen werden. Um Daten physisch zu speichern, ist es zusätzlich notwendig, die entsprechende Datenbanktabelle anzulegen. Dazu bietet *Doctrine* die Möglichkeit, Datenbanken zu migrieren[41].

Wie bereits erwähnt, werden bei dieser Migration keine vorhandenen Tabellen geändert, sondern lediglich neu hinzugefügt. Auflistung 5.2 zeigt den Quelltext, welcher notwendig ist um die `Changes`-Tabelle mithilfe einer Migration zu erstellen.

[41] Eine vollständige Anleitung zum Thema Migration gibt es unter http://www.doctrine-project.org/documentation/manual/1_1/en/migrations

```
1  public function up()
2  {
3    $this->createTable('oep_changes',
4    array(
5      'id' => array(
6                  'type' => 'integer',
7                  'primary' => true ,
8                  'autoincrement'=>true
9                ),
10     'identification_map_id' => array(
11                 'type' => 'integer',
12                 'length' => 11
13               ),
14      'object_map_id' => array(
15                 'type' => 'integer',
16                 'length' => 11
17               ),
18     'sync_group' => array(
19                 'type' => 'integer',
20                 'length' => 11
21               ),
22     'sf_guard_user_id' => array(
23                 'type' => 'integer',
24                 'length' => 11
25               ),
26     'state' => array(
27                 'type' => 'integer',
28                 'length' => 4
29               ),
30     'created_at' => array(
31                 'type' => 'timestamp'
32               ),
33     'updated_at' => array(
34                 'type' => 'timestamp'
35               )
36   ), array(
37        'primary'=>array(0=>'id')
38      )
39   );
40 }
```

Listing 5.2: Quelltext für das Hinzufügen der `Changes`-Tabelle mithilfe der *Doctrine*-Migration

5.2 Die Ordnerstruktur von iaOpenErpPlugin

Die Ordnerstruktur für das Plugin entspricht der Empfehlung des *symfony*-Handbuches [42] für den Aufbau eines Plugins. Das bedeutet, dass die Namen der Ordner und deren Inhalt der Standardeinstellung von*symfony* entsprechen.

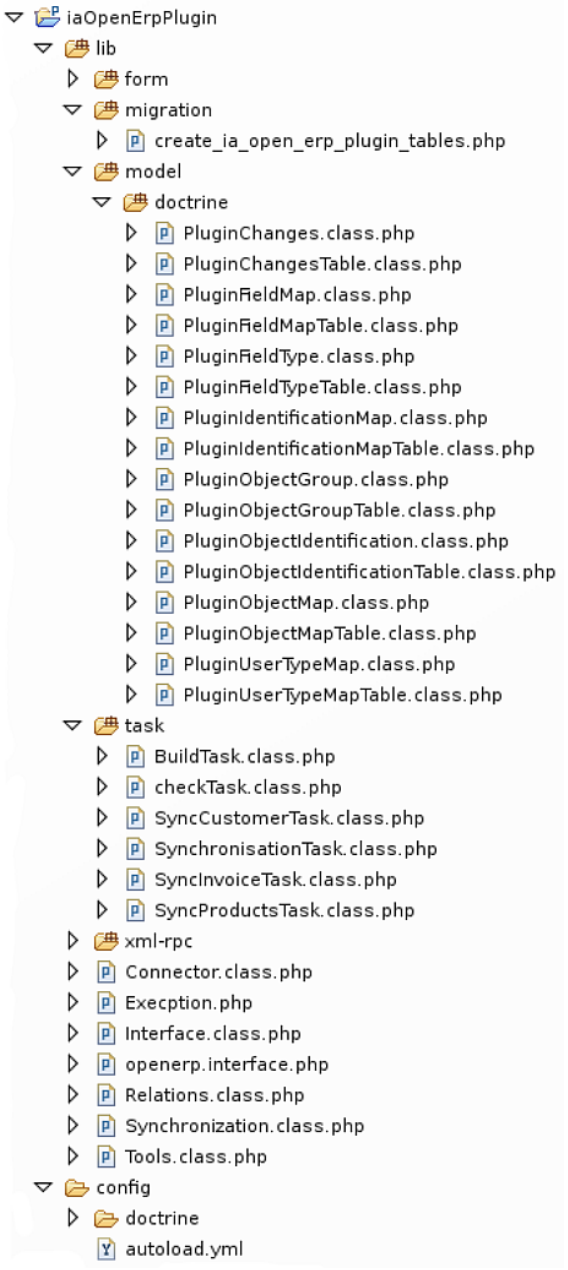

Abbildung 5.1: Ordnerstruktur von iaOpenErpPlugin

[42] [12, vgl. Kapitel 17 Abschnitt Plug-In File Structure].

5.2.1 Das lib-Verzeichnis

In diesem Verzeichnis befinden sich alle für die Synchronisation verwendeten Bibliotheken und Klassen. Diejenigen Klassen, die sich um den Aufbau der Verbindung kümmern oder die Synchronisation unterstützen, liegen direkt auf dieser Ebene. Alle übrigen Klassen sind nach Verwendung geordnet, auf Ordner aufgeteilt.

Im Ordner `migration` befinden sich die Migrationsdateien, welche für die Datenbankanpassung geschrieben werden. Diese werden nicht direkt in diesem Ordner ausgeführt, sondern müssen vorher nach `/lib/migration/doctrine` des *symfony*-Verzeichnisses kopiert werden. Dieser Ordner dient somit nur dem Sammeln der Dateien für die Verbreitung des Plugins.

Der Ordner `model` beinhaltet die von *Doctrine* erstellten Model- und Controller-Klassen. Diese abstrakten Klassen stellen einen Teil der Geschäftslogik und die gesamte Datenlogik für das Plugin bereit. Die Klassen sind ein fester Bestandteil des Plugins. Das Erweitern oder Überschreiben der Klassen ist im `/lib/model/doctrine/iaOpenErpPlugin` des *symfony*-Verzeichnisses möglich. *Doctrine* legt darin beim Erzeugen der Modelklassen automatisch die dazu notwendigen Klassen an.

Diejenigen Tasks, die zum Auslösen und Einstellen der Synchronisation genutzt werden, befinden sich im `task`-Ordner. Die Erklärung der einzelnen Tasks in diesem Ordner erfolgt in späteren Abschnitten.

Die Klassen, die es möglich machen, die *XML-RPC*-Schnittstelle von *OpenERP* über PHP zu nutzen, befinden sich im Ordner `xml-rpc`.

5.2.2 Das config-Verzeichnis

In diesem Verzeichnis sind Teile der Konfigurationsdateien abgelegt, die das `iaOpenErpPlugin` benötigt. Das ist einerseits die in Abschnitt 5.1.1 bereits erwähnte Datei `schema.yml`, die für die Beschreibung der Datenbanktabellen zuständig ist, und andererseits die Datei `autoload.yml`. Darin werden die Teile des Plugins angegeben, die bei dessen Aufruf automatisch geladen werden sollen.

5.3 Nutzung der *XML-RPC*-Schnittstelle von *OpenERP*

Dieser Abschnitt zeigt, wie der Zugriff auf die *XML-RPC*-Schnittstelle von *OpenERP* implementiert wurde und welche Verwendungsmöglichkeiten sich dadurch ergeben.

5.3.1 *XML-RPC* mit PHP nutzen

Die Verwendung von *XML-RPC*-Schnittstellen ist nicht standardmäßig in PHP integriert. Aus diesem Grund wird die freie PHP-Bibliothek *XML-RPC for PHP*[43] verwendet. Die eingesetzte Version hat die Nummer 2.2.1[44].

Die Bibliothek liegt in einem separaten Ordner innerhalb des `iaOpenErpPlugin`-Verzeichnisses. Über die *symfony*-Funktion zum automatischen Laden von Klassen (vgl. 2.3) wird die Bibliothek in *symfony* eingebunden. Dazu ist es notwendig, im `iaOpenErpPlugin`-Konfigurationsverzeichnis die entsprechende Konfigurationsdatei `autoload.yml` anzulegen und die `.inc`-Dateien der Bibliothek zum automatischen Laden anzumelden.

5.3.2 Die Verbindung zwischen *Jobmixer.com* und *OpenERP*

Um die Daten zwischen *OpenERP* und *Jobmixer.com* zu synchronisieren, muss eine Verbindung zwischen beiden Systemen hergestellt werden. Das Bereitstellen der Verbindung und deren Authentifizierung wird von `iaOpenErpPlugin` übernommen. Der dazu erforderliche URL[45] für *OpenERP* setzt sich aus den in der `app.yml`-Konfigurationsdatei abgelegten Verbindungsdaten zusammen. Diese werden zum Bereitstellen der Verbindung durch das Plugin eingelesen. An dieser Stelle wird jedoch nicht die eigentliche Verbindung aufgebaut, da noch nicht bekannt ist, zu welchem Zweck diese aufgebaut wird. Es wird vielmehr nur das Grundgerüst zur Herstellung einer Verbindung erzeugt. Der Ablauf für den Aufbau des Grundgerüstes der Verbindung ist in Abbildung 5.2 dargestellt.

Der erste Schritt für den Verbindungsaufbau ist das Erzeugen einer Instanz von `OpenErpInterface`. Dabei wird vom Konstruktor von `OpenErpInterface` eine Instanz von `OpenErpConnector` erzeugt. Wenn dies geschehen ist, wird durch den Aufruf von `getConnection()` eine Verbindung zu *OpenERP* vorbereitet.

[43] http://phpxmlrpc.sourceforge.net/
[44] Stand 13. 05. 2009.
[45] Uniform Ressource Locator (URL)

Die Authentifizierung der Verbindung erfolgt durch den Aufruf der `connect()`-Methode innerhalb von `getConnection()`. Die Methode baut eine Verbindung zu einem speziellem URL des *OpenERP*-Server auf und sendet die Benutzerdaten zur Authentifizierung. Sind die Benutzerdaten korrekt, wird die `Id` des *OpenERP*-Benutzers zurück gegeben. Durch Angabe der `Id` bei späteren Verbindungen wird der Benutzer authentisiert. Der Rückgabewert der `getConnection()`-Methode ist eine Instanz der `xmlrpcmsg`-Klasse der *XML-RPC for PHP*-Bibliothek.

