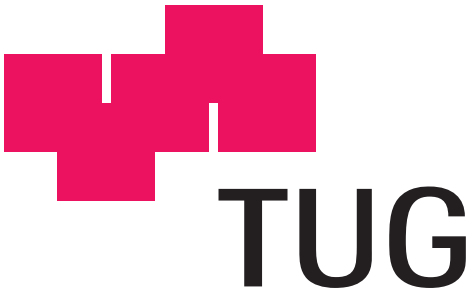
Thomas Truskaller

Master Thesis

Data Integration into a Gene Expression Database



Institute of Biomedical Engineering,
University of Technology, Graz, Austria
Inffeldgasse 18, A-8010 Graz
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Gert Pfurtscheller

Supervisor:
Dipl.-Ing Gerhard Thallinger

Evaluator:
Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Graz, September 2003

Abstract

German

Das Ziel dieser Diplomarbeit war die Untersuchung und Entwicklung einer neuen Form von "Integrativer Software Entwicklung" für daten- und webbasierte Anwendungen in der Bioinformatik. Im Besonderen wurde auf die Vereinfachung des Entwicklungsprozesses und Abstraktion durch Softwaremodellierung basierend auf der Unified Modeling Language (UML) eingegangen.

Es wurde der Open Source Code Generator AndroMDA in Bezug auf Erweiterbarkeit und Anpassbarkeit seiner Grundfunktionen und Codegenerierungsvorschriften auf datenbasierte Webanwendungen, wie sie vor allem in der Bioinformatik häufig vorkommen, untersucht.

Es hat sich herausgestellt, dass die Verwendung von geeigneten Werkzeugen und Techniken, wie UML CASE Tools, Code Generatoren und MDA, das Umsetzen von generellen Anforderungen an die Software Entwicklung wie z.B. Wartbarkeit, Wiederverwendbarkeit, Robustheit und Konsistenz erleichtern.

Der Schritt zu "Generativen Software Enwicklungsmethoden" erfordert Anfangs einen höheren Einarbeitungs- und Lernaufwand, liefert aber mit Fortdauer des Projektes wesentliche Vorteile.

Stichwörter: MDA, J2EE, Generative Software Entwicklung, AndroMDA

English

The goal of this thesis was to investigate and develop a new form of ”Integrative Software Development” for data driven and web based applications in the bioinformatics domain. The simplification of the development process and abstraction with UML based software modelling techniques were the main point of interest.

One of the main tasks was the investigation of the open source code generator AndroMDA in terms of its ability for adoption and extensibility of the core functionality and its code generation templates for data driven web based applications.

Facilitating proper tools and technologies like UML CASE tools, code generators and MDA, simplifies the development of software and increases its general requirements like maintainability, reusability, robustness and consistency.

The learning effort when using ”generative” software development methods is higher at the beginning, but this pays off through faster application development and better maintainability.

Keywords: MDA, J2EE, Generative Software Entwicklung, AndroMDA

Contents

List of Figures

List of Tables

Glossary

CASE tools Computer Aided Software Engineering tools for modelling software systems based on the UML language.

CIM A computation independent model in MDA is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification.

CORBA CORBA is the acronym for Common Object Request Broker Architecture, OMGs open, vendor-independent architecture and infrastructure that computer applications can use to work together over networks.

CRUD The acronym for create, read, update and delete.

DNA Deoxyribonucleic acid (DNA) is a double-stranded helix of nucleotides which carries the genetic information of a cell.

DTD Document Type Definition. An optional part of an XML document which specifies constraints on the valid tags and tag sequences that can be in the XML document.

EJB An Enterprise Java Bean is a server-side component of the J2EE platform that encapsulates the business logic of an application.

framework APIs that are intended to simplify the design and coding process.

Hibernate Hibernate is a object/relational persistence and query service for Java.

IDL Interface Definition Language is the OMGs standard language for defining the interfaces for all CORBA objects.

Java Object oriented, high-level programming language and platform developed from Sun.

JDBC Java database connectivity. This set of application programming interfaces (APIs) provides a standard mechanism to allow Java applications access a database.

JDO Java Data Objects is a specification to enable transparent persistence of Java objects.

JNDI Java Naming and Directory Interface specification enables Java platform-based applications to access multiple naming and directory services.

JSF JavaServer Faces, a proposal to define an architecture and APIs that simplify the process of building J2EE web tier applications.

JSP Java Server Pages is a language similar to Microsoft's ASP for creating dynamic web pages. Sun has created a language specification. With JSP, a single source text file is created containing both HTML (or XML) tags and Java-like scriptlets. The JSP-aware web server creates, compiles and runs a servlet from the source text.

