Automating update processes for web applications using the generative programming paradigm

Bachelor thesis
by
Oliver Schlicht

presented at
Chair of Software Technology
Professor Colin Atkinson, Ph. D.
Department of Mathematics and Computer Science
University of Mannheim

in cooperation with

September 2006

Tutor:
Gutheil, Matthias • Chair of Software Technology • University of Mannheim

I. Contents

II. Abstract

In contrast to traditional programming approaches like object orientation, the generative programming paradigm is about designing software families instead of single software systems. The system family acts as base for creating a generator. This generator software generates concrete highly specialized system instances using the family's sources combined with a destination domain specific configuration file. Therefore, the generated systems sometimes highly vary regarding composition and functionality depending on given requirements.

An update server that propagates changes made to the source code repository to the instances of family is to be specified. Based on a detailed requirements analysis, an update server system has to be specified as well as a prototype implementation in J2EE.

III. Summary

After giving you a short introduction into generative programming in chapter 1.1, I will give an overview about the system scenery I have to deal with and the resulting piece of software has to be integrated in. Afterwards I am going to have a look at a software systems lifecycle in generative programming environments, which should clarify the driving forces for this project.

In chapter 2 I will summarize the high level requirements for the software system to be built to define the update process steps, which will be the heart of the project in chapter 3. Thereby I will a give a reasonable rationale for a given way of partitioning the process and describe the resulting phases in detail in chapter 3.3.

As I have won a deeper understanding of the problem domain and requirements, I will explain the architecture and platform selection reasons in chapters 4.1 to 4.2. Then I will give a more detailed look on architectural layers as well as specific important components of the system.

In chapter 5 I will give an outlook on future versions of the system summarizing all the acquainted features we know we will have to deal with.

The appendix contains a detailed description of the packages of the software as well as a list of the used 3rd party libraries.

IV. Abbreviations

AOP	Aspect Oriented Programming
API	Application Programmer's Interface
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DI	Dependency Injection
DS	Destination System
DSL	Domain Specific Language
DTO	Data Transfer Object
EAR	Enterprise ARchive
EJB	Enterprise Java Beans
FTP	File Transfer Protocol
GDM	Generative Domain Model
GP	Generative Programming
HTML	HyperText Markup Language
IC	Implementation Component
IDE	Integrated Development Environment
JDBC	Java DataBase Connectivity
JMS	Java Messaging System
OO	Object Orientation / Object Oriented
PHP	PHP Hypertext Preprocessor
POJO	Plain Old Java Object
SS	Software System
SF	Software Family
SVN	SubVersioN
URL	Uniform Resource Locator
WAR	Web Archive
WSDL	Web Service Description Language
XML	eXtensible Markup Language

1 Introduction

1.1 Generative Programming

> *„Generative Programming*[1] *(GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge." (Czarnecki, 2002)*

Nowadays software engineering is taking the step other industries have bridged already a few decades before. While object orientation (OO) has strongly developed over the years, one main expectation towards it has never really been fulfilled – reusability in a sense that the automobile industry for example uses it.

GP explicitly addresses that way of developing software as it intends to move its focus from designing single software systems (SS) to designing software system families (SF) like banking software or insurance software. Furthermore, the aim is to generate concrete SS based on the *generative domain model* (GDM), which consists of *problem space*, *configuration knowledge*, which is implemented as a generator system and *solution space* (see Figure 1-1).

The problem space consists of all features of the SF and is implemented by a domain specific language (DSL), which allows formally capturing stakeholder's requirements and hiding the generators complexity behind. The configuration knowledge describes rules of allowed and disallowed feature combinations, default settings and so on. Finally, the solution space consists of implementation components (IC) that might be classes, functions or only a few lines of source code. ICs tend to be highly reusable and redundant to the least possible extend. With help of ICs and a domain specific configuration file the generator is able to create a destination system (DS) that can be considered as an instance of a SF. At this, DS is semantically equivalent to SS, but a generated system – not specifically developed.

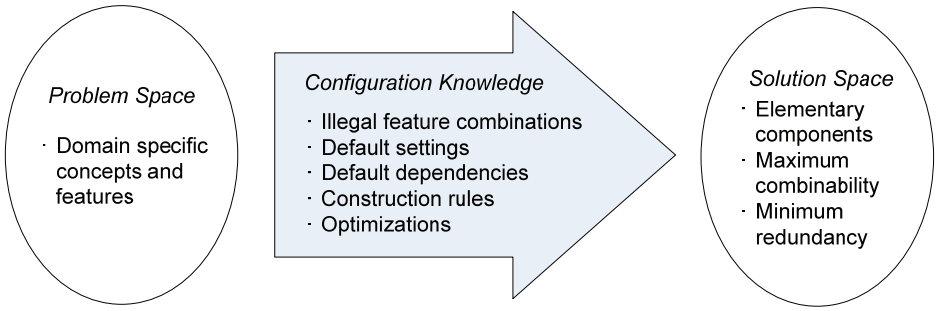

Figure 1-1: Elements of a generative domain model [CE00]

[1] For further information refer to [CE00]. I'll only give a short summary on that topic due to the fact that a complete overview would go beyond the scope of my work.

1.1.1 Domain

The generative domain model in some respect extends the term *domain* from the first to the second definition of domain as Mark Simon gives them in [SCK+96]:

1. Domain as "the real world"
2. Domain as a set of systems

While OO mostly uses the first version of the definition to determine the scope of the system to build, GP refers to the second one including the knowledge of how to compose those given sets to software systems [CE00]

With the extension of the term "domain", GP defines a new development approach which wraps the OO development process – Domain Engineering (DE)

1.1.2 Domain Engineering

Domain Engineering can be divided into micro processes *domain analysis*, *domain design* and *domain implementation*.

Domain analysis is mainly about discovering all features of a certain SF, identifying dependencies and interdependencies of features and defining default presets to handle missing requirement declarations. Furthermore you clearly define the scope of the system family's domain. As in the known OO process brainstorming and consultation of system family's stakeholders are the main ways to do. The formal approach for this process is *feature modeling*. For further information refer to [CE00].

In the domain design phase you identify implementation components, develop the SF- and DS architecture, the DSL and make first decisions, how to map the GDM to a concrete implementation platform (e.g. C++ Template metaprogramming or JavaBeans + AspectJ).