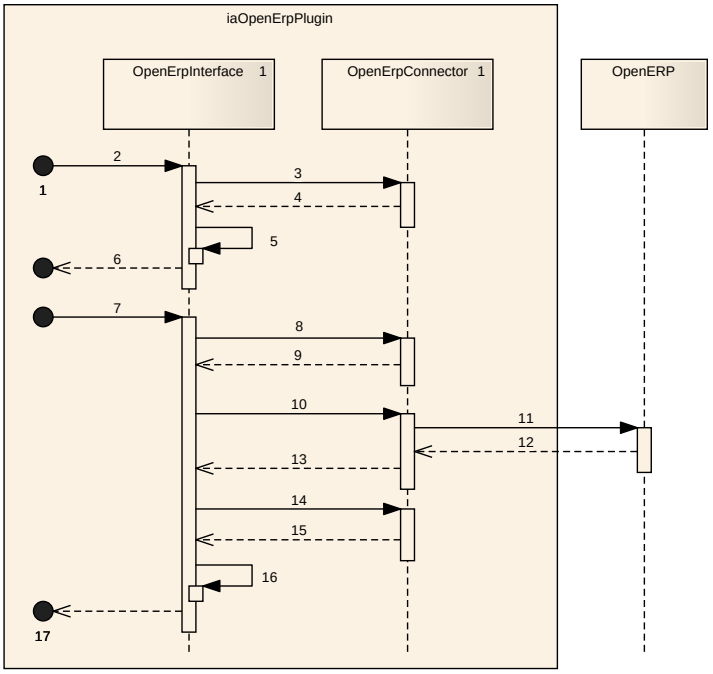

Abbildung 5.2: Aufbau einer Verbindung zwischen *Jobmixer.com* und *OpenERP* über die *XML-RPC*-Schnittstelle

1. Es wird eine Verbindung zu *OpenERP* angefordert.
2. Durch den Aufruf von `OpenErpInterface::getInstance()` wird eine Instanz von `OpenErpInterface` angefordet.
3. Im Konstruktor von `OpenErpInterface` wird eine Instanz von `OpenErpConnector` angefordert.
4. Es wird eine Instanz von `OpenErpConnector` zurückgegeben.
5. Der Konstruktor erzeugt eine neue `OpenErpInterface`-Instanz.
6. Die `OpenErpInterface`-Instanz wird zurückgegeben.

7. Um das Grundgerüst zu erzeugen wird die `getConnection()`-Methode von `OpenErpInterface` aufgerufen.

8. Die Methode `getDatabaseName()` fordert den Namen der *OpenERP*-Datenbank für die Verbindung von `OpenErpConnector` an.

9. Die Klasse `OpenErpConnector` liefert den Datenbanknamen zurück.

10. Um den Benutzer zu authentifizieren, wird die `connect()`-Methode von `OpenErpConnector` aufgerufen.

11. Die `connect()`-Methode verbindet sich zu *OpenERP* und lässt den Benutzer authentisieren.

12. *OpenERP* gibt nach der Authentisierung des Benutzers ein `xmlrpcval`-Objekt mit dessen `Id` zurück. Diese dient zusammen mit dem Passwort der Authentifizierung des Benutzers bei weiteren Verbindungen.

13. Nachdem die `connect()`-Methode das von *OpenERP* zurückgegebene `xmlrpcval`-Objekt ausgelesen hat, gibt sie die `Id` zurück.

14. Die Methode `getPassword()` fordert das Passwort für die Verbindung von `OpenErpConnector` an.

15. Die Klasse `OpenErpConnector` liefert das Passwort zurück.

16. Die `getConnection()`-Methode liefert ein `xmlrpcmsg`-Objekt mit dem fertigen Grundgerüst für die Verbindung zu *OpenERP* zurück.

17. Mithilfe des Grundgerüstes ist es jetzt möglich, *OpenERP*-Methoden über die *XML-RPC*-Schnittstelle aufzurufen.

5.3.3 Aufruf von *OpenERP*-Methoden über *XML-RPC*

Das Aufrufen von *OpenERP*-Methoden über *XML-RPC* erfolgt durch die `execute()`-Methode (Listing 5.3) von `OpenErpInterface`. Als Parameter werden der Name des *OpenERP*-Objektes, der Name der aufzurufenden Methode und die Argumente in Form eines Arrays für die Methode übergeben. Die Authentifizierung der Verbindung übernimmt, wie in Abschnitt 5.3.2 beschrieben, die `getConnection()`-Methode.

Die übergebenen Parameter werden, durch die `execute()`-Methode, einer von `getConnection()` zurückgegebenen Instanz der `xmlrpcmsg`-Klasse hinzugefügt. Wenn

alle Parameter angehangen sind, wird mithilfe einer `xmlrpc_client`-Instanz eine Anfrage an *OpenERP* gesendet.

Das zurückgegebene `xmlrpcresp`-Objekt wird ausgewertet und – wenn kein Fehler aufgetreten ist – in ein Array umgewandelt. Dieses Array wird dann zur weiteren Verarbeitung an die aufrufende Methode zurückgegeben.

```
1  public function execute($object_name, $method_name, $args = null)
2  {
3    $msg = $this->getConnection();
4    $msg->addParam(new xmlrpcval($object_name, "string"));
5    $msg->addParam(new xmlrpcval($method_name, "string"));
6    if (true === is_array($args))
7    {
8      foreach ($args as $arg_type => $arg)
9      {
10       if ( true === is_integer($arg_type) and
11            true === is_array($arg) )
12         {
13           $arg_type = 'array';
14         }
15         $msg->addParam(new xmlrpcval($arg, $arg_type));
16     }
17   }
18   $resp = $this->openErpConnetion->getClient()->send($msg);
19   if (0 === $resp->value() )
20   {
21     throw new iaOpenErpExecption($resp->faultString());
22   }
23   $val = $resp->value()->scalarval();
24   return $val;
25 }
```

Listing 5.3: Die `execute()`-Methode dient zum Aufrufen von *OpenERP*-Methoden über *XML-RPC*

5.3.4 Datensuche mit *XML-RPC*

Zum Suchen von Daten in *OpenERP* steht über *XML-RPC* die `search()`-Methode zur Verfügung. Die Nutzung der Methode wird in der `OpenErpInterface`-Klasse gekapselt. Der Aufruf der Suche sieht folgendermaßen aus:

```php
/* Sucht das Produkt mit der Id 42 in OpenERP heraus  */
OpenErpInterface::getInstance()->
search(
  /* OpenERP Klassenname */
  'product.product',
  array(
    /* <Name des Suchfeldes> => <Suchwert> */
    'id' => 42
  )
);

/* Sucht alle Produkte aus OpenERP heraus, deren Id nicht 42 ist */
OpenErpInterface::getInstance()->
search(
  'product.product',
  array(
    /* Name des Suchefeldes => Suchwert */
    'id' => 42,
    /* Suchoperator Standard: "=" */
    '!='
  )
);
```

Listing 5.4: Aufruf der `search()`-Methode

Da die `search`-Methode nicht existiert, wird der Aufruf von der `__call()`-Methode abgefangen. Diese erzeugt mithilfe von `getSearchArray` ein für die Suche formatiertes Array. Dieses wird dann mit dem Namen des *OpenERP*-Objektes und `search()` als Methodenname der `execute()`-Methode als Parameter übergeben. Wenn keine Fehler aufgetreten sind, liefert diese ein Array mit den `Ids` der gefundenen Objekte zurück.

5.3.5 Datentransfer über *XML-RPC*

Für den Datentransfer stellt jedes *OpenERP*-Objekt eine `read()`, `create()` und `write()`-Methode[46] zur Verfügung. Diese werden ebenfalls mithilfe der `__call()`-Methode gekapselt. Da die jeweilige Kodierung der *Jobmixer.com*-und der *OpenERP*-Datenbank auf `UTF-8` eingestellt ist, muss die `$GLOBALS['xmlrpc_internalencoding']`-Einstellung der *XML-RPC for PHP*-Bibliothek von `ISO-8859-1` auf `UTF-8` umgestellt werden.

[46] Die `write()`-Methode wird von `iaOpenErpPlugin` als `update()`-Methode zur Verfügung gestellt.

Die `read()`-Methode

Mithilfe der `read()`-Methode können die Daten von *OpenERP*-Objekten ausgelesen werden. Dazu muss als erster Parameter der Klassenname für die auszulesende *OpenERP*-Tabelle angegeben werden. Als zweiter Parameter wird ein Array erwartet, welches ein Array mit den ausgelesenen `Ids` der Datensätze und ein Array, das angibt, welche Felder abgefragt werden sollen, enthält. Das `id`-Feld wird immer ausgelesen, auch wenn es nicht angegeben wurde. Der Rückgabewert ist ein Array aus `xmlrpcval`-Objekten, die ausgelesen wurden.

```
1  /* Auslesen von Name und Preis von dem Produkt mit den Ids 42 und 23 */
2  OpenErpInterface::getInstance()->
3  read
4  (
5    /* OpenERP Klassenname */
6    'product.product',
7    array(
8      array(
9        array(
10         /* "xmlrpcval"-Objekte mit Ids der auszulesenden Objekte */
11         new xmlrpcval(23, 'int'),
12         new xmlrpcval(42, 'int'),
13       )
14       array(
15         /* "xmlrpcval"-Objekte mit den auszulesenden Feldern */
16         new xmlrpcval('name', 'string'),
17         new xmlrpcval('list_price', 'double'),
18       )
19     )
20   )
21 );
```

Listing 5.5: Aufruf der `read()`-Methode

Die create()-Methode

Der Aufruf der `create()`-Methode erzeugt ein neues *OpenERP*-Objekt von dem im ersten Parameter angegebenen Objektnamen. Der zweite Parameter enthält die Daten von *Jobmixer.com*, die in dem neuen Objekt gespeichert werden sollen.

```
/* Erzeugen eines neuen OpenERP Produktes */
OpenErpInterface::getInstance()->
create(
  /* OpenERP Klassenname */
  'product.product',
  array(
    array(
      /**
       * "xmlrpcval"-Objekte mit den Daten des Jobmixer.com-Objektes
       * '<OpenERP Feldname>' => <Inhalt des Feldes als "xmlrpcval"-Objekt>
       */
      'name' = new xmlrpcval('Testprodukt', 'string'),
      'price' = new xmlrpcval(1984, 'double'),
    )
  )
);
```

Listing 5.6: Aufruf der `create()`-Methode

Das übergebene Array wird innerhalb der `__call()`-Methode von `OpenErpInterface` durch ein weiteres Array gekapselt, als Typ wird `struct` (vgl. Tabelle 2.1) angegeben. Dies ist notwendig, da Arrays, welche die zu speichernden Daten enthalten, den Typ `struct` haben müssen.