JSTL JavaServer Pages Standard Tag Library. A standard tag library that encapsulates core features common to many JSP pages as simple tags. JSTL contains support for common, structural tasks. Support includes iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags.

J2EE Java 2 Platform, Enterprise Edition is a set of coordinated specifications and practices that together enable solutions for developing, deploying, and managing multi-tier server-centric applications.

MARS Microarray Analysis and Retrieval System is used for storing, processing and organising microarray data based on Java technologies (EJB, JSP, Servlets etc.) and Open Source frameworks (Struts etc.). It is developed by the TU-Graz Bioinformatics Group.

MDA Model Driven Architecture is an OMG standard with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MGED The Microarray Gene Expression Data Society is an international organisation of biologists, computer scientists, and data analysts that

aims to facilitate the sharing of microarray data generated by functional genomics and proteomics experiments.

Metadata A general term for data that in some sense describes information (data).

Microarray Sometimes called a gene chip or a DNA chip. Microarrays consist of large numbers of molecules (often, but not always, DNA) distributed in a grid in a very small space. Microarrays permit scientists to study gene expression by providing a snapshot of all the genes that are active in a cell at a particular time.

Model A model is generally usually used to denote a description of something, typically something in the real world.

MOF The Meta-Object Facility specification defines an abstract language and a framework for specifying, constructing and managing technology neutral metamodels.

MVC Model / View / Controller. A user-interface design pattern pioneered by Smalltalk and later co-opted by Java et. al. The Model is typically a data-bearing business object, the View is an object capable of presenting some form of visual output from the Model, and the Controller is responsible for processing user events.

NCBI The National Center for Biotechnology Information

.NET At its core, Microsoft .NET is about integrating services. To make use of .NET is in essence to connect information using the Microsoft .NET family of software technologies. The technologies break down into three chunks, the .NET Framework, .NET Web Services, and ASP.NET.

OMG The Object Management Group is an open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

ORM tool Object Relational Mapping tool, storing objects in relational databases.

PIM A Platform Independent Model is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.

Platform A Platform in MDA is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

PSM A Platform Specific Model in MDA is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

RDBMS Relational database and management system is a collection of programs that enable multiple users within a system to store, modify, and extract information from a database based on relational tables.

RNA Ribonucleic acid. One of the two types of nucleic acids found in all cells. The other is deoxyribonucleic acid (DNA). RNA transmits genetic information from DNA to proteins produced by the cell. Three types of RNA include: messenger RNA (mRNA), transfer RNA (tRNA) and ribosomal RNA (rRNA).

UML OMGs Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.

Web Services Web services are units of application logic with published API's that provide data and services to other applications. They leverage industry-standard communication formats such as XML and SOAP. Web services can be called across platforms and operating systems, and without regard to the underlying programming language.

XMI The XMI Specification defines technology mappings from MOF metamodels to XML Document Type Definitions (DTDs) and XML documents.

XML Extensible Markup Language is an open standard of the World Wide Web Consortium (W3C) designed as a data format for structured document interchange on the web.

XSLT Extensible Style Sheet Language Transformation is a W3C-specified language for transforming XML documents into other XML documents.

Chapter 1

Introduction

This chapter highlights the problems when handling large biological data sets from the programmers point of view. A simple example will show the difficulties in manipulating new kind of data and adopting existing databases. Projects often start with a few and simple requirements, but likely evolve into big heterogenic and complex enterprise applications.

1.1 A Simple Example

At the Institute of Biomedical Engineering - Graz University of Technology the Bioinformatics Group [5] is working on microarray experiments. The group consists of biologists, chemists and software developers. For storing the microarray data a database system called MARS (Microarray Analysis and Retrieval System) was developed based on Java [11] technology.

By and by the additional requirements emerged and the existing code had to be reverse engineered. This is a very tedious and error prone work for programmers because they have to make changes on existing codes and find out if these modifications didn't corrupt the application. New programmers joining the team have to learn the architecture design and coding standards.

The existing MARS system provides basic functionality for storing microarray specific data. Requirement changes should cause minor coding efforts and new modules should be added in a "Plugin" manner.

Potential additional requirements for the MARS system could include:

- new user management

- integration into clinical workflows
- analysis tools for quality control
- integration of laboratory management system
- integration of metabolic data

The work with microarrays is still research work and it is very likely that there will be changes in the process. New technologies and methods will come up and improve appearance of the experiments. A key role for reproducible and reliable results is a broad quality assessment.

1.2 Problem Domain

1.2.1 Biological Data

The human genome consists of about 3 billion bases where each can be A, G, T or C. Storing this information would require about 3000 Mega Byte of disc space. Scientists estimate a number of about 35000 - 45000 genes in the human genome [28].

