Design and Implementation of an Eclipse P2-based Online-Repository for exchanging Cloud Profiles

Bachelor's Thesis

Simon Kund

March 30, 2013

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M. Sc. Sören Frey

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Cloud computing fastens its position in the industry as a means to deliver computational resources dynamically, offering advantages in performance and cost over traditional, inflexible computing resources. To offer a structured and well evaluatable approach of migrating existing software systems to the cloud, the CloudMIG approach introduced a scale for a given software system's suitability and aligment with a given cloud environment. CloudMIG Xpress was created as a toolsuite to assist this approach. In CloudMIG Xpress, cloud profiles are created for cloud products provided by major cloud providers, structuring their constraints as well as information about pricing, to aid automatic evaluation of their suitability for a software system. This work aims at extending CloudMIG XPress by a centralized repository through which these cloud profiles are distributed and evaluating the Eclipse Provisioning Platform (P2) as a means to distribution.

Contents

Chapter 1

Introduction

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."
-Mell and Grance [2011]

Cloud computing fastens its position in the industry as a means to deliver computational resources dynamically, offering advantages in performance and cost over traditional, inflexible computing resources. As the movement towards the cloud is relatively young, businesses, academia and others are in need of tools and guidelines that enable them to migrate their legacy applications to the cloud so they can profit from those advantages. Different cloud service providers offer different approaches toward cloud computing in each of their cloud products, incorporating various payment models and environmental restrictions. Typically, cloud providers will not change their cloud environments to suit the structure of a given user's software or services needs and the user will have to make substantial changes to existing software systems to allow a migration into the cloud obeying a given providers rules.
When migrating legacy software to the cloud it is important to focus on adjusting the software in a way that maximizes the advantages that cloud platforms offer and to choose the right platform for one's purpose. The CloudMIG Approach is a model-based approach for migration of software systems to cloud-optimized applications. It offers an index to measure how well a given application is adjusted to a certain cloud environment, the Cloud Suitability and Alignment (CSA) hierarchy (explained in greater detail in 2.3), allowing for easy comparison of given cloud environments' suitability for a specific application. To enable an easier migration, CloudMIG Xpress was created based on the CloudMIG Approach, as a toolsuite that analyses a given program's code and offers potential cloud deployment scenarios based on the information it has about providers and products. Each cloud product offered comes with certain constraints, some, for example, may have limitations on persistent storage while others may not support a subset of functions a programming language offers. We call these constraints Cloud Environment Constraints (CECs), and they can be manyfold in form and importance [Frey et al. 2012a]. The constraints in effect for a cloud product can be grouped into a so-called cloud profile, alongside cost and other information, and be employed by CloudMIG Xpress.

1.1 Motivation

CloudMIG Xpress offers the capability of locally creating and storing cloud profiles. Whilst this is a good start, it is very cumbersome for a user of CloudMIG Xpress to have to create a set of cloud profiles before she can actually start using it. Cloud providers do not provide uniform documentation, all information has to be gathered manually. We want a structured and automated approach in order to avoid redundant work in creating cloud profiles. Harnessing the power of user-generated content, we can save CloudMIG Xpress' users the redundant work of gathering information and structuring it into a profile. As there is a growing number of cloud providers and products, centralizing their profiles allows for better maintainability and enables users to keep an up-to-date collection of profiles without manually having to maintain them.

1.2 Approach

An internet-accessible repository from which CloudMIG XPress can aquire updates to its locally saved cloud profiles will be created. This repository keeps an up to date collection of cloud profiles for various cloud providers and products. We want it to offer easy maintainability and to integrate well with the existing CloudMIG XPress application's software architecture. The repository is intended to be placed on an internet-accessible server and alongside it we will create a command line application to provide an interface for managing the set of cloud profiles in the repository.

1.3 Goals

This section clarifies the major goals of this work and gives an overview about the components that will be developed in order to accomplish these goals. The three major goals G1 (creating a cloud profile repository), G2 (creating an administrative commandline application) and G3 (CloudMIG Xpress users to upload cloud profiles) are described in the following Sections.

G1 - creating a Cloud Profile P2 Repository

As of now, CloudMIG Xpress allows its users to create and locally save cloud profiles. We want to extend this functionality to enable users to access a pool of predefined profiles that can be updated from a central repository and to prevent redundant work in creating profiles for a given provider. This goal will be accomplished by leveraging Eclipse RCP's built-in capability to access P2 repositories. Profiles will be hosted in a P2 repository and can be centrally maintained. Two components will be developed, as on the one hand the repository itself needs to be created, and on the other hand CloudMIG Xpress needs to be

extended to handle updating from the repository. The repository should be extendable to other user-generated artifacts specific to a cloud provider or product. Future work may introduce these features, they are, however, not in the scope of this thesis, we only mention them here to motivate an extensible implementation.

Goals - G2 - creating an administrative commandline application

Once the repository is placed in a server environment, an administrator should be able to create new profiles and delete or update existing ones. As a server usually does not have a graphical user interface, this application must be accessible from the commandline.

Goals - G3 - CloudMIG Xpress users to upload cloud profiles

We can extend the scenario given in goal G1 to harness community work in generating profiles. By allowing users to upload their locally created cloud profiles back to the repository it is likely that the collection of profiles held in the repository will experience more growth while needing less maintenance from the repository's administrators, compared to having a solely administrator-maintained repository. While the repository's administrator will still be responsible for deciding which user-generated profiles should be included, he no longer needs to create the profiles by himself or to manually include them in the repository. Instead, we are going to create a server-application that provides a network service to which the community can upload profiles and which allows administrators to review and select user-generated profiles for publishing. Due to unforeseen complications during development, this goal has not made it past the requirements engineering stage and will not be mentioned afterwards.

1.4 Document Structure

This section describes the structure of this thesis. By now Chapter 1 should have introduced the reader to the matter. Next we will discuss the foundations of this work and relevant software in Chapter 2. Chapter 3 is dedicated to the requirements elicitation, analyzing use cases and activities. In Chapter 4 the design will be covered, giving an overview of components to be developed and design decisions concerning those. The implementation will be detailed in Chapter 5. An evaluation of the resulting software is found in Chapter 6, in Chapter 7 related work is discussed and finally in Chapter 8 a conclusion is drawn.

Chapter 2

Foundations and Technologies

2.1 Software Modernization

Software modernization refers to the practice of prolonging a software's lifecycle by converting, rewriting or porting it's code to a modern programming language, runtime environment or hardware platform. It is a discipline of software engineering and its strategies are eventually needed when porting software to a different hardware platform, such as the cloud. This process of porting is often also includes retargeting [Kowalczyk and Kwiecinska 2009] as the new platform introduces a new target architecture for the legacy application to run on.

2.2 Cloud Computing

Cloud computing, as defined by [Mell and Grance 2011], is gaining importance in todays computing landscape. As described in chapter 1, it promises to enable savings, financial and economical, and better utilization of resources.
[Armbrust et al. 2009] describe the new aspects in cloud computing from a hardware point of view as follows:

- *The illusion of infinite resources available on demand*, thereby eliminating the need for cloud computing users to plan far ahead for provisioning.
- *The elimination of an up-front commitment by cloud users*, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs.
- *The ability to pay for use of computing resources on a short term basis as needed* (e.g., processors by the hour and storage by the day) and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

This given, there is a strong incentive given to migrate software systems to the cloud.
There are three service models of cloud computing, according to [Mell and Grance 2011]:

- IaaS - Infrastructure as a Service, providing the capability to provision processing, storage, networks, and other fundamental computing resources

- PaaS - Platform as a Service, providing the capability to deploy consumer created applications using programming languages, libraries, services, and tools supported by the provider
- SaaS - Software as a Service, providing the capability to use applications running on a cloud infrastructure, typically accessible from various client devices (e.g., web-based email)

This thesis is focused on IaaS and PaaS providers and products whenever referring to cloud providers or cloud products.