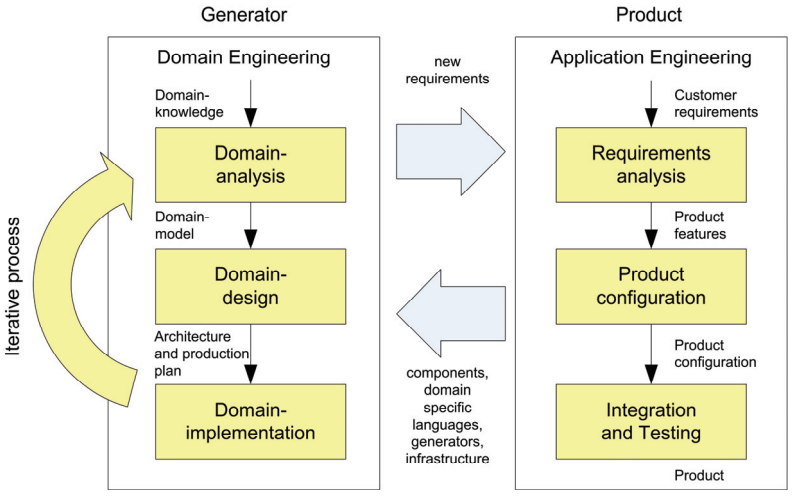

Figure 1-2: Software development based on Domain Engineering [CE00]

At this point you are ready to implement the generator system. Among other things it has the responsibility to check, if – presuming a given configuration file – the system is buildable at all.

The domain implementation covers the programming of the generator system and implementing ICs.

Developing a generative system actually covers two working cycles. You initially work with the destination system, developing architecture, design and features and shift it into the generator. As developing ICs directly in the generator is quite time consuming due to the lack of tools these days, this approach has proved its worth.

1.2 Current system environment and working process

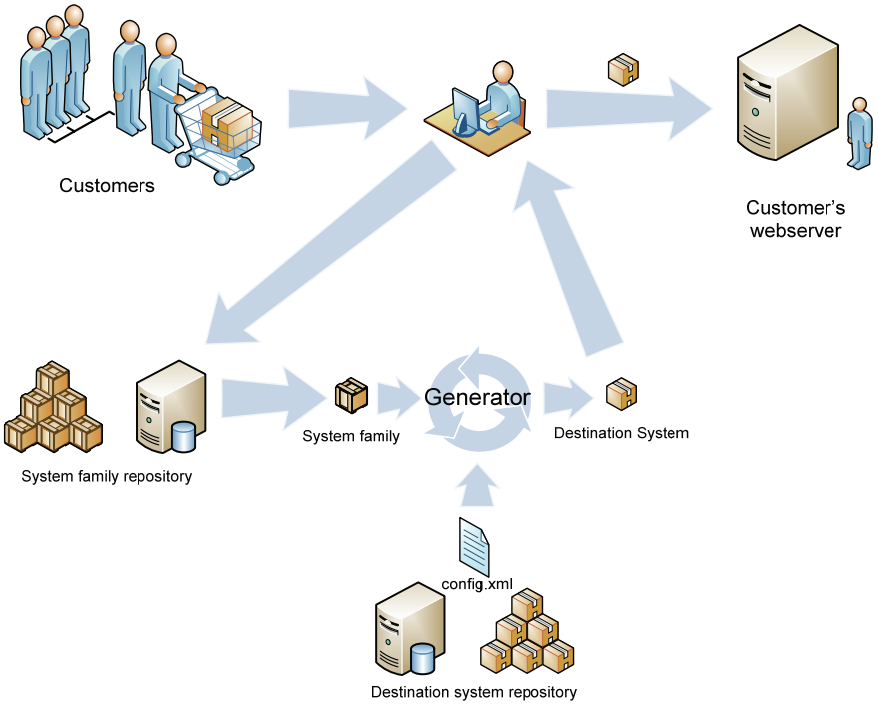

Figure 1-3 - System scenery (before)

The given system environment we have to deal with is basically based on former research work in a project at Fachhochschule Kaiserslautern. In this project Mathias Henss, Thomas Wollny and Prof. Dr. Ulrich Eisenecker developed a base implementation of the generative programming model described above. In three further years of practical application this implementation has become mature and enhanced.

As every software system being developed it is held in a version management system – in our case Subversion (SVN). We differentiate two kinds of repositories, what refers to the content those systems store, not their functionality. The system family's repository stores the generator systems, whereas the destination system's repositories stores generated systems due to legal issues as well as a customer specific configuration file which is explained later. Because the generator system is being developed and changes in functionality we have to store the outcomes of a given software generator at a given point in time to avoid not being able restoring exact generation results with a advanced generator later.

We mostly deal with web systems, so another important part of the scenery is the customer's web server. It can either be located in our computer center or in the cutomer's, which leads to different requirements in shipping the software.

The working process as it is executed right now can be described as follows: The customer configures its software system via a simple web form which produces a XML configuration file containing a DSL set. This file is stored in the destination system's repository for further usage and acts as a system specification. Now we manually have to check out the SF's generator from the SVN repository and equip it with the customer specific configuration file. Afterwards the destination system is generated, checked in into the appropriate repository and finally installed on the customer's web server.

1.2.1 Version handling

To be efficiently able to track different versions of system families and destination systems we need to specify a version management strategy. As Subversion only increases revision numbers with every commit, we need some larger grained mechanism to identify release versions of a system family. Like every other version management system Subversion offers branches to unhinge a separate development path. We're going to use this branch mechanism to mark release versions or, more formally, versions a destination system can be updated to.

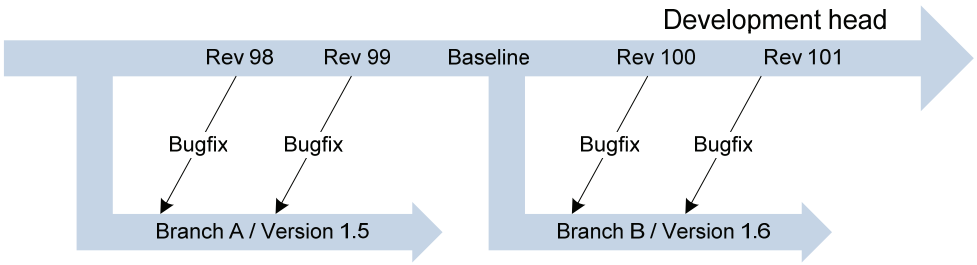

Figure 1-4 - Mapping release versions to branches

As the development baseline is being extended and maintained it is possible merge bugfixes into branches while increasing the revision of the main development path and the branch. So for a destination system we need to store the system family's branch and the revision number at update time. This way we can distinguish between bugfix updates (same branch, different revision) and version updates (different branches).

1.3 Evolution of software systems in GP environment

To efficiently develop update algorithms for GP systems it's reasonable to analyze how the evolution of a GP system differs from a development lifecycle of non-product-line-engineering systems.

The most important difference defers from the domain focus expansion described in 1.1. This ends up in not maintaining a software system as a whole but in maintaining software family components. Thus there is actually no development in the destination system but in the generator system as destination systems can be regarded as snapshot instances of a generator at a given point in time in combination with the configuration file.