Simply spoken: Each gene is a sequence of bases which is the coding instruction for a type of protein. The proteins are key players in the functionality and structure of organisms. The numbers of genes, proteins and other microbiological components show the complexity when dealing with biological data. Without computer systems it would be nearly impossible to process all the biological data flood.

Bioinformatics is an inherently integrative discipline, requiring access to data from a wide range of sources [27]. "The NCBI Handbook" [22] gives an insight into the basic data-storages, -flow, -processing, -querying and -linking tools on the biological domain and is recommend for further readings.

1.2.2 Gene Expression and Microarray Data Management

Microarray technology was introduced in the year 1995 by Schena et al. [34]. To understand microarray experiments technology one have to understand the central dogma of molecular genetics. It describes the way from

deoxyribonucleic acid (DNA) to messenger ribonucleic acid (mRNA) and from mRNA to proteins shown in figure 1.1.

Central dogma of molecular genetics

DNA is a double-stranded helix of nucleotides which carries the genetic information of a cell. It encodes the information for the proteins and is able to duplicate itself to be passed on to the next generation of cells (replication).

mRNA (messenger RNA) is the mediating template between DNA and proteins. The information from a particular gene is transferred from a strand of DNA by the construction of a complementary strand of RNA through a process known as transcription. Three nucleotide segments of RNA, called transfer RNA (tRNA), which are attached to specific amino acids, match up with the template strand of mRNA to order the amino acids correctly. These amino acids are then bonded together to form a protein. This process, called translation occurs in the ribosome, which is composed of proteins and the third kind of RNA, ribosomal RNA (rRNA). The so called "central dogma of molecular genetics" was introduced by Francis Crick 1970 [29].

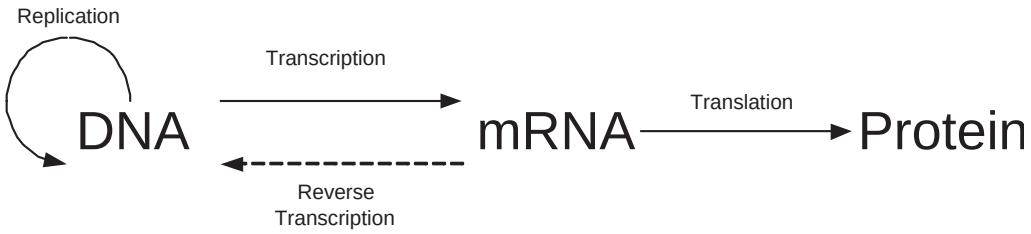


Figure 1.1: The central dogma of genetics.

Gene Expression and Microarray

Gene expression is the activity of a gene in a cell at a certain time point. If the gene is active then the corresponding mRNA is transcribed and the protein translated. In microarray experiments the concentration of the mRNA in the cell is used for measuring the activity of a gene. A microarray experiment can show the gene expression of several thousands of genes (up to 30000 and more genes) at a certain time point on a single glass slide.

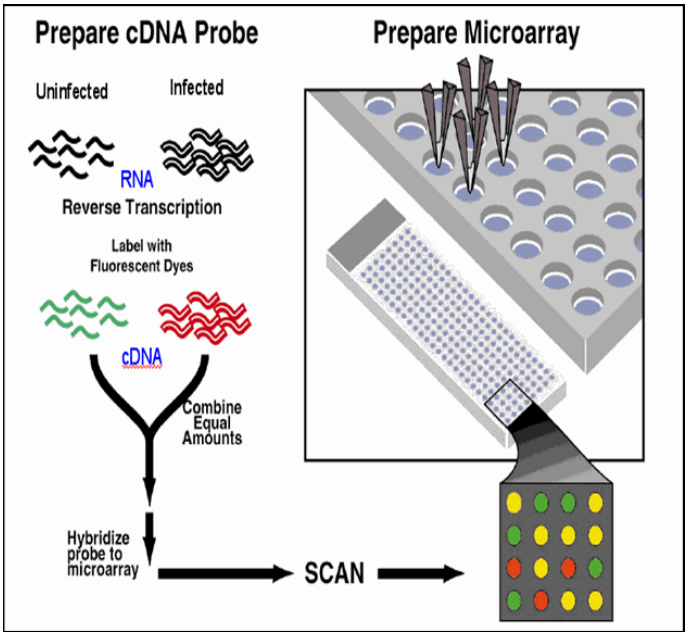


Figure 1.2: Microarray experiment process.

Figure 1.2 shows the generation of a microarray experiment in a simplified form. The DNA microarray experiment process can be divided into two branches: one preparing the DNA chip (microarray) using the chosen target DNAs and the other preparing a hybridization solution containing a mixture of fluorescently labelled cDNAs. The branches merge when the chips are hybridised, where the probes are bound to the samples on the carrier. Bound cDNA is detected using laser technology and the data are stored on a computer system where they can be used for further analysis.