2.3 Cloud Migration and the CloudMIG Approach

Software Migration can be described as the process of moving software systems to a different operating environment. In this thesis we will focus on the migration of SaaS-type software to IaaS and PaaS cloud environments. When migrating software to the cloud, it is important that the refactoring taking place should optimize the application in such manner that the advantages the chosen cloud platform offers are optimally utilized. Software Migration can be automated to a substantial degree. The CloudMIG Approach [Frey and Hasselbring 2011] is a model-based approach and is the foundation of CloudMIG Xpress. It aims to overcome the shortcomings of previous approaches, for example being limited to a specific cloud platform, and to unify and structure information about cloud platforms so they can be semi-automatically evaluated for a given software system. Cloud environment constraints are defined for each cloud environment and assigned a severity which indicates how critical a violation of this constraint is. The highest criticality is called *breaking*. To facilitate comparing and evaluating cloud environments as migration targets for a software the CloudMIG Approach introduces the cloud suitability and alignment (CSA) hierarchy, which ranks the level of software system's alignment with a cloud environment on a five level hierarchy as follows:

- **L0 Cloud incompatible:** At least one CEC violation with severity *Breaking* exists.
- **L1 Cloud compatible:** No CEC violations with severity *Breaking* exist.
- **L2 Cloud ready:** No CEC violations exist.
- **L3 Cloud aligned:** The execution context, utilized cloud services, or the migrated software system itself were configured to achieve an improved resource consumption (measurable in decreased costs that are to this effect charged by the cloud provider) or scalability without pervasively modifying the software system
- **L4 Cloud optimized:** The migrated software system was pervasively modified to enable automated exploitation of the cloud's elasticity. For example, its architecture was restructured to increase the level of paralellization. An evaluation was conducted to identify system parts which would experience an overall benefit from susbtitution or supplement with offered cloud services. These substitutions and supplements were performed.

2.4 Relevant Technologies and Software

The following technologies and programs will be the foundations at the core of the components to be developed.

Eclipse Rich Client Platform

CloudMIG Express is built on Eclipse RCP.[1]
While the Eclipse platform is designed to serve as an open tools platform, it is architected so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform. Applications other than IDEs can be built using a subset of the platform. These rich applications are still based on a dynamic plug-in model, and the UI is built using the same toolkits and extension points. The layout and function of the workbench is under fine-grained control of the plug-in developer in this case.

CloudMIG Xpress

CloudMIG Xpress[2] [Frey et al. 2012b] is a GUI application that provides tool support for the CloudMIG cloud migration approach [Frey and Hasselbring 2011] [Frey et al. 2012a]. It aims to support the migration of existing software systems to IaaS and PaaS-based cloud environments, with a focus on migrating client/server enterprise systems, as those often exhibit varying user demand and therefore profit greatly from the cloud's advantages.

Eclipse P2

Eclipse P2[3], for "provisioning platform", is the engine used to install plugins and manage dependencies in Eclipse SDK and Eclipse RCP Applications.
We will deliver cloud profiles through an Eclipse P2 repository and create a server application which maintains this repository with minimum administrator action.

Extensible Markup Language (XML)

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose

[1] http://www.eclipse.org/home/categories/rcp.php
[2] http://www.cloudmig.org
[3] http://www.eclipse.org/equinox/p2/

constraints on the storage layout and logical structure.[4] In CloudMIG Xpress cloud profiles are implemented in XML.

Document Object Model (DOM)

The *Document Object Model (DOM)* specifies how XML and HTML documents are represented as objects, so that they may be used in object oriented programs. It is an application programming interface (API) for valid HTML and well-formed XML documents [Hors et al. 2004].

[4] http://www.w3.org/TR/xml/

Chapter 3

Requirements

When eliciting requirements for our approach, some requirements are imposed by the software environment and tool configuration present when starting with this work as described in Chapter 2. More requirements are derived from those present and our goals defined in Section 1.3 which the resulting software should enable. At first, we will define roles for the entities involved in the use cases we want to cover, then specify those use cases, and finally the full requirements are specified.

3.1 Roles

In this section we will define the roles participating in the use cases described in the next section. Roles in general allow for grouping of entities involved in activities and use cases, and offer a layer of abstraction when describing their interactions. Roles are especially helpful to clarify responsibilities and possible actions without having to define those for each entity involved (e.g. when Alice and Bob both have the same position at a company their tasks are defined by their job description and needn't be defined individually). Like humans, software or machines can be entities that can fulfill certain roles. There are two main roles, *librarian* and *CloudMIG Xpress user*, and one less extensive role, *Client Software*, which we will define now.

CloudMIG Xpress User

Anyone running CloudMIG Xpress on his local computer can be called a *CloudMIG Xpress user*. This role has the privileges to locally create cloud profiles. Members of this role may submit these profiles for inclusion in the online repository, but the *repository administrator* will finally decide whether these profiles are published.

Librarian

The *Librarian* is responsible for reviewing and publishing of cloud profiles submitted by *CloudMIG Xpress users*. She decides wether a user submitted profile should be included in the repository.

Client Software

We will use the role *Client Software* whenever we describe actions originating from an instance of CloudMIG Xpress that do not require human interaction, for example when CloudMIG Xpress updates its local cloud profiles on startup.

3.2 Use Cases

In this section we will detail relevant use cases. There are three use cases to consider, UC1: Update of local cloud Profiles, UC2: Managing the repository, and UC3: Submitting a cloud profile to the repository.

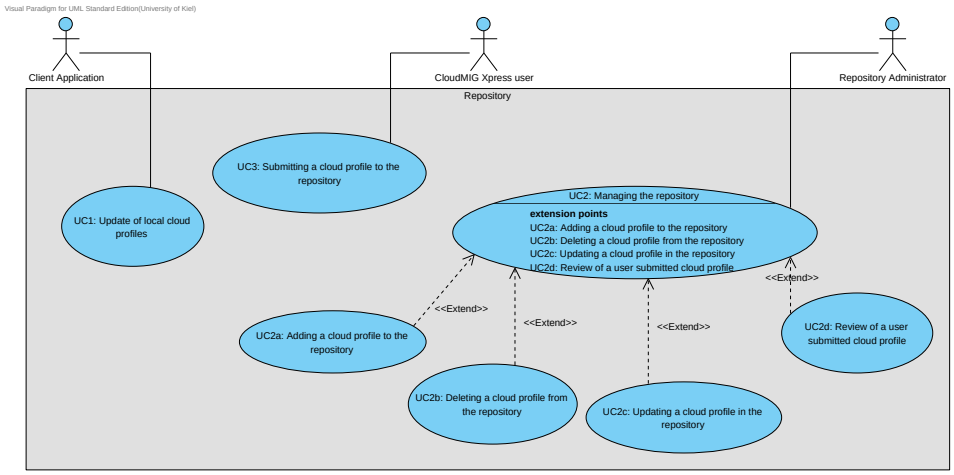

Figure 3.1. Relevant Use Cases

UC1: Update of local cloud profiles

Name	Update of local cloud profiles
Description	A client connects to the repository to check for updates to the repository's cloud profiles and updates its local collection if there is an update available.
Actors	*Client Software*
Preconditions	none
Invariants	none
Postconditions	The client has an up to date set of cloud profiles
Normal flow	1. Check for updates 2. Resume normal operation when no updates are found
Alternative flow	1. Check for updates 2. Install updates if available 3. Restart CloudMIG Xpress

UC2: Managing the Repository

Name	Managing the Repository
Description	A librarian reviews the submitted cloud profiles. She then chooses to publish or discard individual profiles.
Actors	*Librarian*
Includes	adding a cloud profile, deleting a cloud profile, updating a cloud profile
Preconditions	-
Invariants	-
Postconditions	The profiles that the librarian published are included in the repository, others may have been deleted or updated.
Normal flow	1. The librarian adds, deletes or updates cloud profiles
Alternative flow	-

UC3: Submitting cloud profiles to the repository

Name	Submitting cloud profiles to the repository
Description	A *CloudMIG Xpress user* has locally modified or created a cloud profile and submits it back to the repository
Actors	*CloudMIG Xpress user*
Includes	-
Preconditions	A Network Connection is available
Invariants	-
Postconditions	The cloud profiles the user submitted are stored on the server computer and marked as pending review.
Normal flow	1. The user selects profiles for submission and submits those 2. CloudMIG Xpress contacts the server and delivers the data over the network 3. The server application then stores the submitted profiles marked as pending review.
Alternative flow	-

3.3 Activities

In this section the activities connected to the use cases presented previously are presented with a focus on the time flow of actions and actor's responsibilities. At first we will have a look at the updating of local cloud profiles and secondly present the activity flow of submission to the repository and review of cloud profiles.