2 High level requirements

2.1 Overall description

2.1.1 Purpose

Updating GP software systems needs a lot of manual work nowadays. Checking which customers really need an update can be really time-consuming due to the wide variety of possibilities of destination system configuration. Furthermore changes of database schemas require careful checking and transformation to ensure data integrity.

Even without the specialties of generative software systems it's tedious to update systems manually if the number of affected systems exceeds 10. For generative systems this is not manageable least of all.

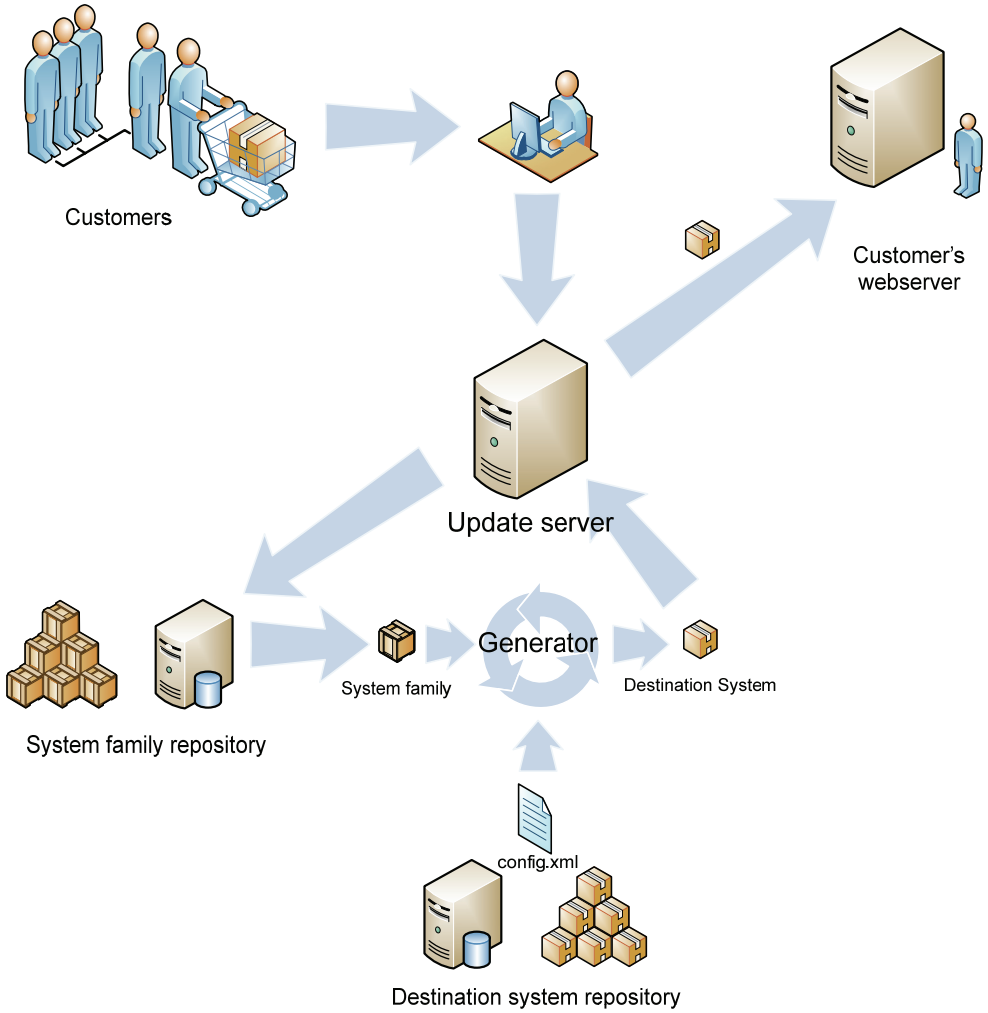

Figure 2-1 - System scenery (after)

2.1.2 Project scope

Goal of the project is to clearly identify requirements for an update server system that helps easing update processes and achieving an automation rate as high as possible. Main focus lies in defining the update process, identifying its parts, collaboration partners, necessities and driving forces. Based on this theoretical work a prototype shall be implemented to deliver a basis for further enhancements and increasing or changing requirements.

As our GP systems focus is primarily on web based systems in Java and PHP, the system explicitly shall address typical problems or tasks regarding client server software systems. Other kinds of systems (embedded software systems e.g.) are not in the focus of this project.

2.1.3 Project vision

The final system scenery shall look like Figure 2-1. An update server system shall take responsibility to execute the update process and act as an orchestrating component.

2.2 Product features

2.2.1 Management of collaborating entities

The system shall be able to manage software families, customers and destination systems where a destination system is defined as the link between a customer and a system family. It has to be enriched with information about the destination system server and a SVN repository that keeps track of the generated versions.

It has to be possible, to register, edit and deregister customers and system families from the system. Furthermore you have to be able to create, edit and remove destination systems from the update server.

2.2.2 Defining update jobs

The system shall allow you to define update jobs. An update job is defined as an update procedure for at least one customer and exactly one software system. The possibility to define an update job for all software systems of a single customer is to be left out.

An update job can either be timed as *immediately*, which means that the installation is run right after its definition, or to a given point of time. This is necessary due to the fact, that destination systems could be hosted and used in different time zones and you would not want to interrupt them working.

2.2.3 Executing update jobs

Defined update jobs have to be prepared and executed. If the update job has been scheduled to a certain point in time, it has to be *installed* right at this time. Preparation can be timed freely as long as it does not contradict execution time settings.

2.3 User classes

For now we simply have only one role of users, as up to this point the system will be run by a small set of persons only. So everyone who is allowed to enter the system is also allowed to manipulate all the data. As the system could be possible be handed to reselling companies, we are going to introduce a client system in later versions. Details are described in chapter 5.

2.4 Interface requirements

2.4.1 Communication interfaces

As the versioning of the software families is currently handled with Subversion the update server has to be able to communicate with SVN Repositories. Destination systems as we focus on here are mostly web systems, so that it is necessary to deploy them to the destination environment via FTP.

2.4.2 User interface

To easily manage system families, customers and other business entities and processes a web interface shall be implemented. It shall also be possible to get information about the systems' and update jobs' running status.

2.5 Nonfunctional requirements

2.5.1 Asynchronism

The update process itself shall be triggered asynchronously, as it can take a few minutes to check out the system family, generate it, prepare the system and install it on the remote server. Doing this, it shall be possible to keep track of the update process via the web interface.

2.5.2 Transactionality of the installation process

The installation process shall be wrapped into a transaction. This means that it is either successful or has failed. If it is successful, everything is fine. Otherwise, the old version of the system has to be up and running as it did before.