With microarray technology one can obtain a snapshot of the activity for nearly all genes in a cell for a certain time point. It is possible to process the data with computational methods and analyse the whole bunch of genes for specific profiles without focusing on one specific gene.

Standards for microarray experiments are administered at the MGED [26] site. Storing microarray data can be done in various techniques: in XML files, Relational Database Management Systems (RDBMS) databases and more. The web resource [1] gives a listing of microarray databases on the web.

1.2.3 Laboratory Management

The work with microarray experiments is tightly coupled to laboratory work. Using a quality management (QM) system is an essential requirement for reliable and usable data. Additional QM technologies will emerge which have to be integrated into the existing MARS system. To reduce coding and programming work an efficient way of programming and data integration has to be developed.

1.2.4 Enterprise Application

The demand for integration of biological data and experiments with clinical research and production is increasing. Applications are facing an increase in information and data to be managed. The steady expansion of applications is not exclusively limited on biological domain although it is one of the most growing ones. The solutions developed during this work are applicable for any kind of software projects.

1.2.5 General Software Requirements

Software development is a mature engineering discipline and general requirements to be followed were found out long times ago. High quality software has to fulfill the premises listed in table 1.1:

Requirements	**Meaning**
Robustness	code must have a defined behavior under defined circumstances
Consistent code	code has to be consistent and follow predefined *Style Guidelines*
Simple code	code should follow the Keep It Small and Simple (KISS) philosophy
Modularity	use interface definitions for separation
Flexibility	design reusable components applicable for similar requirements
Portability	important is the choice of the Integrated Development Environment (IDE) and programming language
Intuitive code	code must be readable like a book (use representative names for classes, variables and methods)
Performance	separate subjective and objective performance
Maintainability	given when all the other requirements are fulfilled
Extensibility	given when all the other requirements are fulfilled

Table 1.1: General Software Requirements [32].

Chapter 2

Objectives

The goals for the integration framework were derived from the collected experience on several software projects implemented on the Bioinformatics Group Graz [5]. Most applications developed are multi layered, web based and written in the Java programming language.

2.1 Project Goals

The following project goals were defined before starting working on the implementation framework.

- To implement a framework for easy integration of data into a gene expression database. Since the type of data collected during the process of biological experiments and analysis is not known, a dynamic and flexible solution for adding the different kinds of data should be built. This data could be metabolic data, gel images for quality control or other biological experiments.
- To find a mechanism for easier and faster programming of data driven, multilayered and web based applications, where also general software requirements are taken into account.

Chapter 3

Methods

This chapter will give a brief introduction of the technologies used in the integration framework, references and links for further reading are included in each topic.

3.1 The OMG's Key Standards

The Object Management Group (OMG) [18] was formed to help reduce complexity, lower costs, and speed up the introduction of new software applications. The OMG accomplishes this goal through the introduction of the Model Driven Architecture (MDA) architectural framework and its detailed specifications. These specifications should lead the industry towards interoperable, reusable, portable software components and data models based on standard models [30].

OMG has adopted a number of technologies, which combined enable the model-driven approach. These include Unified Modelling Language (UML), XML Metadata Interchange (XMI) Format, Meta Objects Facility (MOF), specific MOF models as well as UML profiles. All OMG specifications are available on the web [6].

3.1.1 Model Driven Architecture

The MDA is a new way of defining specifications and developing applications, based on a platform-independent model (PIM)[1]. A complete MDA specification consists of a distinct platform-independent UML base model, plus one or more platform-specific models (PSM) and interface definition sets, each describing how the base model is implemented on a specific middleware platform.

The MDA aims to separate business or application logic from the underlying platform technology. Platform-independent applications built using MDA and associated standards can be implemented using a wide range of open and proprietary platforms including Common Object Request Broker Architecture (CORBA), J2EE, .NET, Web Services or other web-based platforms.

It is not necessary to repeat the process of modelling an application or system's functionality and behavior each time a new technology (e.g. .NET, EJB 2.0) comes along. With MDA function and behavior are modeled only once. Mapping from a PIM through a PSM to the supported MDA platforms will be implemented by tools, easing the task of supporting new or different technologies.

The MDA provides an approach for and enables tools to be provided for:

- specifying a system independently of the platform that supports it
- specifying platforms
- choosing a particular platform for the system
- transforming the system specification into one for a particular platform

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. MDA separates implementation details from business functions.

[1] A platform in general is a set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns. Any subsystem that depends on the platform can use these technologies without concern for the details of how the functionality provided by the platform is implemented.