Update of local cloud profiles in CloudMIG Xpress

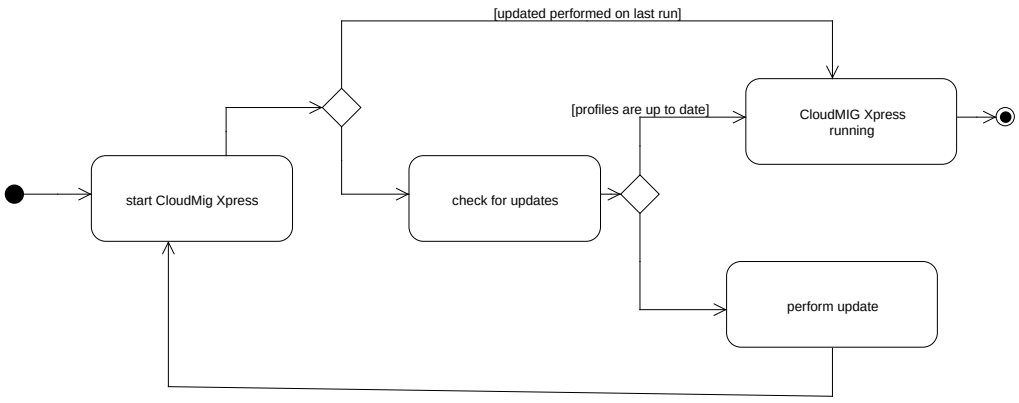

Figure 3.2. Update of local cloud profiles in CloudMIG Xpress

Figure 3.2 shows the desired flow of activities for updates from the repository. When CloudMIG Xpress starts up, it should check whether the locally saved profiles have just been updated. In case they have not just been updated, the repository is contacted and queried for new versions available. If there are no updates to be made, CloudMIG Xpress continues normal startup and transitions into running state, else P2 will handle the update and restart CloudMIG Xpress.

Submission and review of a cloud profile

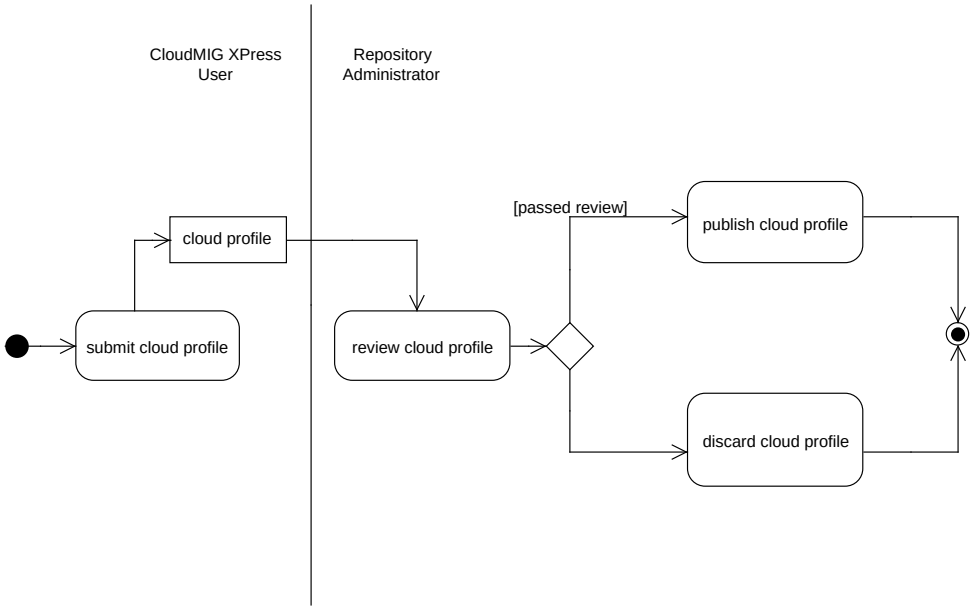

Figure 3.3. Submission and review of a cloud profile and seperation of responsibilities

Figure 3.3 shows submission and review of a cloud profile. After a cloud profile is submitted by a *CloudMIG XPress user* (this refers to new profiles initially submitted, as well as updates to existing profiles), a *librarian* reviews the profile. If he finds it qualifies for publishing, it is published (and in case of an update replaces the previous version) and subsequently available for download by clients. If the administrator finds the profile should not be published, she can decide to discard it or to leave it pending review. A published profiles' state can be reverted to pending review or be completely discarded (e.g. when the provider for this profile discontinues service and the information is no longer of any use).

3.4 Requirements specification

From our goals and use cases we can derive the following requirements, broken down for each component:

3.4.1 P2 Repository for cloud profiles

The first component whose requirements we will detail is the repository itself. This is the component responsible for storing, versioning and serving the cloud profiles to clients.

P2 repository non-functional requirements

- implemented as a P2 repository
- availability is important
- basic security should prevent unauthorized changes to the profiles

P2 repository functional requirements

- stores cloud profiles
- supports versioning for files in the repository
- allows clients to connect through WAN

3.4.2 P2 client

The second component is the client. It manages the collection of cloud profiles in CloudMIG Xpress, contacts the repository to check for available updates and applies them.

Client non-functional Requirements

- implemented as a P2 client
- stability and modularity desired

Client functional Requirements

- manages local cloud profiles (versioning and storage)
- can contact the P2 repository and compare local and remote versions to determine if updates are available
- executes updates from the repository
- informs users about updates progress

3.4.3 Administrative application

The administrative application allows the librarian to manipulate the cloud profile repository without technical knowledge about it.

Administrative application non-functional Requirements

▻ should be accessible only by librarians

Administrative application functional Requirements

▻ allows for adding, updating and deleting of proviles

▻ offers commandline accessibility

3.4.4 Server Application

Lastly, the server application component manages submission of cloud profiles to the repository. Clients connect to it via WAN access, it stores the data they deliver, so that the administrative application can be used to publish them.

Server Application non-functional Requirements

▻ availability and stability desired

Server Application functional Requirements

▻ listens on the network, allowing clients to connect and submit cloud profiles

Chapter 4

Architecture

This chapter details design decisions made. The resulting design serves as a guideline and foundation for the implementation described in Chapter 5. At first the neccessary terminology is introduced, then an overview of components is given, explaining the modularization we want to employ to achieve separation of concerns, then the design of those individual components is explained and finally the components' deployment is described.

4.1 Terminology

This section introduces the terminology used throughout the rest of this chapter. First, terms related to Eclipse RCP and then terms related to P2 are disambiguated.