3 Process definition

In this chapter I will define the update process, its objectives, sub processes and identify the most important problems and challenges that appear in them. I try to describe them as comprehensive as possible, knowing that implementing all of the needed features is beyond the scope of this project. This seems reasonable to me to, because with a specification as complete as possible, extension points for further development can be implemented already now.

3.1 Objectives

As the update process is a complex process it is necessary to strip it down into smaller sub processes restraining complexity. Furthermore this helps to make parts of the process exchangeable. For this version I decided to have one single central update processing component that gets configured at startup time and thereby gets to know about the process steps. This provides a basis for further enhancements as you can read about in 5.1.

A very self-evident objective is to keep the design and implementation of the process as natural as possible to the real world process.

3.2 How to split the update process

As the set of destination systems can be arranged completely orthogonal to the process steps, we have two possibilities to split the process.

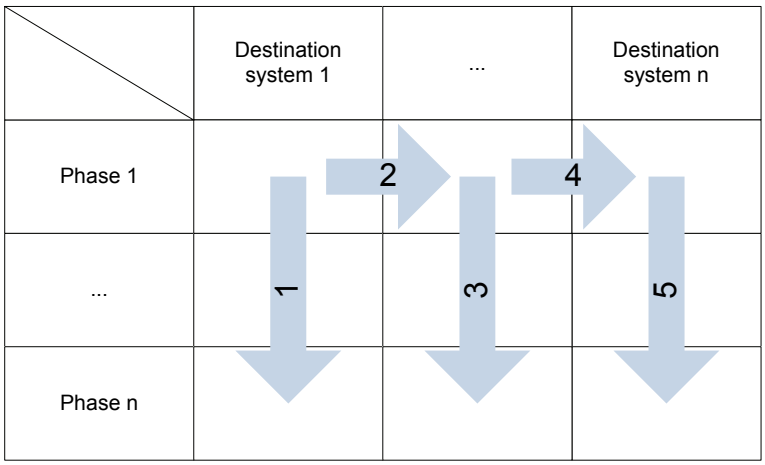

Figure 3-1 - Applying phases to each destination system

We could take the destination systems one after another, apply sub processes to each and switch to the next destination system afterwards (see Figure 3-1). This would be advantageous as we could finish destination systems step by step, not stopping all the systems if maybe phase three fails for the last destination system. The main disadvantage is that we can not apply a number of phases to all the systems, suspending the update job for a while and continue at a later point in time. As we want to schedule update jobs perhaps to 14 days in future this feature becomes highly critical. Furthermore this model would result in starting the update job right at scheduled time, e.g. in 14 days. If

there is an error occurring then and the update time is critical we get into serious trouble.

With the model shown in Figure 3-2 we can avoid those problems paying the fee not to be able to finish one destination system before the other one starts.

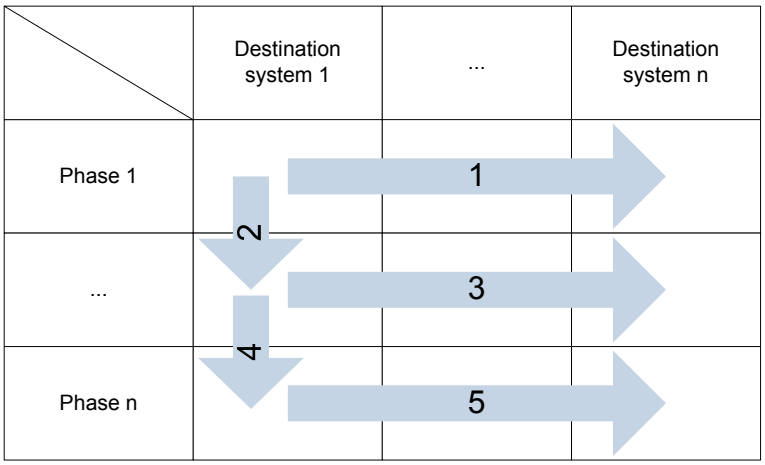

Figure 3-2 - Applying destination systems to each phase

Another reason to decide for this model is that we can therefore address the requirement in 2.2.3. We can start preparation right after the update job was defined as such, what can help to solve problems during preparation and does not endanger the successful installation as much as it would, if we started preparation right before execution time. Starting preparation as early as possible also makes sense as we cannot give detailed assumptions about the duration of the preparation at all because this highly depends on the given software family.

3.3 Process definition

So I decided to split the entire process into 4 large grained process parts: preparation phase, transfer phase, scheduled phase and installation phase.

3.3.1 Preparation phase

The preparation phase acts as the starting point of the entire process. What we need to do first is selecting the customers that are actually affected by a planned update.

Due to the development evolution process of a software family it is possible that in a new version of the SF we only made changes to 3 of 5 features. If there are customers whose destination systems do not include the 3 features changed, they therefore do not need to be updated in a classic sense. Those systems actually already are at the latest version, as a generation of the destination systems with the new generator would result in exactly the same system as with the old one. Therefore we only need to note that those systems are at a new version now.

If we have a valid list of affected customers the next step is to check out the generator from the system family's Subversion repository. To generate a customer specific destina-

tion system we have to equip the generator with the customer specific configuration file from the destination systems Subversion repository.

This is where the next point of trouble pops in. Changes to the system family could have caused a change to the configuration file format. So this is the place where the system has to check, whether such a change occurred and should the situation arise transforms the customer's configuration file into the valid new format. As this is also a complex topic and requires deep analysis of feature constellations in GP software systems we assume that configuration file formats do not change for this implementation.

As we have the generator prepared it is started and generates the customer specific destination system. Afterwards the installer file for the system is prepared. This can occur in wide varieties: simple zip compressing the system into an archive, being built to a WAR or EAR, or even being attached to a third party installer. For my prototype implementation I choose to implement the simple zip functionality.

The generated and prepared destination systems are stored in a customer specific temporary folder where they remain until transfer phase.

3.3.2 Transfer phase

The transfer phase includes the transfer of the prepared destination systems to the destination host. This will be mostly done via FTP in one of its variations. But of course other ways of transferring are imaginable. One way, for example, would be sending the system via email to the customer.

3.3.3 Scheduled phase

This phase reflects our need to have the actual installation process being scheduled to a given point in time and thus legitimates our choice of our update job's partitioning model two.

After being transferred to the destination host the update process enters scheduled phase. This phase correlates to timing settings given at the update jobs definition time. All jobs are chained according to their scheduled installation time. Immediately scheduled jobs are heading the chain, as the name suggests. This means, that job that are scheduled as immediately, actually only pass the scheduled phase while really scheduled jobs remain in that phase until their scheduled installation time.

These observations lead to another requirement regarding the implementation of update executing component. We have to be able to suspend the entire update process to restart it at a later time at exactly the suspended point.

3.3.4 Installation phase

As the job enters the installation phase the actual update work as we know it from simple software systems is to be done. As we deal with different kinds of destination systems we should leave execution details to an according installer component and rather concentrate on defining an interface that can be accessed and has to be implemented by the installer.