3.1.2 Unified Modelling Language

Modelling is the design of software applications before coding and an essential part of large software projects, helpful for medium and even small projects as well [20].

The OMG's UML [19] specification defines a graphical language to specify, visualize, and document models of software systems, including their structure and design. (UML can be used for business modelling and modelling of other non-software systems too.) Using any of the large number of UML-based tools on the market[2], future applications requirements can be analysed and designed representing the results using UML's standard diagram types. Models used with MDA can be expressed using the UML language.

The UML Specification includes the following:

- a formal definition of the UML metamodel; that is, the abstract language for specifying UML models
- a (non-normative) graphic notation for expressing UML models
- a set of CORBA Interface Definition Language (IDL) interfaces for representing and managing UML models
- a XML Metadata Interchange (XMI) format for UML model interchange.

The graphic notation defines twelve types of diagrams with its model elements, divided into three categories shown in table 3.1:

Diagram Category	Diagram Name
Structural	Class Diagram, Object Diagram, Component Diagram, Deployment Diagram
Behavior	Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram, Statechart Diagram
Model Management	Packages, Subsystems, Models

Table 3.1: UML diagram types

[2] A list of UML tools can be found on http://www.jeckle.de/umltools.htm (from September 10th, 2003)

3.1.3 Meta Objects Facility

The Meta-Object Facility (MOF) specification defines an abstract language and a framework for specifying, constructing and managing technology neutral metamodels. A metamodel is in effect an abstract language for some kind of metadata. Examples include the metamodels for UML, Common Warehouse Meta-Model (CWM) and the MOF itself as well as others in various OMG specifications in progress.

In addition, the MOF defines a framework for implementing repositories that hold metadata (e.g. models) described by the metamodels.

The MOF specification includes the following:

- a formal definition of the MOF meta-metamodel; that is, the abstract language for specifying MOF metamodels
- a mapping from arbitrary MOF metamodels to CORBA IDL that produces IDL interfaces for managing any kind of metadata
- a set of ”reflective” CORBA IDL interfaces for managing metadata independent of the metamodel
- a set of CORBA IDL interfaces for representing and managing MOF metamodels
- an XMI format for MOF metamodel interchange

3.1.4 XML Metadata Interchange Format

The objective of XML Metadata Interchange (XMI) is to allow the exchange of objects from the OMG's Object Analysis and Design Facility. It is based on the XML (Extensible Markup Language) standard.

The XMI Specification defines technology mappings from MOF metamodels to XML Document Type Definitions (DTDs) and XML documents. These mappings can be used to define an interchange format for metadata conforming to a given MOF metamodel.

Table 3.2 shows the four layered architecture of the OMG metadata architecture:

Meta-level	MOF term	Example	DTD/XMI
M3	meta-metamodel	The "MOF Model" (an abstract syntax for defining metamodels)	MOF DTD
M2	meta-metadata, metamodel	UML Metamodel, CWMI Metamodel, etc.	UML DTD
M1	metadata, model	UML Models, Warehouse Schemas, etc.	UML models as XMI documents
M0	data	modeled systems, Warehouse databases, etc.	

Table 3.2: The MOF metadata framework is typically depicted as a four layer architecture

There are 2 versions of the XMI specification, as follows:

- **Version 1.2:** Defines the main purpose of XMI which is to enable easy interchange of metadata between modelling tools (based on the OMG-UML) and metadata repositories (OMG-MOF) in distributed heterogeneous environments.
- **Version 2.0:** Defines XMI Version 2 which leverages new features in XML Schemas that are not available in DTDs.

The main difference from version 1.2 to 2.0 is the use of XML Schemas, a more comprehensive form of XML document validation, instead of XML DTDs.

3.2 Code Generation

A generator is a program that takes a higher level specification, describing the problem domain, and configuration (transformation) information, to produce a specific solution (implementation). The generative domain model [31] puts this into a schema shown in figure 3.1.

The automation of software production speeds up software development and reduces development costs, while also improving quality, reducing errors, requiring less maintenance efforts and consequently leading to lower maintenance costs.

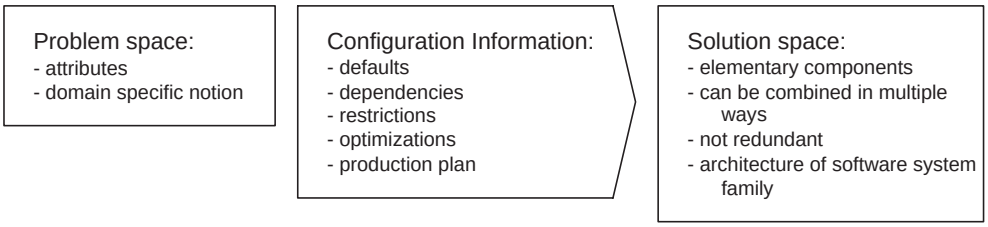


Figure 3.1: The Generative Domain Model consists of the Problem Space, the Configuration Information and the Solution Space [35].