Die `update()`-Methode

```
/* Aktualisieren von Name und Preis des Produktes mit der Id 42 */
OpenErpInterface::getInstance()->
update(
  /* OpenERP Klassenname */
  'product.product',
  array(
    array(
      /* "xmlrpcval"-Objekt mit der Id des zu aktualisierenten OpenERP Objektes */
      new xmlrpcval(42, 'int')
      array
      (
        /**
         * "xmlrpcval"-Objekte mit den Daten des Jobmixer.com-Objektes
         * '<OpenERP Feldname>' => <Inhalt des Feldes als "xmlrpcval" Objekt>
         */
        'name' = new xmlrpcval('Testprodukt aktualisiert', 'string'),
        'price' = new xmlrpcval(1337, 'double'),
      )
    )
  )
);
```

Listing 5.7: Aufruf der `update()`-Methode

Die `update()`-Methode kapselt die `write()`-Methode von *OpenERP* und aktualisiert ein bereits bestehendes *OpenERP*-Objekt. Als Parameter werden der Name des Objektes, die `Id` des *OpenERP*-Objektes und die aktualisierten Daten in Form eines Arrays erwartet. Die aktualisierten Daten werden, wie bei der `create`-Methode, durch die `__call()`-Methode von `OpenErpInterface` gekapselt.

5.4 Erfassen von Änderungen

Zum Erfassen von Änderungen werden vom `iaOpenErpPlugin` zwei Möglichkeiten bereitgestellt. Die erste Möglichkeit erfasst die Änderung abhängig von einem *Jobmixer.com*-Objekt. Im Gegensatz dazu benötigt die zweite Möglichkeit nur den Namen der *Jobmixer.com*-Klasse die synchronisiert werden soll. Die folgenden beiden Abschnitte werden diese Möglichkeiten genauer erläutern.

5.4.1 Erfassen einer Änderung mit *Jobmixer.com*–Objekt

```
1   Changes::register(<Jobmixer.com Objekt>);
```

Listing 5.8: Registrieren einer Änderung mit *Jobmixer.com*-Objekt

Diese Art der Erfassung wird für Änderungen benutzt, die auf *Jobmixer.com* gemacht wurden und zu *OpenERP* synchronisiert werden sollen. Nachdem die `register()`-Methode aufgerufen und das aktualisierte Objekt übergeben ist, werden die passenden Einträge in der `oep_object_map`-Tabelle herausgesucht. Für jedes zu der Änderung passende Objekt wird überprüft, ob es bereits für die Synchronisation vorgemerkt ist. Wenn das der Fall ist, wird keine weitere Aktion ausgeführt. Falls das Objekt jedoch noch nicht registriert ist, wird überprüft, ob das Objekt vorher schon einmal synchronisiert wurde. Ist dies der Fall, wird es mit dem Status `SYNC_STATE_UPDATE`[47] für die Synchronisation vorgemerkt. Wenn es noch nicht registriert wurde, erhält es den Status `SYNC_STATE_NEW` und wird für die Synchronisation vorgemerkt. Dabei wird für das Objekt auch ein Eintrag in der `oep_identification_map`-Tabelle erstellt. Dieser Eintrag wird für die Zuordnung von *OpenERP*-Objekten und *Jobmixer.com*-Objekten benötigt und für die Überprüfung, ob das Objekt schon einmal synchronisiert wurde.

[47] Eine Übersicht über die verschiedenen Status liefert Tabelle 4.2 auf Seite 32.

5.4.2 Erfassen einer Änderung ohne *Jobmixer.com*-Objekt

Wenn ein Objekt von *OpenERP* nach *Jobmixer.com* synchronisiert werden muss und es noch nicht existiert, dann kann diese Art der Erfassung genutzt werden. Als Parameter muss nur der Name der *Jobmixer.com*-Klasse angegeben werden, mit der synchronisiert werden soll. Auch hier wird überprüft, ob das Objekt bereits zur Synchronisation angemeldet ist. Wenn dies nicht der Fall ist, wird ein entsprechender Eintrag in der `oep_changes`-Tabelle gemacht.

```
1 Changes::emptyRegister(<Jobmixer.com Objekt>);
```

Listing 5.9: Registrieren einer Änderung ohne *Jobmixer.com*-Objekt

5.4.3 Erkennen von bereits synchronisierten Objekten

Ob ein *Jobmixer.com*-Objekt bereits synchronisiert wurde, lässt sich mithilfe der `oep_identification_map`-Tabelle feststellen. Wie in Abschnitt 4.6.2 bereits erwähnt, findet an dieser Stelle das Mapping der Objekte statt. Die `isJobmixerObjectRegistrated()`-Methode der `PluginIdentificationMapTable`-Klasse nutzt dies, um festzustellen ob ein Objekt bereits synchronisiert wurde. Dazu überprüft sie, ob ein Objekt bereits eingetragen ist.

5.5 Ausführen der Synchronisation

Um die Synchronisation auszuführen, stellt die `OpenErpSynchronization`-Klasse zwei Methoden bereit: die `start()`-Methode (Listing 5.10) und die `run()`-Methode (Listing 5.11). Die `start()`-Methode liest alle `Changes`-Objekte für die übergebene Richtung unter Verwendung der `getReadyChangesByDirection()`-Methode aus und übergibt jede einzelne Änderung an die `run()`-Methode.

```
1  public function start( $direction )
2  {
3    switch ($direction)
4    {
5      case ObjectMap::DIRECTION_UP:
6        $changes = Doctrine::getTable('Changes')->
7                   getReadyChangesIdsByDirection(ObjectMap::DIRECTION_UP);
8        $direction_name = 'up';
9        break;
10     case ObjectMap::DIRECTION_DOWN:
11       $changes = Doctrine::getTable('Changes')->
12                  getReadyChangesIdsByDirection(ObjectMap::DIRECTION_DOWN);
13       $direction_name = 'down';
14       break;
15   }
16   foreach ($changes as $change)
17   {
18     $this->run($direction_name, $change);
19   }
20 }
```

Listing 5.10: Methode zum Starten der Synchronisation in die angegebene Richtung

Mit der übergebenen Änderung und Richtung ruft die `run()`-Methode durch Verwendung der `ReflectionClass`-Klasse von PHP die entsprechende Synchronisationsklasse auf. Durch die Verwendung der `ReflectionClass`-Klasse ist es möglich, zu überprüfen, ob die entsprechende Klasse instanziert werden kann. Ist das der Fall, wird die entsprechende Methode für die Synchronisierung des Objektes mithilfe der `ReflectionMethod`-Klasse aufgerufen. Danach übernehmen die aufgerufene Synchronisationsklasse und die entsprechende Methode die Synchronisation der Änderung. Dies wird solange durchgeführt, bis alle Änderungen abgearbeitet sind.

```
1  public function run( $direction_name, PluginChanges $change )
2  {
3    $class = new ReflectionClass($change->getGroup()->getName() . 'Synchronisation');
4    if ($class->isInstantiable())
5    {
6      $method_name = $direction_name .
7                     OpenErpPluginTools::camelize($change->
8                                                  getChangesObject()->
9                                                  getOpenerpObject());
10     $method = new ReflectionMethod($change->getGroup()->getName() .
11                                    'Synchronisation', $method_name);
12     $method->invoke($class->newInstance(), $change);
13   }
14 }
```

Listing 5.11: Aufruf der entsprechenden Synchronisationsmethode

5.6 Helfer für die Synchronisation

Um die Synchronisation zu vereinfachen, gibt es Methoden, die wiederkehrende Abläufe kapseln. Für diese Methoden steht die `OpenErpPluginTools`-Klasse zur Verfügung, in der alle diese Methoden gesammelt werden. Für einzelne Zwecke kann es notwendig sein, diese Helfer direkt an ein Objekt zu binden. In solchen Fällen sind die Methoden direkt für das Objekt implementiert. Dieser Abschnitt gibt einen Überblick über die Methoden und ihre Aufgaben.

5.6.1 Daten formatiert für die Synchronisation bereitstellen

Für die Synchronisation zu *OpenERP* ist es notwendig, dass alle Daten desjenigen *Jobmixer.com*-Objektes das synchronisiert werden soll, in das richtige Format umgewandelt werden. Um diesen Prozess zu vereinfachen, besitzt jedes `Change`-Objekt die Methode `getDataToStore()`. Diese Methode sucht alle für das Objekt in der Datei `openerp.yml` angegebenen Felder heraus und formatiert sie entsprechend, wie es für die Synchronisation notwendig ist (Listing 5.14).

```
1 array
2 (
3   /**
4   * "xmlrpcval"-Objekte mit den Daten des Jobmixer.com-Objektes
5   * '<OpenERP Feldname>' => <Inhalt des Feldes als "xmlrpcval" Objekt>
6   */
7   'name' = new xmlrpcval('Testprodukt', 'string'),
8   'price' = new xmlrpcval(2342, 'double'),
9 )
```

Listing 5.12: Formatierte *Jobmixer.com*-Daten für die Synchronisation

5.6.2 Konvertierung von *XML-RPC*-Object-Arrays

Zum einfacheren Verarbeiten der (von der `read()`-Methode) zurückgelieferten Daten, ist es hilfreich das erhaltene `xmlrpcval`-Objekt, welches ein Array mit weiteren `xmlrpcval`-Objekten enthält, in ein Array ohne `xmlrpcval`-Objekte umzuwandeln. Für diesen Zweck stellt die `OpenErpPluginTools`-Klasse die statische Methode `convertXmlRpcObjectArrayToArray()` zur Verfügung. Diese erwartet als Parameter das `xmlrpcval`-Objekt und liefert ein Array ohne `xmlrpcval`-Objekte zurück.

```
/*
 * Aufbau eines xmlrpcval-Objektes, dass ein Array
 * aus xmlrpcval-Objekten enth lt.
 */
xmlrpcval Object
(
  [me] => Array
    (
      [struct] => Array
        (
          [id] => xmlrpcval Object
            (
              [me] => Array
                (
                  [int] => 11
                )
              [mytype] => 1
              [_php_class] =>
            )
          [name] => xmlrpcval Object
            (
              [me] => Array
                (
                  [string] => Lebenslauf Erstellung
                )
              [mytype] => 1
              [_php_class] =>
            )
        )
    )
  [mytype] => 3
  [_php_class] =>
)

/*
 * Das xmlrpcval-Objekt nachdem es durch
 * convertXmlRpcObjectArrayToArray() umgewandelt wurde.
 */
array
  'id' => int 42
  'name' => string 'Testprodukt' (length=11)
```

Listing 5.13: Das `xmlrpcval`-Objekt vor der Umwandlung durch `convertXmlRpcObjectArrayToArray()` und das daraus erzeugte Array.