So the main task of this phase is to trigger the remote installer and to keep track of its work reflecting its state to the update job's state.

4 Implementation

4.1 High level architecture

The application is layered into 4 logical layers: business objects, data access layer, service layer and presentation layer.

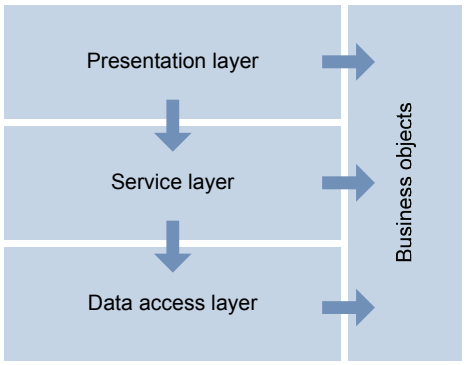

Figure 4-1 - Application layers

This model is actually a slight variant of the well known 3-tier-architecture although we leave it to logical separation not physical. This also explains why I regard the business objects layer as a separate layer although it is not horizontal.

Subsystems

The service layer itself can be subdivided into subsystems or components. I defined the subsystems / components `SystemFamilyManager`, `CustomerManager` and `UpdateManager`. The first two are basically responsible for management of business entities, storing their data in the database. The `UpdateManager` is the central access point for planning, scheduling and executing the updates, update jobs and so on. It consists of one major subcomponent, the `UpdateEngine` that takes responsibility for the update process itself.

Driving forces

Figure 4-2 shows allowed dependencies between components. The ones marked with that small lollipop expose their services via an external interface (a web GUI in our case). Other components only internally used. I tried to design subsystems with the goal to split the service layer at the borders of deployable units what will get very useful if we decide to split the application physically to. So it would be possible to deploy the calculation extensive subsystems like the `UpdateManager` on a separate machine to increase performance.

External libraries

I wrote wrapper components for 3rd party libraries, what can be regarded as implementation of the classical adapter or façade pattern mentioned in [GHJV94] or [Fow02]. This

eases the exchange of those libraries a lot decreases dependency to a concrete library to the most minimum extend possible.

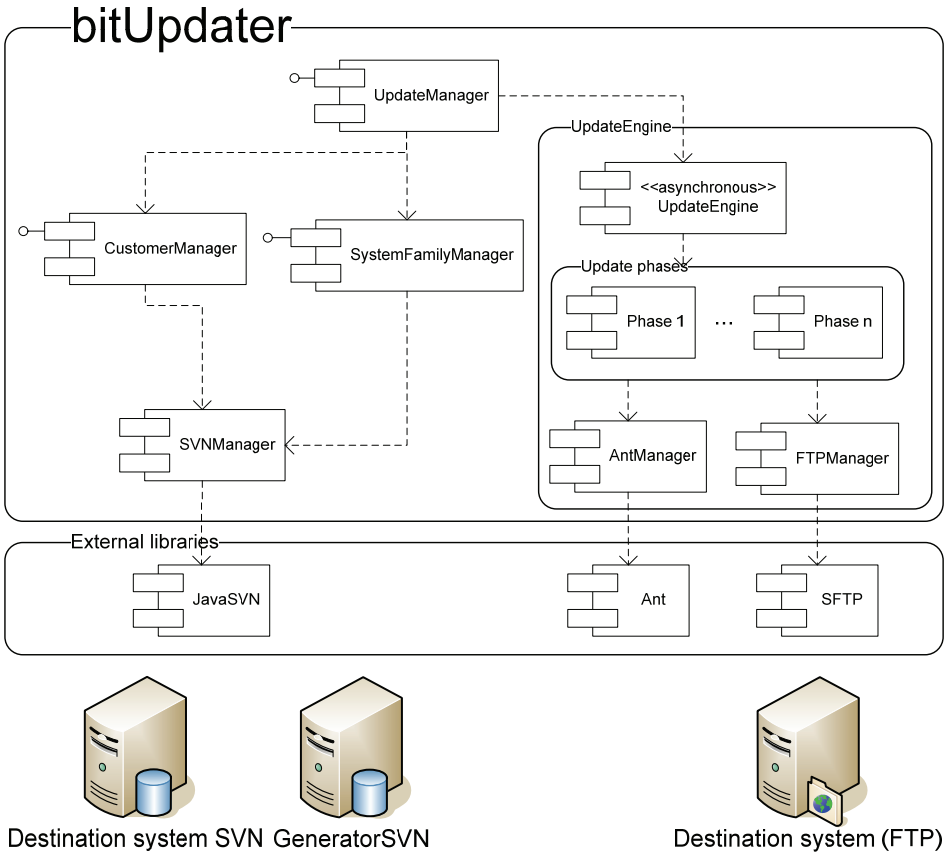

Figure 4-2 - Component dependencies

4.1.1 SonarJ

To maintain architecture decisions and efficiently enforce them I used the tool SonarJ [SonarJ]. It allows you to easily define an architecture consisting of layers (horizontal partitions) and vertical slices that can either regard all layers or single ones. Thereby you define subsystem you can define allowed dependencies for. The next step is to assign each of your application's packages to one of the subsystems.

SonarJ now checks the system for illegal dependencies, cyclic dependencies and so on. With the help of an Eclipse [Eclipse] plugin these restrictions can be enforced directly while coding. Furthermore SonarJ can calculate metrics, like average component dependencies of your software. So it a quite useful tool to help you ensuring quality of your system.

4.2 Platform selection

Due to the prototypical state of the system and upcoming changes expected, we need a flexible lightweight platform that doesn't take influence on business logic code or at least only slightly does.

As the J2EE/EJB standard should be a reliable technical basis for applications of that domain I considered it to be the technology of choice. But regarding that the EJB stack is nowadays considered as cumbersome and heavily introducing a lot of development overhead ([JH04], [Jo02], [TCLL03]) and does not allow to work with threads, I have been on the lookout for something more agile and lightweight. This is where Spring pops in.

4.2.1 Spring

Spring [Spring] is a lightweight dependency injection container, implementing the dependency inversion principle first mentioned and worked out by Martin Fowler [Fow04]. Essentially, it provides a business component programming model of independent POJOs being wired together at configuration time. Thus the components are connected from outside and do not have to instantiate dependent components themselves, which makes the components more loosely coupled. Spring also comes along with a lot of thin integration layers over widely used standards like JMS, JavaMail, JDBC and so on, that often expose a very complex and unusable API to be programmed against. So Spring offers easy integration of a lot of libraries. Providing all these features Spring emphasizes to be not invasive to your code or at least as little as possible. A good starting point to get introduced into the Spring world is [Wol06], which gives a wide overview about it, even if it does not go into detail that much.