3.2.1 Types of Code Generators

So called "Closed" generators on one side provide fixed - but optimised - solutions, without the possibility of modification and adoption in the problem and configuration space of the generative domain model. The specification (problem domain) is defined in a proprietary, textual or graphical language. Examples for this kind of generators are CORBA-IDL and EJB compiler. "Open" generators on the other side have no restrictions in the generated code and its structure. The developers are provided with the basic functionality of code generation with the possibility to define the meta model and configuration (transformation) rules for their requirements. Examples are Template-Based Systems and Frame Processors as well as Abstract Syntax Tree based (AST) Application Programming Interface (API) [35]. There are systems comprising both types mentioned above. Most of the time they have a fixed modelling language and free definable generation rules [36].

Code generators on the market have varying characteristics in the way how to adopt the generator and the generation rules for the needs of the developers. Generation rules (often called "cartridges") define the transformation from the domain model to the implementation code.

Cartridges hold information about the specific technical solution and should have a hierarchical structure for reusability. Hand written cartridges are difficult to maintain, they also lack a way of visualization and reusability. A better solution is to describe the generation rules in a formal model and generate the cartridges (generation of generation rules). In this way the benefits of the MDA (Model Driven Architecture) can be ported to the development of code generators. *ArcStyler* is a model based code generator provided by *Interactive Objects*[3]. Its code generation rules are hierarchical

[3] http://www.io-software.com/ (from September 10th, 2003)

structured and itself generated from UML models [25] for easier maintenance.

3.2.2 UML Based Code Generators

UML (see sec. 3.1.2) provides a graphical language for describing software. With XMI (see sec. 3.1.4) you can serialize and transfer the UML models among different UML tools. An UML based code generator ideally reads the UML model from a model repository (XMI file, direct from memory etc.) and generates platform specific code out of it. Many CASE tools provide code generators which are more or less "Closed" generators. Very often they have proprietary features and only interact with models of one CASE tool instead of the XMI standard. One UML based code generator is AndroMDA, which will be described in more detail in section 3.3.

3.3 AndroMDA

The Open Source project AndroMDA [2] is a template based code generator framework written in Java - it takes an UML model from a CASE-tool and generates custom components based on Velocity [23] templates. The project provides tools for a model based and generative software development following the MDA-paradigm. The goal of AndroMDA is to develop a toolbox for complete MDA based development.

Figure 3.2 gives an overview of the development cycle in AndroMDA. On the input side it takes an UML model and builds a corresponding metamodel using the *Metadata Repository* (MDR), an Open Source project of "Netbeans" [17]. Currently only the MOF/UML 1.4 in MOF/XMI 1.2 is supported.

The instantiated objects of the model are sent to Velocity, an Open Source scripting engine which can generate any text document from the model depending on the generation rules (cartridges).

AndroMDA evolved from the UML2EJB project and cartridges for EJB (Enterprise Java Bean) components exist. Cartridges for the O/R mapping tool Hibernate [9] as an alternative persistence technology and cartridges for the Struts [21] framework for web applications were developed in the last period of the project. It is up to the developers to write their own cartridges or modify the existing ones. There might be cartridges for .NET, CORBA

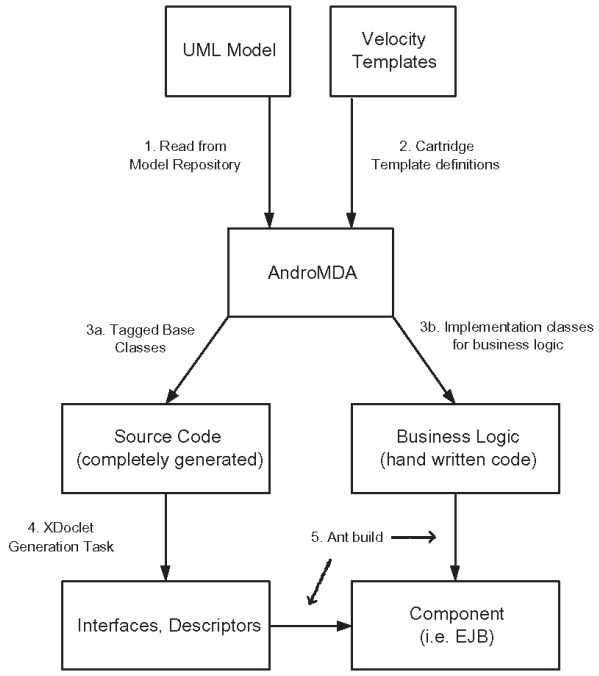


Figure 3.2: AndroMDA code generation process [33].

and any other technology as well.