5.6.3 Überprüfung des Datentypes der zurückgegebenen Daten

Für den Fall, dass Felder in *OpenERP* nicht ausgefüllt sind, erscheinen diese nach der Synchronisation leer. Da *Doctrine* bei diesen Feldern den Typ nicht erkennt und sie da-

durch nicht gespeichert werden können, bekommen diese Felder den Wert `NULL` zugewiesen. Auf diese Weise können die Felder durch *Doctrine* verarbeitet werden.

Um zu überprüfen, ob die von der `read()`-Methode zurückgelieferten Daten nicht leer sind, existiert in der `OpenErpPluginTools`-Klasse die `validateAndCorrectFieldValueDataTypes()`-Methode. Als Parameter erwartet sie eine `Doctrine_Collection` mit `FieldMaps`, die überprüft werden sollen, und ein Array in dem die Felder und deren Daten enthalten sind. Die Methode überprüft sämtliche Felder auf den richtigen Datentyp. Sollte das Feld leer sein, wird es auf `NULL` gesetzt. Der Rückgabewert ist ein Array mit den korrigierten Daten.

```
1 OpenErpPluginTools::validateAndCorrectFieldValueDataTypes
2 (
3   <Doctrine_Collection mit allen Feldern fuer das synchronisierte Objekt>,
4   <Array mit den Daten aus OpenERP die ueberprueft werden sollen>
5 )
```

Listing 5.14: Aufruf von `validateAndCorrectFieldValueDataTypes()`

5.6.4 Das zur Änderung passende *Jobmixer.com*-Objekt erhalten

Jedem aus einem *Jobmixer.com* erzeugten `Change`-Objekt ist es möglich, über die `getChangedJobmixerRecord()`-Methode das passende *Jobmixer.com*-Objekt zu erhalten. Die Methode sucht anhand der in `oep_identification_map` gespeicherten *Jobmixer.com*-`Id` und des zum `Change`-Objekt gehörenden `ObjectMap`-Objekt das entsprechende *Jobmixer.com*-Objekt.

5.7 Die Konfigurationsdatei `openerp.yml`

Die Konfigurationsdatei ist, wie die Dateiendung bereits erkennen lässt, im YAML-Format angelegt. Das bedeutet, dass bei Einträgen die entsprechende Syntax beachtet werden muss. Listing 5.15 zeigt die Struktur der Konfigurationsdatei. Die Reihenfolge der einzelnen Oberpunkte[48], kann beliebig gewählt werden.

[48] Oberpunkte sind `Objects`, `UserType`, `ObjectGroup` und `FieldType`.

```
Objects:
  <OpenERP Objekt Name>:
    group:     <Name der Gruppe zu der das Objekt gehoert>
    direction: <Angabe der Richtung in die das Objekt synchronisiert werden soll>
    fields:
      <Jobmixer.com Objekt>:
        UserType:               <Jobmixer.com Benutzergruppe>
        <OpenERP Feldname>:
          jobmixer_name:        <Name des Jobmixer.com Feldes>
          type:                 <Angabe des Types>

UserType:
  <Jobmixer.com Benutzergruppe>: <Id der entsprechenden OpenERP Benutzergruppe>

ObjectGroup:
  <Objektgruppen>

FieldType:
  <Dateitypen fuer die Felder>
```

Listing 5.15: Struktur der Konfigurationsdatei

5.7.1 Die Bereiche `UserType`, `ObjectGroup` und `FieldType`

Mit den hier angegebenen Werten werden die entsprechenden Tabellen (`oep_user_type`, `oep_object_group` und `oep_field_type`) gefüllt. Der Aufbau und die Aufgaben der Tabellen, wurden bereits in Abschnitt 4.6.2 erklärt. Das Beispiel in Listing 5.16 zeigt, wie die Einstellungen angegeben werden müssen.

5.7.2 Der Bereich `Objects`

Unter `Objects` wird das Mapping der Objekte und Felder konfiguriert. Die Angaben für `group` und `UserType` beziehen sich auf die Einträge in `ObjectGroup` und `UserType`. Der `group` Eintrag bei `Objects` (in Beispiel 5.16) bezieht sich auf einen der Einträge von `ObjectGroup`. Das doppelte Angeben der Werte hilft, Fehler bei den Einstellungen zu vermeiden, da es nicht möglich ist, die Einstellungen für ein Objekt zu speichern, wenn diese Angaben nicht übereinstimmen.

Bei der Angabe von Feldern ist es möglich, den Inhalt des *OpenERP*-Feldes aus mehreren *Jobmixer.com*-Feldern zusammenzustellen. Das Zusammenfügen von mehreren *Jobmixer.com*–Feldern muss mit einem „+“ oder einem „$\star$“ gekennzeichnet werden. Wenn

ein „+“ angegeben wird, dann werden die Inhalte der beiden Felder mit einem Leerzeichen getrennt und eingetragen. Bei einem „⋆“ werden die Inhalte ohne Trennzeichen eingefügt.

Tabelle 5.1: Synchronisationsrichtungen

Synchronisationsrichtung	numerischer Wert	Konstante
Jobmixer.com → *OpenERP*	1	DIRECTION_UP
OpenERP → *Jobmixer.com*	2	DIRECTION_DOWN
beide Richtungen	3	DIRECTION_BOTH

Als `direction` wird ein numerischer Wert angegeben, der festlegt, in welche Richtung das Objekt synchronisiert werden soll. In Tabelle 5.1 werden alle Richtungen mit den dazugehörigen numerischen Werten aufgelistet. Das Objekt aus dem Beispiel soll dementsprechend in beide Richtungen synchronisiert werden.

```
1  Objects:
2    erp.objekt:
3      group:     Gruppe1
4      direction: 3 # vgl. Tabelle 5.1
5      fields:
6        JobmixerObjektTyp1:
7          UserType:               UserType1
8          name:
9            # der Inhalt von jobmixer_name wird beim Auslesen
10           # durch ein Leerzeichen getrennt. Bsp: "name 42"
11           jobmixer_name:        jobmixer_name+id
12           type:                 Datentyp1
13       JobmixerObjektTyp2:
14         UserType:               UserType2
15         name:
16           # der Inhalt von jobmixer_name wird beim Auslesen
17           # zusammen geschrieben. Bsp: "name42"
18           jobmixer_name:        jobmixer_name|id
19           type:                 Datentyp1
20
21 UserType:
22     #Benutzertyp Bezeichner  OpenERP Id des Benutzertypes
23     UserType1:                          1
24     UserType2:                          2
25
26 ObjectGroup:
27   Gruppe1
28   Gruppe2
29
30 FieldType:
31   Datentyp1
32   Datentyp2
```

Listing 5.16: Die `openerp.yml`-Datei

5.8 Übernehmen der Einstellungen

Um die Einstellungen, die in der Datei `openerp.yml` gemacht wurden, anzuwenden, müssen diese in der *Jobmixer.com*-Datenbank gespeichert werden. Für diese Aufgabe steht der `BuildTask` von `iaOpenErpPlugin` zur Verfügung.

```
1 php symfony openerp:build <Anwendungsname>
```

Listing 5.17: Aufruf des `BuildTasks`

5.8.1 Parsen der Einstellungen

Um die Einstellungen zu erhalten, liest der Task zuerst die Datei `openerp.yml` aus. Dazu wird der von *symfony* bereitgestellte YAML-Parser genutzt (Listing 5.18). Nachdem der Parser die Datei verarbeitet hat, gibt er den Inhalt als Array zurück.

```
1 $yamlArray = sfYaml::load(sfConfig::get('sf_config_dir') . '/openerp.yml');
```

Listing 5.18: Aufruf des *symfony*-YAML-Parsers sfYaml

5.8.2 Löschen veralteter Daten

Anschließend werden die existierenden Einträge von `FieldMap` und `FieldType` gelöscht und durch aktuelle Einträge ersetzt. Die Inhalte der anderen Tabellen dürfen nicht gelöscht werden, da diese durch direkte oder indirekte Relationen mit `IdentificationMap` zusammenhängen. Wenn die Inhalte dennoch gelöscht werden, können die bereits synchronisierten Daten nicht mehr ordentlich zugeordnet werden.

```
1  $tables = array( 'FieldMap' , 'FieldType');
2  $conn = Doctrine_Manager::connection();
3  foreach ( $tables as $table )
4  {
5    $tableToReset = Doctrine::getTable($table);
6    $recordsToDelete = $tableToReset->findAll();
7    $recordsToDelete->delete();
8    $recordsToDelete->save();
9    $statement = $conn->
10   prepare('alter table ' . $tableToReset->getTableName() . ' auto_increment = 1');
11   $statement->execute();
12 }
```

Listing 5.19: Löschen alter Daten in `FieldMap` und `FieldType`

Listing 5.19 zeigt den Teil des Quellcodes, der das Löschen übernimmt. Die Objektnamen der Tabellen, die gelöscht werden sollen, sind in einem Array angegeben. *Doctrine* lädt anhand der Namen die entsprechende Tabelle und löscht alle darin enthaltenen Daten. Zuletzt wird der Zähler für den Primärschlüssel zurückgesetzt.

5.8.3 Abspeichern der Einstellungen in der Datenbank

Anschließend werden die Einstellungen in die Datenbank eingetragen. Dazu werden zuerst die Daten für `UserType`, `ObjectGroup` und `FieldType` gespeichert. Dies geschieht zu Beginn, da sie selbst keine eigene Relation besitzen und für das Erfassen der weiteren Einstellungen notwendig sind. Als nächstes werden die Objekte (`ObjectMap`) und die Felder (`FieldMap`) mit allen Relationen erzeugt und gespeichert. Beim Eintragen der Daten wird darauf geachtet, bereits existierende Einträge nicht erneut einzutragen. Dazu wird, bevor ein Eintrag erzeugt wird, überprüft, ob dieser bereits existiert. Sollte dies der Fall sein, wird das Eintragen abgebrochen und beim nächsten Datensatz fortgesetzt.