4.2.1.1 Used Spring features

As Spring is highly separated into modules it leaves the choice of what part libraries to use to the programmer. So you easily can use only some features of spring while ignoring the ones you do not need. So here are features I used to implement the application.

Core

The core of Spring is a dependency inversion mechanism, that can be used in almost every programming situation. It means that every business component does not instantiate its dependent objects itsself (which would highly couple the components among each other). Spring provides a XML configurable factory (the so called ApplicationContext) that gets instatiated at applicatation startup time and preconfigures all your system components. This affects almost all of your programming model: first, you are encouraged to more and more program against interfaces which makes your components more loosely coupled and thus leads to a more mature system design. Second, it highly extends testability of your application regarding unit tests as well as integration tests as you can easily mock component dependecies.

Spring offers two ways to inject dependent components and thereby declaring them as required. With setter injection you offer a setter method for each of your depending components. Spring then sets them at configration time. Using constructor injection you offer a constructor taking all dependent components as arguments. This has the advantage, that Spring can only sucessfully instantiate the service object if you offer all the required components in the configuration file. This helps you to be sure to have a correct initialized service component right at configuration time. With the help of a good IDE ([MyEclipseIDE], [SpringIDE] e.g.), this can also be ensured while you are writing configration.

ORM integration

As you the take a closer look at the object relation mapping software market, you will find mostly 3 applications to choose from: iBatis, TopLink and Hibernate (excluding the direct way via JDBC or the heavyweight EntityBean mechanism of EJB). I choose Hibernate 3 because Spring once again offers easy integration of it and I wanted to get rid of crude database access operations like inserts, deletes and updates.

AOP integration

A lot of legacy application components are mostly filled with technical code that deals with integration issues of a specific platform and only concentrates little on actual business code. Most of this technical code regards transactions, repeated data modification (setting a "modified" date on business entities e.g.). Those cross cutting concerns are adressed by aspect oriented programming. Spring already offered declarative aspected oriented programming in version 1.X. Since version 2.0 I includes an offical implementation of the AspectJ pointcut language which one more time makes AOP[1] more easy and efficent.

I used the AOP modules of Spring to declare transactional behaviour of business component methods as well as touching business entities (marking their modification date) prior to saving them.

JMS

To implement asynchronism of my update engine I used the Lingo [Lingo] JmsProxyFactoryBean that acts as integration point for a message broker to make a service component message driven – entirely with declarative AOP means. No need to touch business code. Neither your calling component knows about its target method being invoked message driven nor the target object itsself. To find out more about the implementation details refer to 6.1.

WebServices

As the installers of the destination systems are exposed as web services I need to obviously need to access them via web service, too. Spring allows you to once again make this completely transprent to the application code, simply configuring it. All you need is a Java interface and an applicable WSDL file and your business component is only depending on that particular interface. All the proxy and stub creation is done by Spring.

Testing

Spring eases testing a lot. Not necessarily by a specific module – rather by encouraging you to create loosely coupled components causing a very integration of mocking frameworks like Easymock [Easymock]. Nevertheless Spring offers an extended base class for JUnit [Wes05] tests, that provide access to the ApplicationContext and automatically roll back database transactions after testing.

[1] AOP is a quite powerful programming approach supplementing traditional object oriented progamming. For more information refer to [Böh05]

Web frontend

Spring comes along with a web framework based on the MVC pattern from the gang of four [GHJV94]. It differs from most of the other well known web framworks like Struts or Tapestry, as it is equiped with the Spring core feature, letting business service components being injected into controllers.

4.3 Layers in detail

Describing functionality and implementation of each of the layers independently in detail, I'm going to work bottom up.

Each of the layers (except the business objects layer whose special role is covered in 4.3.1) is cover by a well defined interface to shield upper layers from implementation details of the underlying layer.

4.3.1 Business objects

The business objects layer is the home of our domain model classes and actually no horizontal layer as the other ones. It is rather some kind of vertical layer, because it contains the objects and data structures all the other layers use. Most of the domain model exactly refers to the defined terms and concepts above. As business objects are held persistent, there are some common properties that all the objects share due to the relational database model. These properties are addressed by the business object's super class BO.

4.3.2 Data access layer

I have implemented a dedicated data access layer which refers to the Data Access Objects Pattern mentioned in [ACM03] but differs in little details. So I avoided to implement Data Transfer Objects and use my business objects directly instead, as the DTO pattern is widely regarded as an anti pattern these days.

A common DAO super class implements most of the so called CRUD (Create, Read, Update, Delete) methods so business object specific DAOs only have to implement special data access logic.

4.3.3 Service layer

The Service layer is the main business logic layer. Here you find all the logical components and subsystems. They only refer to other components or DAO interfaces. The package structure is designed to always wrap one component in one package along with a sub package called *impl*. The main package defines externally visible artifacts like interfaces and exceptions, *impl* then accommodates the concrete implementation and is highly exchangeable. With the help of SonarJ (see chapter 4.1.1) you can restrict valid dependencies to certain packages so that a separation of this kind is quite feasible.

4.3.3.1 Update engine

As the heart of the application is the `UpdateEngine` component, I will have a more detailed look at it right now.

Primarily the process and the phases described in chapter 3 are implemented as an ordered list. As the given Java data structures do not implement all the features needed, I

implemented an `UpdatePhaseChain`. It acts as a wrapper class for an `ArrayList` enriched with some features like knowing, if a list item is the first or last one of the list, or mapping between a phase identification key and the actual phase. This chain is configured declaratively via Spring configuration mechanisms and thus instantiated at application startup time.

Concrete update phases have to extend `AbstractPhase`, which defines the execute method that triggers the phase. As we have seen in chapter 3.3, we need two different kinds of phases – some that directly lead to the next phase, and some that suspend the update job until a given point in time. One could have the idea to create two abstract subclasses ob `AbstractPhase` that either forward to the next phase or interrupt the update process until the scheduled point in time. This is mostly the same way like servlet filters are implemented. The problem is, that phases would not only decide what to do, they also would execute the decision, which would result in a scheduled phase waiting 14 days in a loop if the update job is scheduled to in 14 days.

To me it seemed more feasible to define an enumeration of return values, and let the engine decide how to proceed. This leaves the semantic definition of these return values to the engine programmer while the programmer of the phase implementation doesn't have to implicitly choose behavior with choosing a class to extend. Furthermore a concrete phase can lead to different proceeding behavior and the proceeding behavior itself is implemented at one concrete place, not being scattered through a lot of implementations.