The EJB and Hibernate cartridges generate java source code tagged with additional information readable for the XDoclet [24] code generator framework. XDoclet processes the source code, generates interface and deployment descriptors and completes the code generation process. An introduction to XDoclet will be given in section 3.5.

Figure 3.3 shows the roles participating in the development process with AndroMDA. The architect is responsible for architecture- and template design and must be a specialist on the implementation technology. The domain modeler is a specialist for the business domain and models its processes in a PIM model. The developer adds supplementary information to the model (i.e. tagged values) and provides the model for the code generator. After the code generation the developer has to code business logic into the implementation classes. The deploy process is supported by Ant build tasks. At the end of the process the developer has to test the components. An introduction into AndroMDA and its main features can be found in [33].

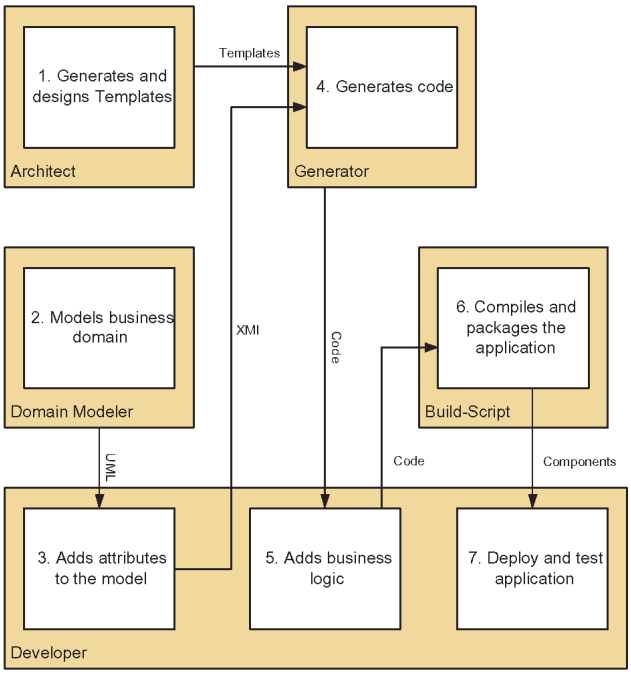


Figure 3.3: Participants in the development process with AndroMDA [33].

3.3.1 Modelling in the AndroMDA

AndroMDA is a tool with a medium amount of intelligence built into it. Such a tool always has to rely upon clearly defined preconditions. In the case of AndroMDA these preconditions are modelling conventions. A UML model for use with AndroMDA cannot be structured arbitrarily, it has to adhere to a certain structure that can be interpreted by AndroMDA cartridges.

Not too many constraints should be imposed on the modeler - after all, a model should represent the conceptual reality of the customer's requirements and the structure of the application domain.

AndroMDA uses stereotypes[4] on different kinds of model elements, each with its own special meaning or intention shown in table 3.3:

[4]UML Stereotypes are a kind of "labels" that you attach to modelling elements to classify them.

Stereotype	Model element	Meaning
Entity	class	business object or domain object that knows much and does little
Service	class	dynamic element or activity that does something to the entities
PrimaryKey	attribute of an entity class	mark attribute as primary key so that the entity can be found in a database system
FinderMethod	operation of an entity class	associate this method with a database query to find this entity
EntityRef ServiceRef	dependency between class A and class B	make A depend on B when a cartridge needs this info
Exception	dependency between class A and class B	make B an exception thrown by all business methods of A
WebAction	class	Web Action (controller) class
WebForm	class	Web form (model) class
WebJSP	class	JavaServer (view) Page

Table 3.3: Stereotypes in AndroMDA.

3.3.2 Exception Handling in AndroMDA

UML dependencies with the stereotype $<<Exception>>$ model the exception handling in AndroMDAs default cartridges. They point from $<<Entity>>$ or $<<Service>>$ components to an exception classes. This will cause AndroMDA to generate "throws" clauses which contain this exception class in the methods of the components. AndroMDA does not generate exception classes, they have to be written manually. See figure 3.4 for an example.