Eclipse RCP Terminology

- **Product** - The purpose of a product is to define application-specific branding on top of a configuration of Eclipse plug-ins. Minimally, a product defines the ID of the application it is associated with and provides a name, description, and unique ID of its own.[1]

- **Plugin** - On disk, an Eclipse based product is structured as a collection of *plugins*. Each *plugin* contains the code that provides some of the product's functionality. The code and other files for a plug-in are installed on the local computer, and get activated automatically as required.[2]

- **Feature** - A product's plugins are grouped together into features. A feature is a unit of separately downloadable and installable functionality. Features themselves normally do not include any code.[3]

[1] http://wiki.eclipse.org/FAQ_What_is_an_Eclipse_product

[2] http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-25.htm

[3] http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-25.htm

▻ **Bundle** - In Eclipse the smallest unit of modularization is a plug-in. The terms plug-in and bundle are (almost) interchangeable. An Eclipse plug-in is also an OSGi bundle and vice versa.[4]

P2 Provisioning Platform Terminology

Based loosely on the Eclipse P2 Wiki page[5] the following list disambiguates P2 related terms.

▻ **P2 Repository** - A *P2 repository* consists of a *metadata repository* and an *artifact repository*, which do not neccessarily have to reside on the same machine. We will often refer to it simply as *repository* when it is not needed to differentiate between its components.

▻ **P2 Client** - A *P2 client* is a P2 enabled Eclipse RCP application which performs updates or installations of its components from a *repository*

▻ **Metadata Repository** - A *metadata repository* contains *installable units* and a P2 enabled RCP application contacts it in order to retrieve metadata.

▻ **Artifact Repository** - An *artifact repository* contains *artifacts*. The client selectively transfers *artifacts* once it has processed the metadata from the *metadata repository* and determined which *artifacts* to acquire.

▻ **Installable Unit** - An *installable unit (IU)* is only metadata describing things that can be provisioned. So an IU for a bundle is not the bundle but a description of the bundle: its name, version, capabilities, requirements, etc.. The bundle JAR is an *artifact*.

▻ **Artifact** - An *artifact* is the actual content being installed by a client. Bundle JARs and executable files are examples of artifacts.

▻ **Provisioning Profile** - A *provisioning profile* is a list of *IUs* that make up a system. Whenever an *IU* is installed, its metadata is added to the profile and its *artifacts* can then be accessed. Therefore, a *provisioning profile* is the target of *provisioning operations*.

[4] http://www.vogella.com/articles/OSGi/article.html
[5] http://wiki.eclipse.org/Equinox/p2/Concepts

- **Provisioning Planner** - A *provisioning planner* is responsible for determining what should be done to a given profile to reshape it as requested. That is, given the current state of a profile, a description of the desired end state of that profile and metadata describing the available IUs, a planner produces a list of provisioning operations (e.g., install, update or uninstall) to perform on the related IUs.

- **Provisioning Agent** - The provisioning infrastructure on client machines is generally referred to as the *provisioning agent*. *Agents* can manage themselves as well as other profiles. An *agent* may run separate from any other Eclipse system being managed or may be embedded inside of another Eclipse system. *Agents* can manage many profiles and a given system may have many agents running on it.

- **Provisioning Phase** - *Provisioning operations* usually consist of several steps or *provisioning phases*. Examples for *provisioning phases* are the *configure, install, uninstall* and *unconfigure* phases native to Eclipse RCP applications.

- **Touchpoint** - A *touchpoint* is a part of the engine that is responsible for integrating the provisioning system to a particular runtime or management system. P2 enabled RCP applications have several default touchpoints, corresponding to *provisioning phases* and sets of default actions (such as unzip, copy, bundle installation and many more).

- **Advice File** - An *advice file*, often found as *p2.inf* on the file system, offers capabilities to add information to a P2-enabled application at build time. It can be used to publish custom actions, add *repositories* or to define requirements, among other things.

4.2 Components

The software system to be implemented can be semantically broken down into four pieces, as shown in Figure 4.1. On the client side (shown on the right in Figure 4.1) there is a collection of cloud profiles and a P2 client . The server side also has a collection of cloud profiles, contained in a P2 repository, as well as an administrative component that allows administrative interaction.

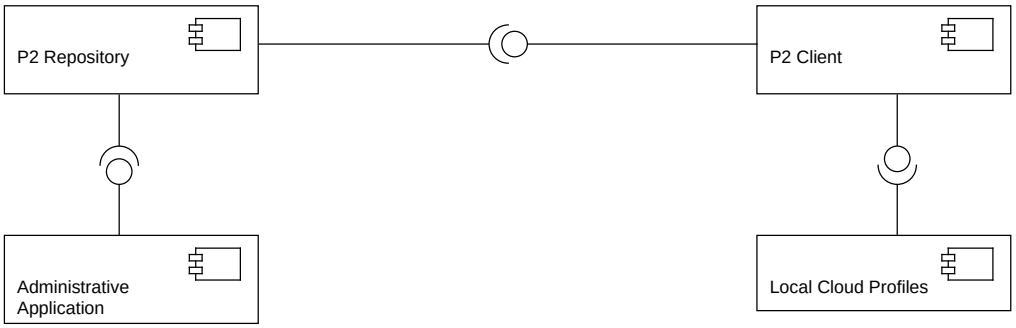

Figure 4.1. Components

4.3 Repository Design

A *P2 repository* consists of a *metadata repository* and an *artifact repository*, which do not neccessarily have to reside on the same machine. On disk a P2 repository looks as follows:

```
repository/
        artifacts.( xml/jar )
        content.( xml/jar )
        plugins/
                plugin-version.qualifier.jar
        features/
                feature-version.qualifier.jar
```

The repository itself is completely file based and is not running any software. It can be accessed as a local file or via HTTP, the P2 client then reads its metadata and computes the updates to be made. In case the client needs to acquire new features or plugins the *artifacts.xml* file is consulted to find out their download location. During implementation, a plugin for cloud profiles will be created and its metadata is written to the repository,

therefore the repository itself is not really implemented, but the contained *installable units* that will be created constitute it.

4.4 P2 Client Design

The P2 client design for integration into CloudMIG Xpress follows the official example to enable updating at startup of CloudMIG Xpress.[6] However, the exemplary code does not include triggering of additional actions and we will need to implement a custom touchpoint action to trigger a database update in CloudMIG Xpress, as described in Section 4.5. The following class diagram (Figure 4.2) illustrates the dependencies to P2 our new utility class, which is triggered from CloudMIG Xpress' *ApplicationWorkbenchWindowAdvisor* during startup, will have.