Asynchronism / Engine state

As the update process can take a lot of time and we don't want to let the user interface wait for the process to complete, we invoke the engine asynchronously. This is done with the support of Lingo [Lingo], which helps to make a service implementation message driven. But as this of course does not decrease the length of the process itself, we need some mechanism to observe the engine's state. As you might guess, this a typical usage scenario for the Observer pattern introduced in [GHJV94] and further referenced in [BMRSS96]. I use it in a slightly modified way as I define an interface for an `UpdateEngineObserver`, let our managing component (`UpdateManager`) implement this interface and store a reference to the observer inside the engine. An observer list, as the original pattern suggests, seemed overhead to me, as it is designed to capture observers dynamically offering functionality to dynamically add and remove observers. As I consider the update engine as an internal component of the update manager subsystem I actually do not want to expose the functionality to observe the engine to other components. To provide a single access port to the engine I want to define the update manager's interface. So there is no need to have the flexible way observing the engine.

4.3.3.2 Accessing installers via web services

I decided to trigger the remote installers via web services, which introduced another field of technology. Once again Spring was a nice helper to shield the complexity of the technology from the programmer. It offers a `JaxRpcPortProxyFactoryBean` that, as well as all the other Spring factory beans, is based on the idea of dynamic proxies as you can find out in [Blo00]. The factory is actually thought to be configured in a spring configuration file. It only needs to know the URL of ad WSDL file and a local interface.

It will create the web service stubs automatically and components can use the remote service by simply letting the service being injected.

The problem in my case was that, there is no single WSDL URL I could configure. The URL depends on the given destination system and where its installer is located.

Thus the `InstallManager` component does not get an instance of the remote installer service from spring, but rather holds a partially preconfigured `JaxRpcPortProxyFactoryBean` and invokes the instantiation of the web service proxies for every different destination system.

4.3.4 Presentation layer

The presentation layer is implemented using SpringMVC (see 4.2.1.1) and Spring WebFlow (see [SpringWebFlow]) and mainly follows well know web frontend patterns like composite view e.g.

5 Prospects

Although the current implementation can be regarded as good basis for further development, I know about further requirements that sooner or later will have to be implemented.

5.1 Multi client capabilities

As we might make the system available to destination system resellers, we have to extend the user management, of course. We will have to distinguish between different clients and their employees that work with the system. Therefore we need to restrict access to certain system families, allowing the clients to only create destination systems for a configured set of system families.

5.2 Variable update process

As a result of 5.1 it will be surely necessary to extend the current update process configuration mechanism. We exclusively bind a certain process configuration to the update engine at application startup time so far. But as client's needs for various different update processes could arise, it is rather reasonable to bind the process to the client – or even better – to define a set of preconfigured update phase chains, allowing them to be accessed by clients. So we could configure that client A is allowed to start update processes X and Y, while client B is only allowed to access X. The processes will mainly differ in different implementations of certain phases. So it should be possible to configure various versions of update processes to make them available to the clients.

5.2.1 Initial generation

Until now we acted on the assumption that we have existing destination systems being registered in the update server. As we want to hand out reselling accounts to clients, it will be necessary to also manage initial generation of destination systems. Therefore the integration of a HTML form to specify the system is necessary. The customer specific configuration file has to be created and a lot of other setup steps have to be taken. As you can see, the process differs significantly from the update process specified here. So this issue could also be addressed by redesigning the update process implementation.

5.3 Exposing service layer functionality

In the current version we manage business entities and functionality via a web interface. As we're currently working a set of Eclipse plugins to ease the development of generative software systems desirable to be able to mark a concrete version of a generator as a new release version or create update jobs right from the IDE.

6 Appendix

6.1 Package description

As the application is based in the package `de.bitExpert.j2ee.bitUpdater` I am going to leave that prefix out. Almost every package includes another package called `impl`. Externally visible components like interfaces or exception are located in the main package and define the provided interface for a given component. `impl` then contains the implementation of this component.

The project is split into three source folders. `src` contains the actual application containing the business objects layer, service layer and data access layer as well as application aspects. `websrc` contains the entire presentation layer. I separated it from the core application, as it could be possible that we want to have different GUI set on top of it. So exchanging the presentation layer is easier. `testsrc` contains the test cases for the application. I separated them from the core, because they do not need to be deployed to a productive system.

Core application

`aspects`

This package contains aspect classes implementing cross cutting concerns like setting modification date of BOs before saving them or simple logging.

`business`

The `business` package is the home of all our business entity classes that is dealt with in all the system layers. Specific sorting classes can be found in the sub package `sorters`.

`conf.hibernate`

Here you find the mapping files for business objects that are needed by Hibernate to map business objects to relational database tables.

`dao`

This package contains data access components as the name suggests.

`service`

This package contains the core components like `AuthenticationManager`, `SystemFamilyManager` and `CustomerManager` that implement managing functionality of business entities as well as main exception classes of the application as they have to be handled by layers applied above the service layer. The `AbstractService` class that acts as base class for all service classes is also located in this package. It mostly only instatiates the log4j logger [Log4j]. Furthermore it acts as a wrapper package for all service layer components as the name suggests.

`service.update`

The `service.update` package contains exception classes that act as base classes for detailed update exceptions to be catched by the `UpdateManager`.

`service.update.ant`

This package contains the `AntManager` component that shields the application from the 3rd party library API exposing a application specific API. `AntRunner` is the class to start Ant [Ant]. As we need ant to start the generator to build a destination system it offers mainly only the `runBuild(DestinationSystem destinationSystem)` method.

`service.update.core`

This package is actually the heart of the application as it containes the UpdateEngine component that is responsible perform the update process described in chapter 3.3.

`service.update.ftp`

This package is the home of the `FTPManager` component that mainly acts as a wrapper for the external SFTP [SFTP] library.

`service.update.install`

The `install` package accommodates the `InstallManager` component that is responsible for invoking the remote installers via web services.

`service.update.phases`

Here we find the implementation of the update phases defined in chapter 3.3. They are invoked by the `UpdateEngine` from the `core` package.

`service.update.svn`

The svn package contains the `SVNManager` component that wraps the JavaSVN [JavaSVN] library and shields the application from its proprietary API.

Web frontend

The implementation of the web frontend is placed in the folder `websrc` followed by the `de.bitExpert.j2ee.bitUpdater.web` prefix as described above.

`actions`

The actions package contains all controller classes (meaning the same as "Action" in Struts) to be executed via the web frontend.

`actions.flow`

This package is the home for special flow actions to be invoked during the SpringWebFlow [SpringWebFlow] flows.

`editors`

Here you find classes that are responsible for mapping HTML form specific identifiers to the form backing objects. E.g. if you have a list of destination systems for a given update job, you need to convert a list of destination system ids coming from the HTML form into the appropriate list or set of destination systems.

`filter`

The filter package contains Spring specific implementations of servlet filters.

`messages`

The message package is the home of all message class implementations that are used to display user messages transporting information out of the controllers to the web page.