5.8.4 Aufräumen der Einstellungen

Wenn das Speichern aller Einträge aus der Datei `openerp.yml` erfolgreich abgeschlossen ist, sind alle Einstellungen übernommen. Es können aber noch alte Einstellungen aus der vorhergehenden Konfiguration vorhanden sein. Davon betroffen sind `ObjectGroup`, `UserType` und `ObjectMap`. Im nächsten Abschnitt des Konfigurationsvorganges werden diese Einträge gelöscht (Listing 5.20). Dazu besitzen die `Table`-Klassen der drei Tabellen, die Methode `deleteOldEntries()`. Dieser Methode wird ein Array mit den aktuellen Einträgen übergeben. Mithilfe dieser Daten überprüft die Methode den Inhalt der Tabelle und löscht alle veralteten Einträge.

```
1 echo "Loesche alte Eintr ge \n";
2 Doctrine::getTable('ObjectGroup')->deleteOldEntries($yamlArray['ObjectGroup']);
3 $userTypeTable->deleteOldEntries($yamlArray['UserType']);
4 $objectMapTable->deleteOldEntries($objectList);
```

Listing 5.20: Aufräumen der Einträge

[48] Der Ausdruck `Table`-Klassen bezieht sich auf die Klassen mit dem Suffix `Table`.

5.8.5 Erzeugen der Synchronisationsklassen und Interfaces

Der letzte Schritt zur vollständigen Konfiguration der Synchronisation, ist das Anlegen der individuellen Klassen für die einzelnen Gruppen der Synchronisation. Wie in Abschnitt 4.7.3 beschrieben, besitzt jede Gruppe eine Klasse, welche den Ablauf der Synchronisation festlegt. Wenn eine Klasse für eine Gruppe noch nicht existiert, wird diese erzeugt. Da sich bei einer Aktualisierung der Konfiguration auch die Methoden dieser Klassen ändern können, muss das Interface, das diese Methoden deklariert, geändert werden. Dazu wird das alte Interface gelöscht und ein neues mit den aktualisierten Methoden erzeugt (Listing 5.21).

```
1  echo "Erstelle Klassen fuer die Synchronisation\n";
2  $abstractPath = sfConfig::get('sf_lib_dir') .
3                  '/model/doctrine/iaOpenErpPlugin/generated/interface';
4  $path = sfConfig::get('sf_lib_dir') .
5                  '/model/doctrine/iaOpenErpPlugin/synchronisation';
6  Doctrine_Lib::makeDirectories($abstractPath);
7  Doctrine_Lib::makeDirectories($path);
8  foreach ( $yamlArray['ObjectGroup'] as $name => $value )
9  {
10     OpenErpPluginTools::createSynchronisationInterface($abstractPath, $name);
11     OpenErpPluginTools::createSynchronisationClass($path, $name);
12 }
```

Listing 5.21: Erstellen oder Aktualisieren der Synchronisationsklassen und der Interfaces

Die erzeugten Dateien werden im Verzeichnis `lib/iaOpenErpPlugin` des *symfony*-Projektes agbespeichert. Die Interfaces werden im Unterordner `generated/interface` abgelegt. Die Klassen für die Synchronisation werden im Unterordner `synchronisation` abgelegt (Abbildung 5.3). Nachdem dies geschehen ist, befindet sich die Konfiguration der Synchronisation auf dem aktuellesten Stand.

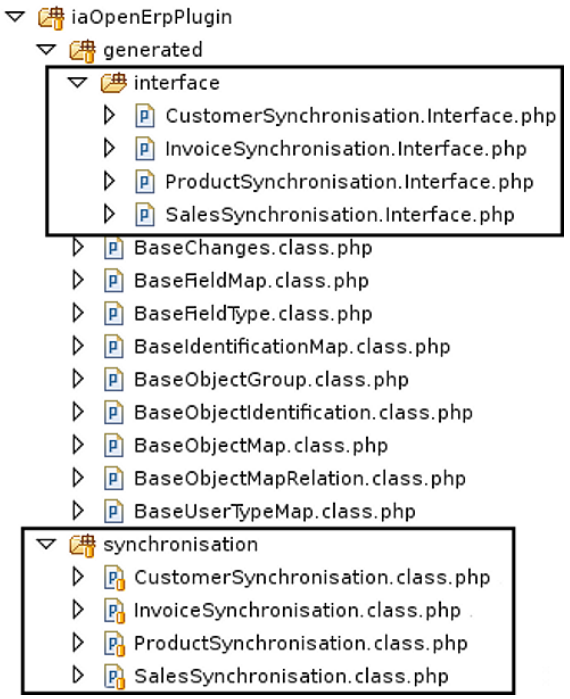

Abbildung 5.3: Ordnerstruktur der automatisch erzeugten Klassen und Interfaces

5.9 Auslösen der Synchronisation

Die Synchronisationsvorgänge werden durch einen *symfony*-Task ausgelöst. Dabei haben die Synchronisationsvorgänge für Rechnungen und Produkte jeweils einen eigenen Task. Die folgenden Abschnitte gehen auf alle zur Verfügung stehenden Tasks und ihre Aufgabe genauer ein.

5.9.1 Der SynchronisationTask

Der `SynchronisationTask` ist der wichtigste Task. Er sorgt dafür, dass die von *Jobmixer.com* eingetragenen Änderungen mit *OpenERP* oder die Änderungen von *OpenERP* mit *Jobmixer.com* synchronisiert werden. Beim Aufruf der Synchronisation in eine der beiden Richtungen (Listing 5.22) wird für jedes zur Synchronisation angemeldete Objekt die dazu passende Synchronisationsklasse und die jeweilige Methode aufgerufen. Wenn z. B. ein neuer Benutzer zur Synchronisation angemeldet wurde, wird die Klasse `CustomerSynchronisation` instanziert und die Methode `downResPartner()` aufgerufen. Mit diesem Task werden hauptsächlich Kundendaten und Bestellungen synchronisiert.

```
1 php symfony openerp:sync <Anwendungsname> up   # OpenERP -> Jobmixer.com
2 php symfony openerp:sync <Anwendungsname> down # OpenERP <- Jobmixer.com
```

Listing 5.22: Aufruf des `SynchronisationTasks`

5.9.2 Der SyncInvoiceTask

Dieser Task ist speziell für die Synchronisation von Rechnungsdaten gedacht. Er erzeugt selbstständig die Einträge für die Rechnungssynchronisation und startet diese anschließend. Nachdem die Synchronisation abgeschlossen ist, werden die erzeugten Einträge für die Synchronisation wieder entfernt.

```
1 php symfony openerp:invoice <Anwendungsname>
```

Listing 5.23: Aufruf des `SyncInvoiceTasks`

5.9.3 Der SyncProductTask

Mit dem `SyncProductTask` verhält es sich mit dem `SyncInvoiceTask` (siehe Abschnitt 5.9.2). Auch dieser Task erzeugt eigenständig die Einträge für die Synchronisation, führt diese aus und löscht die Einträge anschließend wieder.

```
1 php symfony openerp:product <Anwendungsname>
```

Listing 5.24: Aufruf des `SyncProductTasks`

5.9.4 Der SyncCustomerTask

Dieser Task ist für die Initialisierung zu Beginn der Synchronisation gedacht. Er erfasst die Geschäftskunden von *Jobmixer.com* und erzeugt für jeden Geschäftskunden einen Eintrag für die Synchronisation. Diese Einträge können danach mit dem `SynchronisationTask` (siehe Abschnitt 5.9.1) mit *OpenERP* synchronisiert werden.

```
1 php symfony openerp:customer <Anwendungsname>
```

Listing 5.25: Aufruf des `SyncCustomerTasks`

[48] Aufruf erfolgt durch `registerEmpty()`-Methode (siehe Abschnitt 5.4)

5.10 Absicherung der Kommunikation

Für die Absicherung der Kommunikation zwischen *OpenERP* und *Jobmixer.com* soll (wie in Abschnitt 4.11 erwähnt wurde) ein SSH-Tunnel erzeugt werden. Dazu muss auf dem Server von *Jobmixer.com* der in Listing 5.26 gezeigte Befehl aufgerufen werden. Dieser baut eine geschützte Verbindung zum internen *IN AUDITO GmbH*-Netzwerk auf. Wenn dies geschehen ist, kann der Datenaustausch zwischen *Jobmixer.com* und *OpenERP* beginnen.

```
1 ssh -L <Zielport>:              # ssh -L 1984:
2        <Ziel-IP-Adresse>:       #        192.168.1.1
3        <urspruenglicher Port>   #        8069
4        <Zielnetzwerk>           #        foo.bar.de
5     -p <Zielnetzwerkport>       #     -p 9999
6     -l <Benutzername>           #     -l frank.weigmann
```

Listing 5.26: Befehl zum Aufbau eines SSH-Tunnels zwischen dem *Jobmixer.com*-Server und dem *OpenERP*-Server

<Zielport>
: Der Port, auf dem *OpenERP* für *Jobmixer.com* erreichbar sein soll.

<Ziel-IP-Adresse>
: Die IP des virtuellen Servers innerhalb des *IN AUDITO GmbH*-Netzwerkes.

<ursprünglicher Port>
: Der Port, auf dem *OpenERP* auf dem virtuellen Server zur Verfügung steht.

<Zielnetzwerk>
: Die Adresse, unter der das Zielnetzwerk aus dem Internet erreichbar ist.

<Zielnetzwerkport>
: Der Port für das <Zielnetzwerk>.

<Benutzername>
: Der Benutzername zum Anmelden im <Zielnetzwerk>.

Kapitel 6

Test der Synchronisation

Dieses Kapitel der Diplomarbeit beschäftigt sich mit dem Testen der Synchronisation. Dazu wird auf die Testumgebung genauer eingegangen und anhand von zwei Beispielen der Ablauf der Unit Tests veranschaulicht. Zu Beginn werden jedoch noch zwei Fragen geklärt.