[6] http://wiki.eclipse.org/Equinox/p2/Adding_Self-Update_to_an_RCP_Application

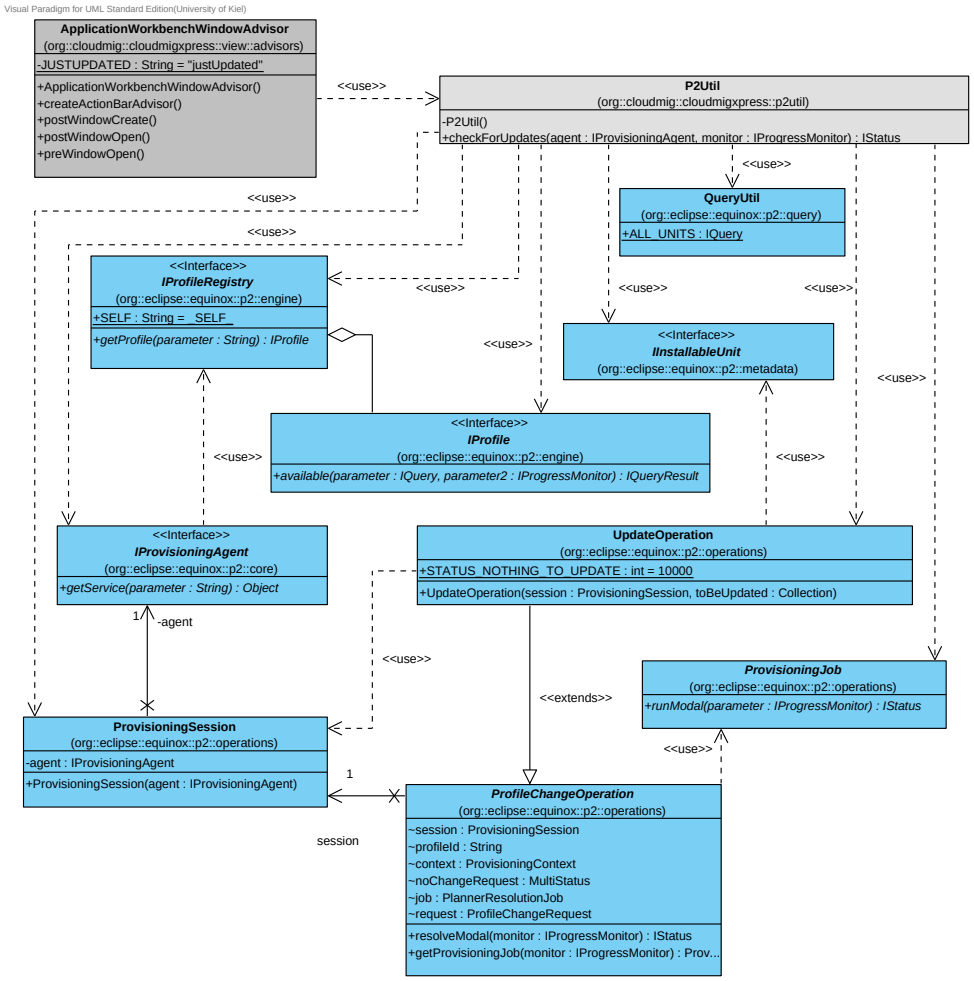

Figure 4.2. P2Util class and its dependencies into P2

4.5 Custom P2 Touchpoint Action

Several *touchpoint actions* are available natively in P2 (a list can be found in the Eclipse online-help.[7]) and while they are sufficient in many use cases, none of them provide the capability of calling arbitrary methods in the system whose *provisioning profile* we are managing. We want to update CloudMIG Xpress' Derby database with the new cloud profile information whenever an update occurs and need a way to achieve this. Of course we could just call additional methods within our *checkForUpdates* routine presented in section 5.1, but this is not ideal in terms of cohesion and coupling, as well as it provides a poor seperation of concerns. As previously mentioned, we want to keep that routine generic and capable of handling updates to any component of CloudMIG Xpress, thus placing feature specific code here, we would have to make sure it either does not run or does nothing in case another component is updated. Furthermore it would make us have to change this routine again in case the layout of our cloud profile feature changes, which is undesirable. A custom *touchpoint action* provides a far more elegant solution: By delivering a plugin which contains an executeable artifact with our feature during the update, we can access publicly available methods from CloudMIG Xpress and do everything we desire, however when the feature layout changes, we only need to alter this plugin to fit CloudMIG Xpress' interfaces and do not need to deploy a new version of CloudMIG Xpress itself, granting us greater flexibility and loose coupling. The following figure 4.3 illustrates how the custom touchpoint interacts with CloudMIG XPress' *CloudProfileFacade*, which is a facade astracting database and file operations relating to cloud profiles.

[7] http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/p2_actions_touchpoints.html

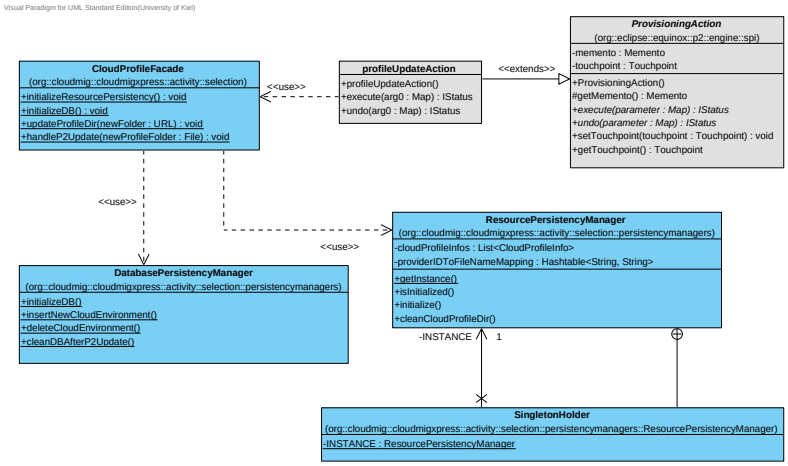

Figure 4.3. Custom P2 Touchpoint Action

4.6 Administrative Application Design

The administrative application will be held simple. It will contain a launcher class based on the Apache Commos CLI[8] to parse command line options and a utility class using JDOM[9] to perform the neccessary manipulations and version increases on the repository's metadata. The tasks it should perform are:

- add, delete and update cloud profiles contained in a plugin in the repository
- update the version number in the plugins manifest file
- update the version number in the feature.xml file of the feature containing the cloud profile plugin
- update the version numbers on provided and required installable units in the repository's metadata.xml file
- update the version numbers in the repository's artifacts.xml file
- relocation of plugin and feature jars to reflect the new version numbers

[8] http://commons.apache.org/proper/commons-cli/
[9] http://www.jdom.org/

4.7 Deployment

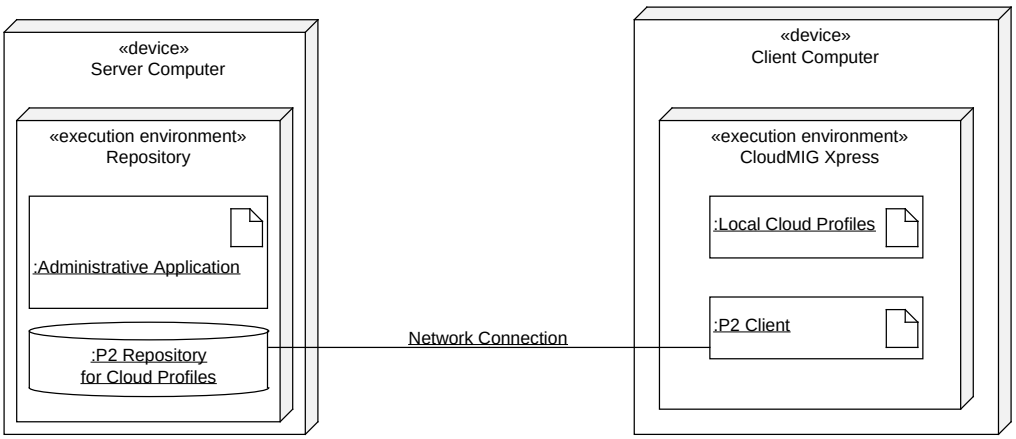

Figure 4.4. Deployment

Figure 4.4 describes the deployment of the software components. The server application is centrally hosted on a server computer, offering network access to clients. It includes the administrative toolset as well as the repository itself. There may be an arbitrary number of client computers (only one is exemplarily shown) running CloudMIG Xpress, these can connect to the server over a network to update their local set of profiles.

Chapter 5

Implementation

In this chapter we will describe the implementation, beginning with detailing key aspects of the implementation of P2 client and repository and finally the administrative application.