`tiles`

Her you find the implementation of Tiles specific user interface components as well as the controllers for them

`utils`

The utils package contains a helper class to ease accessing HTML form informations in controllers, e.g. testing which button was pressed to submit the form.

`validators`

Here you find the validator classes for our business entities

Testing

`dao.mock`

This package contains in-memory implementations of the DAO interfaces. They are rarely uses, as most of the test cases can run with real mocks. If it is necessary to compare business objects to decide if a test was successful, we need to fake storage behavior of the DAOs.

`integration`

The integration package accommodates integration test cases that refer to a specific underlying database state and access to the according Subversion repositories. A complete application context is being instantiated to set up the entire component web. The `BitUpdaterIntegrationTest` acts as base class for all the other tests and subclasses `AbstractTransactionalSpringContextTests` which provides access to the application context and automatically rolls back transactions after a test to ensure database integrity.

`integration.config`

This package contains XML configuration for testing purposes. They are subdivided into DAO configuration, service configuration and asynchrony configuration. As the files are linked in the `applicationContext.xml`, you can easily exchange certain parts of the application by just adding or replacing your own configuration to it.

`integration.phases`

Here you find the test cases for the update phases implementations.

`unit`

This package is the home of the applications unit tests. The class BitUpdaterUnitTest sets up some reference data acts as base class for all other test classes in this package and its subpackages. `unit` also contains the test suite `AllTests`, that allows running all unit test at one time.

`util`

This package contains auxiliary classes needed by the test case classes.

6.2 3rd party libraries

ActiveMQ

ActiveMQ is an open source JMS implementation, that is actually part of the Geronimo application server, but can be utilized stand alone as well. It offers a wide range of features as support of JCA, persistance with JDBC and a good Spring integration. [ActiveMQ]

JavaSVN

JavaSVN is a library to ease accessing and working with Subversion repositories. Famous Eclipse plugins like Subclipse or Subversive are based on JavaSVN. [JavaSVN]

Lingo

Lingo is a POJO based remoting and messaging library based on Spring remoting capabilities that helps you making Spring based services available to a message driven, SOA like environement [Lingo]

Quartz

Quartz is scheduling library that lets you invoke jobs repeatedly on a cronjob-like declarative way [Quartz]

SFTP

SFTP [SFTP] is a library to access FTP servers in all its different variations.

Spring WebFlow

With Spring WebFlow activity flows on websites can directly be modeled. Thereby implementation details are not relevant initially. Spring WebFlow can be integrated into Spring MVC applications as well as struts applications. [SpringWebFlow]

Tiles

With tiles, which was initially designed to be integrated into struts you can assemble a webpage of so called tiles. Thereby composing of pages of web portals is simplyfied significantly. [Tiles]

6.3 Glossary

Dependency Injection

As the name suggest DI is a mechanism that "injects" objects their dependencies at runtime. They remain dependent to them but expose their dependencies either through setter methods or constructor arguments. Furthermore the object itself is passive instead of actively creating them actively with a factory e.g. Thereby you assign arbitrary objects to the object, e.g. mocks or objects suiting a different environment.

Destination system

Is one particular instance of a system family generated with a specific configuration file.

System family

A SF is a set of software systems with a wide range of commonalities, e.g. insurance software, banking software or also office software.

Domain specific language

A DSL is a language fulfilling special requirements of a concrete domain. Thus it is tightly bound to that domain and is designed to keep technical aspects separated from the domain concepts.

6.4 Bibliography

[ACM03] D. Alur, J. Crupi, D. Marks – Core J2EE Patterns 2nd Edition (2003) – Prentice Hall, available at http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

[ActiveMQ] ActiveMQ 4.0.1 – open source message broker software – http://www.activemq.org

[Ant] Ant 1.6.5 – http://ant.apache.org

[Blo00] J. Blosser: Explore the Dynamic Proxy API (2000), available from http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy.html

[BMRSS96] F. Buschmann, R. Meunier, H.Rohnert, P. Sommerlad, M. Stal: Pattern-Oriented Software Architecture (1996) – John Wiley & Sons

[Böh05] Oliver Böhm: Aspektorientierte Programmierung mit AspectJ (2005) – dpunkt.verlag

[CE00]	K. Czarnecki, U. Eisenecker: Generative Programming – Methods, Tools, and Applications (2000) – Addison-Wesley
[Easymock]	Easymock 2.2 – http://www.easymock.org
[Fow02]	Martin Fowler et al: Patterns of Enterprise Application Architecture (2002) – Addison Wesley
[Fow04]	Martin Fowler: Inversion of control, available from http://martinfowler.com/articles/injection.html
[Eclipse]	http://www.eclipse.org
[GHJV94]	E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software (1994) – Addison-Wesley
[Hibernate]	Hibernate 3 – http://www.hibernate.org
[Log4j]	Log4j 1.2.13 - http://logging.apache.org/log4j/docs/
[JavaSVN]	JavaSVN 1.0.6 – http://tmate.org/svn
[JH04]	R. Johnson, J. Höller: J2EE Development without EJB (2004) – Wrox
[Jo02]	R. Johnson: Expert One-to-One J2EE Design and Development (2002) - Wrox
[Lingo]	Lingo 1.2.1 – http://lingo.codehaus.org
[MyEclipseIDE]	MyEclipseIDE 4 – http://www.myeclipseide.org
[Quartz]	Quartz 1.5.2 – http://www.opensymphony.com/quartz
[SCK+96]	M. Simons, D. Creps, C. Klinger, L. Levine and D. Allemang: Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14th, 1996, available from http://domain-modeling.com
[SFTP]	Secure FTP Factory 6.0 – http://www.jscape.com/sftp/
[SonarJ]	SonarJ – http://www.hello2morrow.de
[Spring]	Spring 2.0 – http://www.springframework.org
[SpringIDE]	SpringIDE – http://www.springide.org
[SpringWebFlow]	Spring WebFlow – http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/Home
[TCLL03]	B. Tate, M. Clark, B. Lee, P. Linskey: Bitter EJB (2003) - Manning

[Tiles] Tiles – http://struts.apache.org/userGuide/dev_tiles.html

[Tomcat] Tomcat 5.5 – Java web server – http://tomcat.apache.org

[Wes05] F. Westphal: Testgetriebene Entwicklung mit JUnit & Fit (2005) – dpunkt.verlag

[WHE03] T. Wollny, M. Henss, U. Eisenecker – Erstellung eines Generators für eine E-Learning Softwaresystemfamilie (2003), available at http://gp.informatik.fh-kl.de/downloads/doku/Praxissemesterbericht_Wollny_Henss.pdf

[Wol06] E. Wolff: Spring – Framework für die Java-Entwicklung (2006) – dpunkt.verlag

All internet resources were checked in September 2006.

7 Index

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, der 18.09.2006 .....

Unterschrift