Die erste Frage betrifft die Wichtigkeit der Unit Tests für die Anwendung. Das hat zwei Gründe: zum einen helfen Unit Tests schon während der Entwicklung, Fehler zu vermeiden.. Das bedeutet, es wird für jede Methode ein Test entwickelt, bevor diese überhaupt geschrieben wird. Anhand dieser Tests ist es möglich, festzustellen ob eine Methode fehlerfrei funktioniert. Dieses Vorgehen wird als testgetriebene Entwicklung bezeichnet. Und zum Anderen können Änderungen am bestehenden Quellcode schnell und sicher auf ihre Kompatibilität überprüfen werden. Auf diese Weise können Fehler bei der Weiterentwicklung der Anwendung vermieden werden.

Die zweite Frage soll klären, welche Teile der Anwendung getestet werden sollen. Anzustreben ist dabei eine möglichst vollständige Abdeckung der Anwendung durch Tests. Dies soll für `iaOpenErpPlugin` dadurch erreicht werden, dass die einzelnen Methoden der beteiligten Klassen getestet werden. Um die Tests zu komplettieren, werden die einzelnen Abläufe für die Synchronisation der Bestelldaten, Rechnungsdaten, Kundendaten und Produktdaten getestet. So kann festgestellt werden, ob die Synchronisationsmethoden für die einzelnen *OpenERP*-Objekte (vgl. 4.7.3) ordnungsgemäß funktionieren.

6.1 Die Testumgebung

Da das Testen einer Anwendung nicht in der Produktionsumgebung geschehen kann und auch keine echten Kundendaten zum Einsatz kommen sollen, ist eine spezielle Testumgebung notwendig. Diese wird für *Jobmixer.com* durch das in *symfony* integrierte `lime`-Testframework bereitgestellt, welches die Anwendung auf die Testumgebung umstellt und auf eine separate Datenbank mit Testdaten zugreift. Das Ausführen der Tests wird von einem lokalen System durchgeführt.

Die Testumgebung, die für *OpenERP* geschaffen wurde, besteht aus einem virtuellen Server, auf dem nur das Testsystem läuft. Dieses besteht aus einer *OpenERP*-Installation, wie sie auf dem Produktivsystem genutzt wird, und einer separaten Testdatenbank. Damit jeder Entwickler von seinem lokalen System aus manuelle Tests durchführen kann, wird für jeden eine separate *OpenERP*-Datenbank angelegt, auf die nur der entsprechende Entwickler Zugriff hat. Auf diese Weise werden Überschneidungen bei der Entwicklung (z. B. doppelte Benutzernamen) in *OpenERP* vermieden.

Die Unit Tests dienen zwar ursprünglich nicht dem Testen von Abläufen, da aber bei der Synchronisation nur Methoden aufgerufen werden, die sich durch Unit Tests überprüfen lassen, können die Abläufe durch das aufeinanderfolgende Testen der einzelnen verwendeten Methoden überprüft werden.

6.2 Die Testdaten

Die Testdaten für die Anwendung werden in einer *symfony*-Anwendung in in YAML-Syntax verfassten Dateien abgespeichert. Diese werden als `fixtures` bezeichnet. Zu Beginn eines Testdurchlaufes werden die Daten dann in die Testdatenbank geschrieben und es kann während des Tests auf sie zugegriffen werden.

Die Testdaten, die in *OpenERP* benötigt werden, z. B. Produkte, müssen von Hand eingetragen werden. Da diese Daten nur dem Abrufen dienen und nicht verändert werden, ist dies ein einmaliger Vorgang.

[48] Zu diesem Zweck stellt das *symfony*-Functional Tests zur Verfügung. Diese dienen dazu, das Aufrufen und Durchklicken einer Webseite zu simulieren (vgl. [12]).

6.3 Beispiele

Dieser Abschnitt zeigt zwei Beispiele für Unit Tests mithilfe von *symfony*. Bevor auf die Tests genauer eingegangen wird, sind noch die verwendeten Testbefehle zu erklären. Diese werden in Tabelle 6.1 einzeln dargestellt und erläutert.[49]

Tabelle 6.1: Übersicht über die am häufigsten verwendeten Methoden zum Testen der Anwendung

Methode	Beschreibung
`diag($msg)`	Dient zur Ausgabe eines Textes während der Tests
`ok($test, $msg)`	Überprüft, ob der Wert in `$test` dem boolschen Wert `TRUE` entspricht.
`is($ausdruck1, $ausdruck2, $msg)`	Überprüft, ob `$wert1` und `$wert2` übereinstimmen.
`isnt($ausdruck1, $ausdruck2, $msg)`	Überprüft, ob `$wert1` und `$wert2` nicht übereinstimmen.
`isa_ok($object, $class, $msg)`	Überprüft, ob `$object` vom Typ `$class` ist.
`isa_ok($variable, $type, $msg)`	Überprüft, ob `$variable` den Typ `$type` besitzt.
`is_deeply($array1, $array2, $msg)`	Überprüft, ob die Werte von `$array1` und `$array2` übereinstimmen.

6.3.1 Testen der Produktsynchronisation

Der erste Test beschäftigt sich mit dem Teilablauf für die Produktsynchronisation (Listing 6.1). Dazu wird zuerst eine neue Instanz der Klasse `lime_test` mit der Anzahl, in diesem Fall 8, der geplanten Tests erzeugt. Damit keine alten Daten den Test stören, werden diese zu Beginn gelöscht. Wenn dies geschehen ist, werden die Produkte und die Produktkategorien zur Synchronisation angemeldet und die Tabelle `Changes` darauf überprüft, ob die beiden Einträge vorhanden sind. Bevor die Synchronisation der Produktdaten gestartet wird, werden `Product` und `ProductCategory` noch dahingehend überprüft, ob sie leer sind. Nach der Synchronisation werden `Product` und `ProductCategory` ein zweites Mal überprüft, diesmal dahingehend, ob sie nicht mehr leer sind. Danach wird anhand eines Testproduktes überprüft, ob der Inhalt richtig übertragen wurde.

[49] Eine komplette Übersicht über alle zur Verfügung stehenden Methoden zum Testen der Anwendung befindet sich in [7, Seite 313, Tabelle 18.1].

```
include(dirname(__FILE__).'/../../bootstrap/unit.php');
$t = new lime_test(8, new lime_output_color());

Doctrine::getTable('Changes')->findAll()->delete();
Doctrine::getTable('IdentificationMap')->findAll()->delete();
Doctrine::getTable('ObjectIdentification')->findAll()->delete();
$products = Doctrine::getTable('Product')->findAll()->delete();
$productCategories = Doctrine::getTable('ProductCategory')->findAll()->delete();
Changes::registerEmpty('Product');
$productChange = Doctrine_Query::create()->
                                from('Changes c, c.ChangesObject co')->
                                where('co.jobmixer_object = "Product"')->
                                fetchOne();
$t->isa_ok($productChange, 'Changes',
           'Produkte stehen sind fuer Synchronisation angemeldet');
Changes::registerEmpty('ProductCategory');
$productCategoryChange = Doctrine_Query::create()->
                                  from('Changes c, c.ChangesObject co')->
                                  where('co.jobmixer_object = "ProductCategory"')->
                                  fetchOne();
$t->isa_ok($productCategoryChange, 'Changes',
           'Produktekategorien stehen sind fuer Synchronisation angemeldet');

$t->is($products->count(), 0, 'Es sind noch keine Produkte vorhanden');
$t->is($productCategories->count(), 0,
       'Es sind noch keine Produktekategorien vorhanden');
OpenErpSynchronization::getInstance()->startUp();
$t->info('Synchronisierung von Testprodukten');
$t->diag('  berprfung   Produktkategorien');
$productCategories = Doctrine::getTable('ProductCategory')->findAll();
$t->isnt($productCategories->count(), 0,
         'Es sind jetzt Produktekategorien vorhanden');
$companyProfileCategory = Doctrine::getTable('ProductCategory')->
                          findOneByName('Firmenprofil');
$t->isa_ok($companyProfileCategory, 'ProductCategory',
           'Produktkategorie "Firmenprofil" erfolgreich synchronisiert.');
$t->diag('Ueberpruefung Firmenprofil');
$products = Doctrine::getTable('Product')->findAll();
$t->isnt($products->count(), 0, 'Es sind jetzt Produkte vorhanden');
$product = Doctrine::getTable('Product')->findOneByShortTerm('FP');
$t->is($product->getName(), 'Firmenprofil',
       'Testprodukt Firmenprofil synchronisiert');
$t->is($product->getDescription(), 'Dies ist ein Firmenprofil',
       'Firmenprofilbeschreibung ist korrekt');
$t->is($product->getPrice(), '59', 'Preis ist korrekt');
$t->is($product->getCategory()->getName(), 'Firmenprofil',
       'Firmenprofil Kategorie ist korrekt');
$t->info('Test abgeschlossen');
```

Listing 6.1: Tests für die Synchronisation der Produktdaten

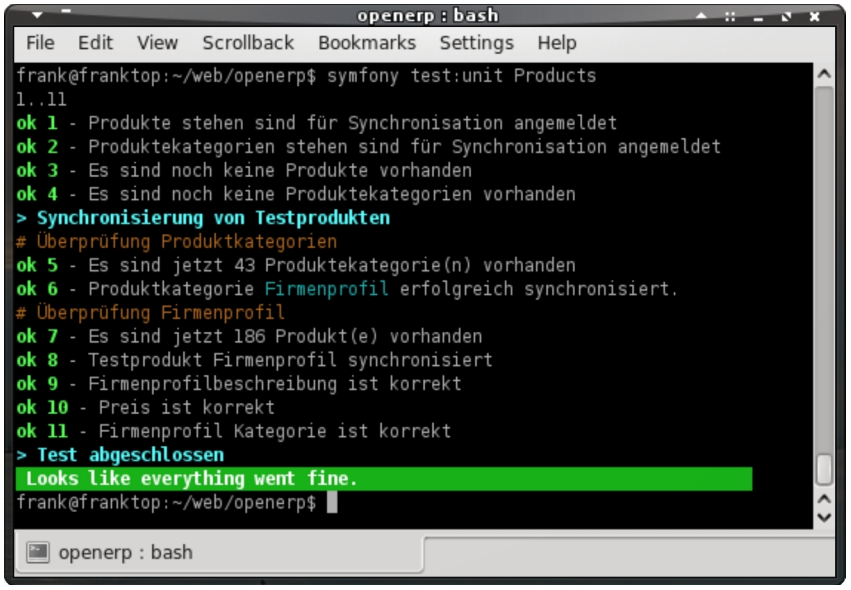

Abbildung 6.1: Erfolgreich ausgeführte Produktsynchronisationstests auf der Konsole

6.3.2 Testen der `PluginFieldMapTable`-Klasse

Beim zweiten Beispieltest (Listing 6.2) handelt es sich um die Überprüfung der zwei Methoden der `PluginFieldMapTable`-Klasse. Wie schon beim vorhergehenden Test wird zuerst eine Instanz der Klasse `lime_test` erzeugt. Anschließend werden alle Objekte erzeugt, die von der `_create`-Methode als Parameter erwartet werden. Danach wird die `_create()`-Methode aufgerufen und das Ergebnis in einer Variable abgelegt. Um festzustellen, ob die Methode korrekt funktioniert, wird der Inhalt der Variable überprüft. Ist der Inhalt ein Objekt vom Typ `FieldMap`, funktioniert die `_create()`-Methode fehlerfrei.