5.1 Enabling P2 in CloudMIG Xpress

CloudMIG Xpress, as previously stated, is built on Eclipse RCP and thus already contains P2 capabilities. We decided to make CloudMIG Xpress automatically update when it starts and now need to choose a fitting entry point to initiate the update process. The following code listing shows the extensions made to the *ApplicationWorkbenchWindowAdvisor* class' *postWindowOpen* method, which gets called after the application window has opened during CloudMIG Xpress' startup:

```
public void postWindowOpen() {
     ..
     final Activator activator = Activator.getDefault();
     final BundleContext bundleContext = activator.getBundle().getBundleContext();
     ServiceReference<?> reference = bundleContext
               .getServiceReference(IProvisioningAgent.SERVICE_NAME);
     if (reference == null) {
            return;
      }
      Object obj = bundleContext.getService(reference);
      final IProvisioningAgent agent = (IProvisioningAgent) obj;
      //if we're restarting after updating, don't check again.
      final IPreferenceStore prefStore = Activator.getDefault().getPreferenceStore();


      // check for updates before starting up. If update is performed, restart.
      IRunnableWithProgress runnable = new IRunnableWithProgress() {
          @Override
          public void run(IProgressMonitor monitor)
                  throws InvocationTargetException, InterruptedException {
              IStatus updateStatus = P2Util.checkForUpdates(agent, monitor);
```

```
            if (updateStatus.getCode() == UpdateOperation.STATUS_NOTHING_TO_UPDATE) {
            } else if (updateStatus.getSeverity() != IStatus.ERROR) {
                  PlatformUI.getWorkbench().restart();
            } else {
                  //LogHelper.log(updateStatus);
            }
        }
    };
    try {
        new ProgressMonitorDialog(null).run(true, true, runnable);
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
    }

}
```

Here we acquire the *provisioning agent* and set up a *runnable,* whose monitor we use to let a user watch the update progress. *Provisioning agent* and *monitor* are then passed to another new class *P2Util,* presented in the following listing:

```
public class P2Util {
    //do not allow instantiation
    private P2Util(){};
    // Check for updates to this application and return a status.
    public static IStatus checkForUpdates(IProvisioningAgent agent,
            IProgressMonitor monitor) throws OperationCanceledException {
        // obtain the provisioning session
        ProvisioningSession session = new ProvisioningSession(agent);

        // get installed units
        IProfileRegistry profileRegistry = (IProfileRegistry) agent
                .getService(IProfileRegistry.SERVICE_NAME);
        IProfile profile = profileRegistry.getProfile(IProfileRegistry.SELF);
        Collection<IInstallableUnit> alreadyInstalled = profile.available(
                QueryUtil.ALL_UNITS, null).toUnmodifiableSet();

        for (IInstallableUnit iu : alreadyInstalled) {
            System.out.println(iu);
        }

        // create an updateOperation for the installed units
        UpdateOperation operation = new UpdateOperation(session,
```

```
                alreadyInstalled);

        // perform the update
        SubMonitor sub = SubMonitor.convert(monitor,
                "Checking for application updates...", 200);
        IStatus status = operation.resolveModal(sub.newChild(100));
        if (status.getCode() == UpdateOperation.STATUS_NOTHING_TO_UPDATE) {
            System.out.println("##NOTHINGTOUPDATE###");
            return status;
        }
        if (status.getSeverity() == IStatus.CANCEL) {
            System.out.println("##CANCELED###");
            throw new OperationCanceledException();
        }

        if (status.getSeverity() != IStatus.ERROR) {
            System.out.println(status.getSeverity() + "0####"
                    + status.getCode());
            ProvisioningJob job = operation.getProvisioningJob(null);
            if (job != null) {
                status = job.runModal(sub.newChild(100));
                if (status.getSeverity() == IStatus.CANCEL) {
                    throw new OperationCanceledException();
                }
            }
        }
        System.out.println(status);
        return status;
    }

}
```

The *P2Util* class has only static methods and is not intended to be instantiated. We use the agent handed to the *checkForUpdates* method to create a new provisioning session, query the *provisioning profile* to acquire a list of already installed *IUs* and then let a new *updateOperation* find out if updates for an *IU* are available. If there are updates available, they are resolved, to make sure there are no conflicts in the profile, and performed. This implementation of the update process is generic and not limited to the updating of cloud profiles, so it will not have to be changed when we decide to provide updates to other components as well. Lastly, CloudMIG Xpress still has no information about which repositories to contact for updats or their respective locations. We place an *advice file* into the *META-INF* directory in the CloudMIG Xpress sources to provide this information. An example of such a file, used

to provide information about a local repository I used for testing, can be seen in the next listing:

```
instructions.configure=\
addRepository(type:0,location:http${#58}//download.eclipse.org/releases/indigo/);\
addRepository(type:1,location:http${#58}//download.eclipse.org/releases/indigo/);
```

This *advice file* adds the *addRepository* instruction to the *configure provisioning phase*. The *type* parameter can be "0", indicating a *metadata repository*, or "1", for an *artifact repository*. As we can see in this example, *metadata repository* and *artifact repository* can be collocated in the same folder without causing any problems. Using an *advice file* to define repositories is helpful for future changes, for example when our contributions will make it into the rolling release of CloudMIG Xpress and the location of the repository changes to an internet address, only this file needs to be changed and the remaining source code does not require any modifications. Now CloudMIG Xpress is ready to receive updates via P2 and we can move on to the implementation of the *P2 repository*, which is described in the next section.

5.2 Creating a P2 Repository

As mentioned in section 4.1, we will have to create a *metadata repository* and an *artifact repository*. This is straightforward, as an empty repository is nothing more than a directory containing a *metadata file*, named *content.xml* by contract, and an *artifacts file*, respectively named *artifacts.xml*. *Features* and *plugins* in the repository are kept in subfolders (named *features*, *bundles*, etc. for the *installable units* and *artifacts* for their *artifacts*), and entries registering them are created in the repositories' *metadata file* and *artifact file*. In this section we will describe the implementation of an Eclipse RCP *feature* holding our cloud profiles and a *plugin* providing a custom touchpoint to offer additional actions during updates, which is neccessary because the existing *provisioning actions* provided by the P2 platform are not wholly sufficient for our envisioned update process. At first the creation of a new *touchpoint action* will be detailed, then the cloud profiles feature implementation and finally the complete repository will be presented.

Custom Touchpoint Action

The implementation of a custom *touchpoint action* in a plugin requires three steps: placement of a *p2.inf* advice file in the plugins *META-INF* directory, declaring the touchpoint action in the *plugin.xml* file, as the following listing shows, and implementing the action itself.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
    <extension
         point="org.eclipse.equinox.p2.engine.actions">
```

```
      <action
            class="org.cloudmig.cloudmigxpress.cloudprofiles.touchpoint.ProfileUpdateAction"
            name="profileUpdateAction"
            touchpointType="org.eclipse.equinox.p2.osgi"
            touchpointVersion="1.0.0"
            version="1.0">
      </action>
   </extension>>
</plugin>
```

The class referenced in the extension point has to extend the class *provisioningAction*[1] from the package *org.eclipse.equinox.p2.engine.spi*. Shown next is an example for such a class.

```
public class ProfileUpdateAction extends ProvisioningAction {
    @Override
    public IStatus execute(Map<String, Object> arg0) {
        try{
                String profilePath = arg0.get("target")+File.separator+"CloudProfiles";
                profilePath=profilePath.replace('\\','/');
                CloudProfileFacade.handleP2Update(profilePath);
        }catch(Exception e){
            e.printStackTrace();
        }

        return Status.OK_STATUS;
    }
}
```

Finally, the *advice file* is used to announce that our plugin provides a *touchpoint action* using the provides keyword, as seen in the next listing.

```
provides.0.namespace=org.eclipse.equinox.p2.osgi
provides.0.name=profileUpdateAction
```

A more complete example for advice file attributes can be found in the Eclipse Wiki[2].

[1] http://help.eclipse.org/indigo/ntopic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/equinox/p2/engine/spi/ProvisioningAction.html

[2] http://wiki.eclipse.org/Equinox/p2/Customizing_Metadata

Plugin containing Cloud Profiles

We will now have a look at a *plugin* containing our cloud profiles and including the plugin shown in the previous section. When this plugin is installed, the custom *touchpoint action* from the touchpoint action plugin should be executed. We can achieve this by specifying this action in the cloud profile plugin's *advice file* (collocated with the *MANIFEST.MF* file), which can loosely be considered a counterpart to the touchpoint plugins advice file. We specify a requirement for the touchpoint plugins *touchpoint action* to be present and add it to the instructions of the install *provisioning phase*, so it is appended to any existing actions of that phase. The *p2.inf* code listed now accomplishes this:

```
metaRequirements.0.namespace=org.eclipse.equinox.p2.osgi
metaRequirements.0.name=profileUpdateAction

instructions.configure=profileUpdateAction(target:${installFolder});
instructions.configure.import=\
org.cloudmig.cloudmigxpress.cloudprofile.touchpoint.profileUpdateAction
```

P2 Repository

Using Eclipses PDE Build Tools we can export the feature and make PDE create its metadata. This results in the generation of *metadata* and *artifact* repositories in the target folder. Listed below are excerpts from example *content.xml* and *artifacts.xml* files generated through PDE. content.xml:

```
<unit id='org.cloudmig.cloudmigxpress.cloudprofile.touchpoint' \
                version='1.0.0.201303240342'>
  <update id='org.cloudmig.cloudmigxpress.cloudprofile.touchpoint' \
                range='[0.0.0,1.0.0.201303240342)' severity='0'/>
  <properties size='1'>
    <property name='org.eclipse.equinox.p2.name' \
                value='Update Instruction Actions for Profile Updates'/>
  </properties>
  <provides size='4'>
    <provided namespace='org.eclipse.equinox.p2.eclipse.type' name='bundle' version='1.0.0'/>
    <provided namespace='osgi.bundle' \
                name='org.cloudmig.cloudmigxpress.cloudprofile.touchpoint' \
                                version='1.0.0.201303240342'/>
    <provided namespace='org.eclipse.equinox.p2.osgi' name='profileUpdateAction' \
                                                                version='0.0.0'/>
    <provided namespace='org.eclipse.equinox.p2.iu\'
                name='org.cloudmig.cloudmigxpress.cloudprofile.touchpoint' \
                version='1.0.0.201303240342'/>
```

```
  </provides>
  <requires size='6'>
    <required namespace='osgi.bundle' name='org.eclipse.equinox.p2.engine' range='0.0.0'/>
    <required namespace='osgi.bundle' name='org.cloudmig.cloudmigxpress' range='0.5.0'/>
    <required namespace='java.package' name='org.eclipse.core.runtime' range='3.4.0'/>
    <required namespace='java.package' name='org.eclipse.equinox.p2.core' range='2.0.0'/>
    <required namespace='java.package' name='org.eclipse.equinox.p2.metadata' range='2.1.0'/>
    <required namespace='java.package' name='org.eclipse.equinox.p2.repository' range='2.0.0'/>
  </requires>
  <artifacts size='1'>
    <artifact classifier='osgi.bundle' \
                id='org.cloudmig.cloudmigxpress.cloudprofile.touchpoint' \
                                                              version='1.0.0.201303240342'/>
  </artifacts>
...
```

artifacts.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<?artifactRepository version='1.1.0'?>
<repository name='Exported Repository'\
         type='org.eclipse.equinox.p2.artifact.repository.simpleRepository' version='1'>
  <properties size='2'>
    <property name='p2.timestamp' value='1364092951956'/>
    <property name='p2.compressed' value='true'/>
  </properties>
...
```

5.3 Adminsitrative Application Implementation

The Administrative application employs DOM operations. An example for such an operation is given in the following listing:

```
private void updateMetadata() {
        Element rootElement = content.getRootElement();
        for (Element unit : rootElement.getChildren("units").get(0)
                        .getChildren("unit")) {
                if (inScope(unit.getAttributeValue("id"))) {
                        incrementVersion(unit);
                        for (Element providedIU : unit.getChildren("provides").get(0)
                                        .getChildren("provided")) {
                                if (inScope(providedIU.getAttributeValue("name"))) {
                                        incrementVersion(providedIU);
                                }
                        }
                }
        }
...
```

Here, the variable *content* represents the *content.xml* file. We use the DOM-Tree and acquire its children of type unit to update the installable units versions. Most operations performed by the administrative application follow this scheme and thus do not need to be presented in greater detail here.

Chapter 6

Evaluation

This chapter is dedicated to evaluate the software resulting from our implementation and the suitability of our approach to reach the given goals. At first we will evaluate the administrative application in a component test and then perform integration tests to see how well the individual components work in conjunction with each other. A deployment test is performed to determine wether deploying our repository and administrative application is possible.

6.1 Component Testing of the administrative Application

In this section we present test cases for the administrative application and the accompanying test results. These tests were mostly performed manually, as this seemed the most feasible approach. Other Components were not tested in component tests other than code reviews, as for those components their interaction is most important and will be tested in integration tests in the Section 6.2.

6.1.1 Test Case 1: Metadata Updates

Scenario:
The administrative application is tested on a preexisting repository. No cloud profiles are added, deleted or updated, only metadata is altered. This functionality forms the basis for the following three test cases, as it is included in each of them. We need to be able to bump versions in the repositories metadata to signal the existence of updates to clients. The repository we are manipulating contains a feature consisting of two plugins, one of those is supposed to have its version increased, the other should be left as is.

Expected Behaviour:
After the metadata update, the repositories *content.xml* and *artifact.xml* files contain updated references with a new version number for the updated plugin, version references for the other are not altered. Version changes are included in the updated plugins manifest file, the *feature.xml* file in the feature containing the plugins reflects the version changes and the updated plugin and feature are repackaged in a jar file with a filename corresponding to the new version.

Actual Behaviour:
Working as expected.

Result:
This test is passed without limitations.

6.1.2 Test Case 2: Adding a Profile to the Repository

Scenario:
A cloud profile is added to the repository via the command line utility.

Expected Behaviour:
After the operation the cloud profile plugin jar contains the new cloud profile among those present before the update, metadata is updated reflecting a version increase. **Actual Behaviour:**
Working as expected.

Result:
This test is passed without limitations.

6.1.3 Test Case 3: Deleting a Profile from the Repository

Scenario:
A cloud profile is deleted from the repository via the command line utility.

Expected Behaviour:
After the operation the cloud profile plugin jar contains the cloud profiles present before the update, except for the one we deleted. Metadata is updated reflecting a version increase.

Actual Behaviour:
Working as expected.

Result:
This test is passed without limitations.

6.1.4 Test Case 4: Updating a Profile to the Repository

Scenario:
A cloud profile is updated via the command line utility.

Expected Behaviour:
After the operation the cloud profile plugin jar contains the updated cloud profile among those present before the update, sans its outdated counterpart, metadata is updated reflecting a version increase.

Actual Behaviour:
Working as expected.

Result:
This test is passed without limitations.

6.2 Integration Tests

The following two test cases serve the evaluation of the interaction between the software components.

6.2.1 Test Case 1: Updating CloudMIG Xpress from our Repository

Components tested:
P2 update handler in CloudMIG Xpress and P2 Repository for cloud profiles

Scenario:
A local repository contains a newer version of the cloud profile plugin than is contained in CloudMIG Xpress. The repository is added to CloudMIG Xpress' P2 meta information via its *advice file*. CloudMIG Xpress is then started.

Expected Behaviour:
CloudMIG Xpress should register the availability of an update and perform it. After a restart the collection of cloud profiles in CloudMIG Xpress reflects the one in the newer plugin version, the newer feature and plugin are installed in CloudMIG Xpress' provisioning profile, replacing the older ones.