Die zweite zu überprüfende Methode sucht alle zu einem *OpenERP*-Objekt gehörenden Felder anhand des Objektnamens heraus. Dazu wird auf das vorher erzeugte `FieldMap`-Objekt zurückgegriffen. Da es als letztes `FieldMap`-Objekt erzeugt und gespeichert wurde, sollte er sich an letzter Stelle befinden. Ist dies der Fall, funktioniert die `getFieldsByOpenerpObjectName()`-Methode richtig. Um das zu testen, wird die sie aufgerufen und das Ergebnis in einer Variable abgelegt. Jetzt wird überprüft, ob der Inhalt der Variable die erwartete Instanz von `Doctrine_Collection` ist. Danach wird überprüft, ob der letzte Eintrag der `Doctrine_Collection` dem vorher angelegten `FieldMap`-Objekt entspricht. Zum Schluss wird das erzeugte `FieldMap`-Objekt wieder gelöscht.

```php
include(dirname(__FILE__).'/../../bootstrap/unit.php');

$t = new lime_test(2, new lime_output_color());

$objectMap = Doctrine::getTable('ObjectMap')->findOneByJobmixerObject('Product');
$fieldMap = Doctrine::getTable('FieldMap')->
            _create('testfeld', 'testfeld', $objectMap, 'string');
$t->diag('Teste _create() Methode');
$t->isa_ok($fieldMap, 'FieldMap', 'Objekt vom Typ FieldMap erzeugt');
$fieldMap->save();

$t->diag('Teste getFieldsByOpenerpObjectName()');
$fields = Doctrine::getTable('FieldMap')->
          getFieldsByOpenerpObjectName('product.product');
$t->isa_ok($fields, 'Doctrine_Collection',
     'Es wird eine Doctrine_Collection zurückgegeben');
$t->is($fields->getLast()->getOpenerpName(),
       'testfeld', 'Es wurden die korrekten Felder gefunden.');
$fieldMap->delete();
```

Listing 6.2: Tests für die `PluginFieldMapTable`-Klasse

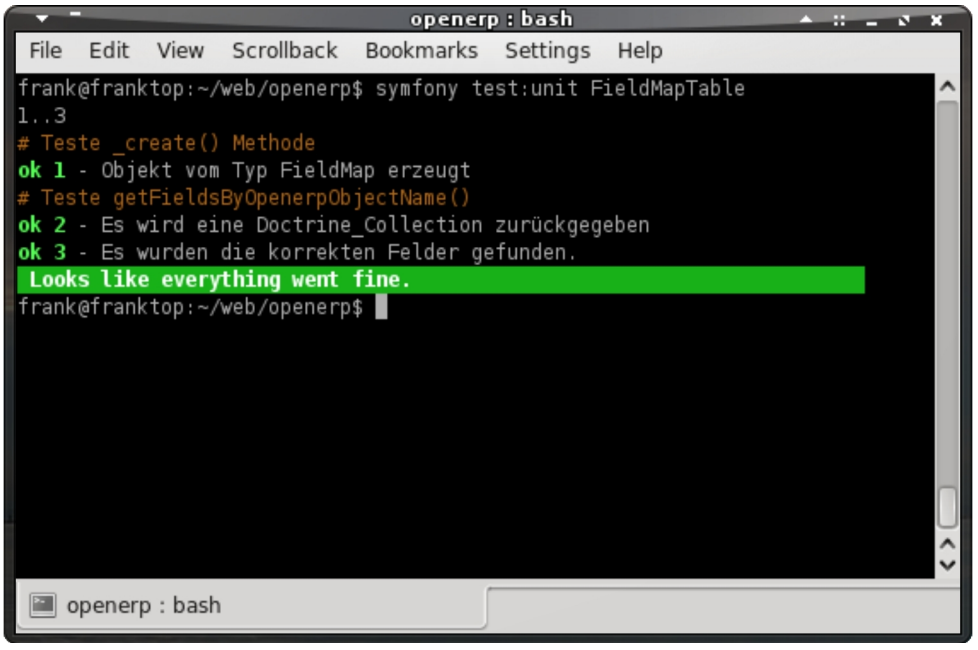

Abbildung 6.2: Erfolgreich ausgeführte Tests für die Methoden von `PluginFieldMapTable` auf der Konsole

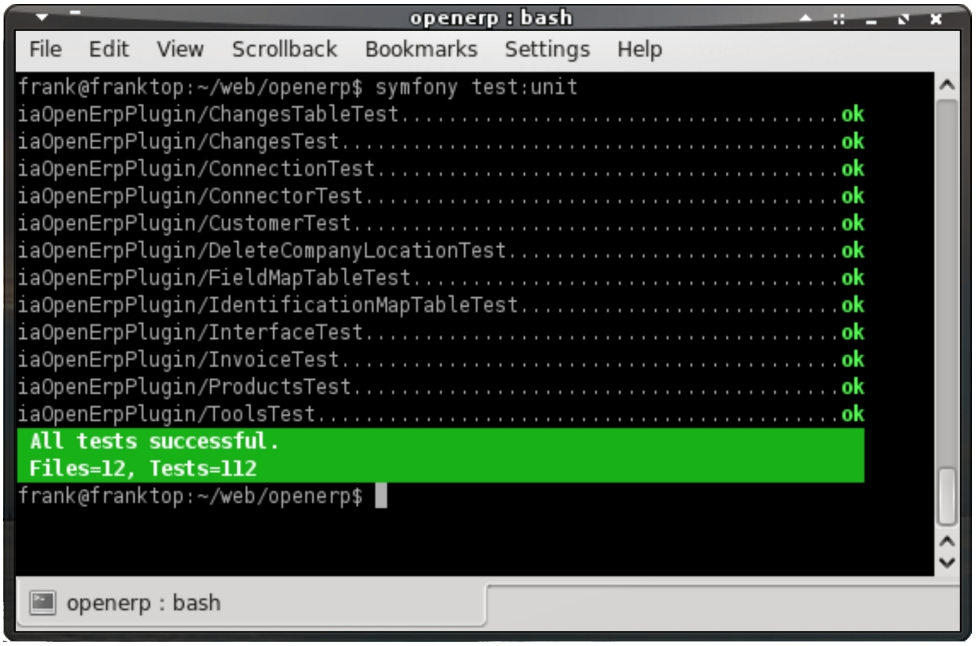

Abbildung 6.3: alle Tests auf der Konsole erfolgreich ausgeführt

Kapitel 7

Fazit und Ausblick

Ziel der Diplomarbeit war es, einen Kommunikationsweg für den Datenaustausch zwischen *Jobmixer.com* und *OpenERP* zu entwerfen und umzusetzen.

Alle an die Synchronisation gestellten Anforderungen konnten erfüllt werden. Die Unterschiede zwischen den Datenstrukturen von *Jobmixer.com* und *OpenERP* wurden durch das Mapping der Objekte und deren Felder ausgeglichen (vgl. Abschnitt 4.6.2). Die Konfiguration des Mappings erfolgte in einer Datei mit YAML-Syntax (vgl. Abschnitt 2.3). Auf diese Weise konnten alle erforderlichen Daten (wie z. B. Produktdaten) angepasst und synchronisiert werden. Um die Funktionalität des `iaOpenErpPlugin` zu gewährleisten, stehen Tests zur Verfügung (vgl. Kapitel "Test der Synchronisation„).

Besonders unkompliziert gestaltete sich das Benutzen der *XML-RPC*-Schnittstelle mithilfe der Bibliothek *XML-RPC for PHP*, da diese alle erforderlichen Zugriffsmethoden implementiert hat (vgl. Abschnitt 5.3.1). Weiterhin wurde durch *Doctrine* ein einfacher und schneller Zugriff auf die Datenbanktabellen von *Jobmixer.com* und die des Plugins sichergestellt. Die Pluginschnittstelle (vgl. Abschnitt 2.3), die Unterstützung mit Tasks (vgl. Abschnitt 2.3) und die integrierte Testumgebung (vgl. Abschnitt 2.3) von *symfony* machten die Integration der Synchronisation in *Jobmixer.com* sehr komfortabel. Da der Großteil der hier aufgezählten Technologien und Techniken schon vor Beginn der Synchronisation von *Jobmixer.com* und *OpenERP* eingesetzt wurde, mussten diese nicht im Rahmen dieser Arbeit implementiert, sondern konnten von Anfang an genutzt werden.

Als einziger negativer Punkt ist die unvollständige Dokumentation von *OpenERP* zu nennen, die es notwendig machte, sich in den Quellcode von *OpenERP* einzuarbeiten.

Über diese Diplomarbeit hinaus kann die Synchronisation von *Jobmixer.com* und *OpenERP* um neue Funktionen erweitert und so an neue Geschäftsprozesse angepasst werden. Zur Überwachung der Synchronisation kann beispielsweise eine Möglichkeit zum Monitoring implementiert werden. Für die Kontrolle der Synchronisation können Werkzeuge entwickelt werden, welche z. B. diejenigen Datensätze überprüfen, die zur Synchronisation angemeldet wurden, und so sicherstellen, dass diese korrekt sind. Die Umsetzung neuer Geschäftsprozesse ist durch die Struktur von `iaOpenErpPlugin` bereits vorbereitet (vgl. Abschnitt 4.7).