Actual Behaviour:
Working as expected.

Result:
This test is passed without limitations.

6.2.2 Test Case 2: Full Integration Test

Components tested:
P2 update handler in CloudMIG Xpress, P2 Repository for cloud profiles and adminsitrative Application

Scenario:
A local repository is updated using the administrative application. The repository is added to CloudMIG Xpress' P2 meta information via its *advice file*. CloudMIG Xpress is then started.

Expected Behaviour:
An update through the administrative application bumps the versions in the repositories metadata, the repository retains validity and an available update should be detected by CloudMIG Xpress on startup. The update is then performed and after a restart the collection of cloud profiles in CloudMIG Xpress reflects the one contained in the updated repository.

Actual Behaviour:
Working as expected.

Result:
This test is passed without limitations.

6.3 Deployment Test

In a final test scenario we will deploy the components into an environment that is close to the one intended for production use.

6.3.1 Test Scenario 1: Performing Updates over the Internet

Scenario:
The repository is placed on a server computer and served over HTTP via the apache web server and its address is added to a local CloudMIG Xpress installation's *advice file*. CloudMIG Xpress is then started to perform an update over HTTP.

Expected Behaviour:
CloudMIG Xpress should perform the update as it does for local repositories. The same behaviour as in Integration Test Case 1 is expected.

Actual behaviour:
CloudMIG Xpress contacts the repository and downloads its *content.xml* file. It then gets stuck in a loop and downloads this file again until a retry limit is reached and finally fails, stating no updates were found.

Result:
This test fails. To make sure that this is not due to the apache configuration, the test was repeated using a local instance of PHP's builtin web-server, with the same result. The same repository when accessed via *file* protocol instead of HTTP allows for flawless updating, thus ruling out a problem with the repository itself. To further debug this scenario the official Eclipse repository (also available via HTTP) was added to CloudMIG Xpress' *advice file*. When trying to update, CloudMIG Xpress displays the same erratic behaviour.

Chapter 7

Related Work

Apart from the work done on the CloudMIG Approach and CloudMIG XPress at the Christian-Albrechts-Universität, other approaches at evaluating a given software systems suitability for a specific cloud environment are developed. One such approach is CloudCmp, which claims to provide a systematic comparator of the performance and cost of cloud providers [Li and andMing Zhang 2010]. Further research approaches the problem from a different angle, researchers from romania and italy try to create a cross platform cloud API, instead of profiling individual products [Petcu et al. 2011].

Chapter 8

Conclusions and Outlook

In this chapter we conclude the work. At first a summary is given, then a brief discussion is held and finally possible future work is presented.

8.1 Summary

We have designed and implemented a software that allows for distribution of cloud profiles to CloudMIG Xpress users and maintenance of the cloud profile repository. Two out of three major goals were reached and the software is mostly ready for production use. It will most likely be included in the CloudMIG Xpress release. In our approach we have applied techniques of software engineering, first eliciting requirements, then designing our software's architecture and finally implementing and testing the software. Problems arose from the use of the P2 provisioning platform, mostly due to its lack of good documentation and no previous experience in using P2, but those do not outweigh the benefits P2 offers for our purpose.

8.2 Discussion

In this section i will discuss wether the problems arising when using the P2 provisioning platform could have been avoided. We have to consider that i had no previous experience in using P2, for a developer who worked with P2 before, the implementation would have been much more intuitive. It is highly dissatisfying that the final deployment test failed, especially because any tutorial and documentation states that there is no additional work needed to move a local repository to a server in the internet. As usually, the devil is in the details, and I am positive that fixing this remaining bug can be achieved easily, once it's origin is determined.
Working with Eclipse P2 was disappointing in some other aspects as well. The documention claims that it is suited to manage arbitrary files, however no documentation on how to achieve this is provided and we needed to structure our repository in features and plugins, a simple collection of arbitrary files would have been preferrable, as it would have facilitated the implementation of the administrative application. Comparing the effort it took to create the repository and P2 client to the effort of implementing the administrative application yields an interesting result: While the administrative application has almost three times as

many lines of code as the repository and client together, it took only about one fifth of the time needed to create those two components to implement the administrative application. This demonstrates how important it is to create a comprehensive documentation for software, as I was mostly left to rely on discussions in forums and mailing lists when trying to solve an arising problem in P2. Most of the time, solving the problem proved trivial in the end, but always required deeper knowledge of P2 that was not officially provided. Furthermore, P2 has some design quirks that seem arbitrary, for example the location for placement of advice files that varies between products, features and plugins. The default use case of updating or installing a feature or plugin are documented well enough, but as soon as your use cases vary slightly from those defaults or include more actions, programming with Eclipse P2 is a journey in the dark, especially as P2 itself consists of numerous components and the programmer is left to figure out which of those components causes a problem. The lack of detailed debugging output from those components adds to the general frustration of working with P2 and often leaves the programmer with no choice but to fix problems by trial and error.

8.3 Future Work

This section is split in two subsections. On the one hand, i will detail further steps that I personally will take and on the other a brief overview of possible further research is presented.

Continuing Work after concluding this Thesis

I will, on a voluntary basis, aid Sören Frey in preparing the next CloudMIG Xpress release to include the components that I developed. At the time of writing no server to place the repository was provided, once it is decided where the repository will be hosted, I will aid in deploying the repository and eliminating the software's remaining bugs.

Future research

Accomplishing Goal G3 and extending the repository to other components of CloudMIG XPress might be covered by future work. Future research could further evaluate how our approach of profiling cloud products holds up in the quickly changing cloud computing landscape of today. As cloud computing is still in its infancy and game changing technology may be introduced any day, all of its facets will remain subject to research in the near future.

Chapter 9

Annex

Attached below is an optical disc containing the software developed during this work. On the disc a readme.txt file is included, it explains the discs directory structure and included software.

Bibliography

[Armbrust et al. 2009] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: a berkeley view of cloud computing. In: *Above the Clouds: A Berkeley View of Cloud Computing*. UC Berkeley Reliable Adaptive Distributed Systems Laboratory, Feb. 2009, page 1. (Cited on page 5)

[Frey and Hasselbring 2011] S. Frey and W. Hasselbring. The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. *International Journal on Advances in Software* 4.3 and 4 (2011), pages 342–353. (Cited on pages 7, 8)

[Frey et al. 2012a] S. Frey, W. Hasselbring, and B. Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process* (2012), n/a–n/a. (Cited on pages 1 and 8)

[Frey et al. 2012b] S. Frey, E. Schulz, M. Rau, and K. Hesse. Cloudmig xpress 0.5 beta user guide. In: *CloudMIG XPress 0.5 Beta User Guide*. Christian Albrechts Universität Kiel, Software Engineering, June 2012, page 3. (Cited on page 8)

[Hors et al. 2004] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document object model (dom) level 3 core specification. In: *Document Object Model (DOM) Level 3 Core Specification*. The World Wide Web Consortium (W3C), Apr. 2004, pages 13–15. (Cited on page 9)

[Kowalczyk and Kwiecinska 2009] K. Kowalczyk and A. Kwiecinska. Model-driven software modernization (2009). (Cited on page 5)

[Li and andMing Zhang 2010] A. Li and X. Y. S. K. andMing Zhang. Cloudcmp: comparing public cloud providers. *IMC '10 Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), pages 1–14. (Cited on page 45)

[Mell and Grance 2011] P. Mell and T. Grance. The nist definition of cloud computing. In: *The NIST definition of cloud computing*. National Institute of Standards and Technology, Sept. 2011, pages 6–7. (Cited on pages 1 and 5)

[Petcu et al. 2011] D. Petcu, C. Craciun, and M. Rak. Towards a cross platform cloud api. *Towards a cross platform cloud API* (2011). (Cited on page 45)