Abschließend ist festzuhalten, dass die Kommunikation zwischen dem ERP *OpenERP* und der Webanwendung *Jobmixer.com* zur Unterstützung des Verkaufsprozesses erfolgreich umgesetzt werden konnte und der Austausch von Bestelldaten, Kundendaten, Produktdaten und Rechnungsdaten somit erfolgreich eingeführt.

Glossar

Active Record

Beim Active Record Entwurfsmuster wird ein Datensatz aus einer Tabelle oder einem View als Objekt dargestellt. Dieses Objekt enthält die Daten und die Anweisungen, wie diese verarbeitet werden sollen.

AJAX

AJAX bietet die Möglichkeit, Teile einer Webseite zu aktualisieren, ohne die komplette Seite neu zu laden. Dazu wird eine vom Benutzer auf der Webseite ausgelöste Anfrage durch den Server bearbeitet und zum Browser zurückgesendet. Ist dies geschehen, wird die Webseite an der entsprechenden Stelle durch Javascript aktualisiert.

CRUD

Create Retrieve Update and Delete. Die vier Grundoperationen einer Webanwendung [7, S. 20].

Data Mapper

Das Data Mapper Entwurfsmuster beschreibt wie Objekte und Datenbanken unabhängig voneinander Daten austauschen können. Die Data Mapper Schicht ist verantwortlich für den Datentransfer und das Objekt und Datenbank voneinander isoliert werden.

Datenbankabstraktionsschicht

Die Datenbankabstraktionsschicht (Database Abstraction Layer) ist eine Programmschnittstelle zur Vereinheitlichung der Verbindung zwischen Anwendung und Datenbank. Auf diese Weise kann die Datenbank gewechselt werden, ohne den Programmcode anzupassen.

Enterprise Ressource Planning

Enterprise-Resource-Planning sind Softwarelösungen für die Steuerung von betrieblichen Geschäftsprozessen. Die Einsatzbereiche reichen von der Organisation

bis hin zu der Verwaltung und der Kontrolle eines Unternehmens. Dazu gehören Finanzwesen, Warenwirtschaft, Lagerhaltung, Produktionsplanung, Disposition, Buchhaltung, Vertrieb und Personalwesen [10].

Entity-Relationship-Model

Das Entity-Relationship-Model dient zum Entwurf von relationalen Datenbanken ist dabei aber unabhänig von der Datenbank. Mit dem Entity-Relationship-Model werden Zusammenhänge von Ereignissen, Handlungen, Objekten und Anwendungen modelliert und in Relation zueinander gesetzt.

Framework

Ein Framework bestimmt die Software-Architektur einer Anwendung und bildet die generelle Rahmenstruktur. Dazu stellt es abstrakte und konkrete Klassen für die Designstruktur zur Verfügung.

Metadata Mapping

Das Metadata Mapping Entwurfsmuster hilft dabei, das Mapping in einer einfachen Tabellenform abzuspeichern.

Model-View-Control

Model-View-Control ist ein Set von Entwurfsmustern um die einzelnen Schichten einer Anwendung voneinander zu trennen. Im View werden die Daten des Models dargestellt. Das Model speichert die Daten der Anwendung, der Controller nimmt die Aktionen des Benutzers entgegen und lässt Model und View daraufhin ihren Status ändern.

Object-Relational Mapper

Object-Relational Mapper wandeln Daten einer relationalen Datenbank beim Lesen in Objekte um, beim Schreiben werden die Objekte dann wieder zurück gewandelt.

PEAR

PEAR ist ein System zur Verteilung von wiederverwendbaren PHP-Komponenten.

PHP-Data-Objects

PHP-Data-Objects sind eine konsistente Schnittstelle, um auf die Daten in der Datenbank zuzugreifen. Dafür stellen PHP-Datenobjekte eine Abstraktionsschicht für den Datenzugriff bereit, mit der alle Funktionen einer Datenbank unabhängig von der Datenbank genutzt werden können. Dieser Vorgang wird durch einen datenbankspezifischen PDO-Treiber realisiert, der für die aktuell genutzte Datenbank vorhanden sein muss.

PHP: Hypertext Preprocessor

PHP ist die Abkürzung für 'PHP: Hypertext Preprocessor' eine weitverbreitete Open-Source Skriptsprache speziell für Webentwicklungen. PHP lässt sich in HTML einbinden. Die Syntax erinnert an C, Java und Perl und ist einfach zu erlernen. Das Hauptziel dieser Sprache ist es, Webentwicklern die Möglichkeit zu geben, schnell dynamisch generierte Webseiten zu erzeugen.

Scaffolding

Als Scaffolding wird das automatische Erstellen eines Programmgerüstes mithilfe von CRUD bezeichnet.

Singleton Entwurfsmuster

Das Singleton-Pattern sichert ab, dass von *einer* Klasse nur eine Instanz existieren kann, und stellt einen globalen Zugriffspunkt auf diese Instanz bereit.

SOAP

SOAP ist ein Protokoll mit dem Status einer W3C Empfehlung. Es dient dem Austausch XML-basierter Nachrichten über ein Netzwerk und zum Aufruf von RPCs. Zudem regelt es das Nachrichtendesign und stellt ein Rahmenwerk zur Verfügung, das es erlaubt, applikationsspezifische Informationen zu übertragen.

SSH

SSH ist ein Sicherungsprotokoll, das die Authentifizierung und die Kommunikation kryptografisch absichert. Für die Authentifizierung ist es möglich, diese über eine Passwortüberprüfung extra abzusichern. Für SSH existieren zwei zueinander inkompatible Varianten, SSH1.x und SSH2.x.

Subversion

Subversion ist ein Open-Source Versionskontrollsystem. Es verwaltet Änderungen, die an Dateien oder Verzeichnissen vorgenommen werden. Dadurch besteht die Möglichkeit, alte Versionen einer Datei wiederherzustellen oder ihre Geschichte zu verfolgen.

Literaturverzeichnis

[1] ACHOUR, Mehdi ; BETZ, Friedhelm ; DOVGAL, Antony u. a.: *PDO - Einführung*. http://de.php.net/manual/de/intro.pdo.php

[2] CLAUS, V. ; SCHWILL, A.: *Duden, Informatik AZ, Fachlexikon für Studium, Ausbildung und Beruf, Dudenverlag, Mannheim Leipzig Wien Zürich, 4.* 2006

[3] COLLINS-SUSSMAN, B. ; FITZPATRICK, B.W. ; PILATO, C.M. ; LICHTENBERG, K.: *Versionskontrolle mit Subversion*. O'Reilly Germany, 2006 http://svnbook.red-bean.com/nightly/de/svn-book.html

[4] FOWLER, Martin: *www.martinfowler.com - Active Record Entwurfsmuster*. http://www.martinfowler.com/eaaCatalog/activeRecord.html

[5] FOWLER, Martin: *www.martinfowler.com - Data Mapper Entwurfsmuster*. http://www.martinfowler.com/eaaCatalog/dataMapper.html

[6] FOWLER, Martin: *www.martinfowler.com - Metadata Mapping Entwurfsmuster*. http://www.martinfowler.com/eaaCatalog/metadataMapping.html

[7] HABEKERN, Timo: *Das Symfony Framework*. Entwickler.press, 2008. – ISBN 978–3–939084–14–3

[8] HENNEBRÜDER, Sebastian ; COMPUTING, Galileo (Hrsg.): *Hibernate - Das Praxisbuch für Entwickler*. 1. Auflage. Galileo Press, 2007

[9] KLAUS LIPINSKI, Dipl.-Ing. u. a. ; GMBH, DATACOM B. (Hrsg.): *itwissen.info - Definition ERM*. http://www.itwissen.info/definition/lexikon/entity-relationship-model-ERM.html

[10] KLAUS LIPINSKI, Dipl.-Ing. u. a. ; GMBH, DATACOM B. (Hrsg.): *itwissen.info - Definition ERP*. http://www.itwissen.info/definition/lexikon/enterprise-resource-planning-ERP.html

[11] KLAUS LIPINSKI, Dipl.-Ing. u. a. ; GMBH, DATACOM B. (Hrsg.): *itwissen.info - Definition Framework*. http://www.itwissen.info/definition/lexikon/Framework-framework.html

[12] POTENCIER, F. ; ZANINOTTO, F.: The Definitive Guide to symfony. In: *Berkeley, CA, USA: Apress* (2007)

[13] SAVOUREL, Yves ; REID, John ; JEWTUSHENKO, Tony ; RAYA, Rodolfo M. ; OASIS (Hrsg.): *XLIFF Version 1.2*. `http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html`. Version: Februar 2008

[14] SCHMID, E.: PHP Handbuch. In: *PHP Documentation Group* (2004). `http://www.php.net/manual/de`

[15] SCHMIDT, S.: *PHP Design Patterns*. O'Reilly, 2007

[16] SCOTT, Joseph: *Definition Datenbankabstraktionsschicht*. `http://joseph.randomnetworks.com/archives/2004/07/08/what-is-a-database-abstraction-layer/`

[17] SPRL, Tiny: *Offizielle OpenERP Webseite*. `http://www.openerp.com/`

[18] STEIGELE, Dr. H. ; CASCADEIT (Hrsg.): *www.4whatitis.ch - Definition SOAP*. `http://www.4whatitis.ch/index.php?page=883`

[19] TATUM, Malcom: *wisegeek.com - Definition Web Application Framework*. `http://www.wisegeek.com/what-is-a-web-application-framework.htm`

[20] WAGE, Jonathan H. ; BORSCHEL, Roman S. ; BLANCO, Guilherme u. a.: *Doctrine Dokumentation*. `http://www.doctrine-project.org/documentation/manual/1_0/de/introduction`

CD-Verzeichnis

OpenERP/
OpenERP-Server und Client für Windows und Linux.

Doctrine/
Der *Doctrine* ORM (Version 1.1.2).

symfony/
Das *symfony*-Framework (Version 1.1.7).

iaOpenErpPlugin/
Der Quellcode von `iaOpenErpPlugin`.

Dokumentation/
Die Dokumentation des `iaOpenErpPlugin` Quellcodes.

Quellen/
Quellen im PDF Format.

- *symfony* Handbuch für Version 1.1
- *Doctrine* Handbuch für Version 1.1

Diplomarbeit.pdf
Die Diplomarbeit als PDF Datei.

index.html
Startet eine lokale HTML Seite mit einem Auswahlmenü.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Mittweida, den 2. Juli 2009

_____

Frank Weigmann