3.3.3 Cartridges

Cartridges were introduced to separate the development of the core generator framework and the code generation templates. A cartridge can be seen as an blueprint of a specific technology (e.g. EJBs, .NET, CORBA, Struts, Java, C# and many more). They plug into the AndroMDA core and drive code

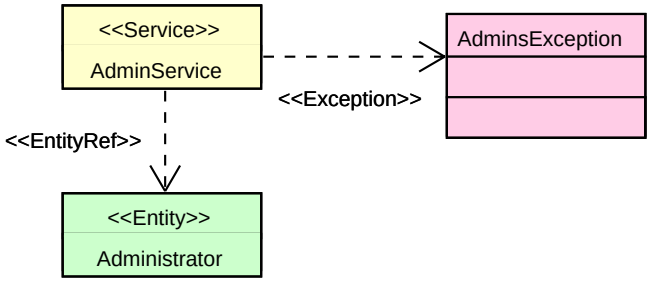


Figure 3.4: Exception handling in the default cartridges of AndroMDA [3].

generation based on templates.

Cartridges in AndroMDA are `jar` files, `zip` files or directories. They contain the Velocity templates, Java classes (needed by the specific cartridge) and a META-INF subdirectory containing a descriptor file named `andromda-cartridge.xml`. AndroMDA searches for the `andromda-cartridge.xml` file on the "classpath" to locate the available cartridges automatically. The core uses this descriptor to find out the capabilities of a cartridge: the supported stereotypes, the outlets (see sec. 3.3.5) and the Velocity templates. An `andromda-cartridge.xml` file could look like listing 3.1:

```
<cartridge name="ejb">
  <property name="persistence" value="ejb" />
  <property ... />
  <stereotype name="Entity" />
  <stereotype ... />
  <outlet name="entity-beans" />
  <outlet ... />
  <template
    stereotype="Entity"
    sheet="templates/EntityBean.vsl"
    outputPattern="{0}/{1}Bean.java"
    outlet="entity-beans"
    overWrite="true" />
  <template ... />
</cartridge>
```

Listing 3.1: AndroMDA cartridge definition example.

Table 5.1 in the appendix shows the XML elements with description used in the `andromda-cartridge.xml`. The attributes of the `<template>` tag are shown in table 5.2 in the appendix.

AndroMDA delivers cartridges for EJBs, Hibernate and Struts. The EJB cartridge generates Session Beans for services and Entity Beans for persistence. The Hibernate cartridge generates plain old java objects and utilises the OR mapping framework Hibernate for persistence. Table 3.4 lists the artefacts[5] generated by the EJB and Hibernate cartridges.

The struts cartridge generates Struts-actions, -forms, JSPs and the `struts-config.xml` file. Modelling the struts framework in a class diagram is rather tedious, therefore another solution facilitating activity graphs in the UML model is under development[6].

Stereotype	**EJB**	**Hibernate**
Entity	Entity Bean	Java object with persistence support plus factory class
Service	Session Bean	Java object
PrimaryKey	marks attribute as ”PK”	marks attribute as ”id”
FinderMethod	finder method on home interface	finder method in associated factory
EntityRef	ejbref from ABean to BBean and getBLocalHome() in ABean	nothing
ServiceRef	ejbref from ABean to BBean getBHome() in ABean	
Exception	”throws B” in all business methods of A	

Table 3.4: Generated artefacts in EJB and Hibernate cartridges.

3.3.4 Velocity Templates

Velocity is a Java-based template engine. It permits anyone to use the simple yet powerful template language to reference Java defined objects [23].

[5] In this term an artefact is a result or product of a code generation process.
[6] Announced on the AndroMDA mailing list

AndroMDA uses velocity templates for the code generation process. In the `andromda-cartridge.xml` the mapping from stereotype to Velocity template is defined. The outlet attribute defines the destination for the generated source.

The cartridge in listing 3.1 would cause AndroMDA to invoke the velocity template `EntityBean.vsl` for each class with the stereotype $<<Entity>>$ in the UML model. AndroMDA passes the objects in table 3.5 to the templates. Through the scripting variables the template developer has access to the methods defined in the Java objects. See the Velocity home page for further readings [23].

Object name	Description	More info
`$model`	Contains the model that was built in the case tool, in the form a UML v1.4 model.	Model
`$class`	Provides a reference to the UML model element for which code should be generated.	In general that object will be of type `PModelElement` and in the case of a class it will be of type `PClassifier`
`$transform`	Helper object that can transform model objects into a printable form that can be used easily from a VTL script.	`SimpleOOHelper`
`$str`	Helper object that can perform string formatting.	`StringUtilsHelper`
`$date`	The current date.	`java.util.Date`

Table 3.5: Objects which AndroMDA passes to the Velocity templates [2].

The template in listing 3.2 just prints the name and the fully qualified name of the class into the output file shown in listing 3.3.

3.3.5 Outlets

Outlets provide a logical name used in the cartridges and the Ant build process. The cartridges define an outlet and map templates to outlets (see table

```
Simple Velocity template!
-------------------------
class name = ${class.name}
fqn = $transform.getFullyQualifiedName($class)
```

Listing 3.2: Simple Velocity template example.

```
Simple Velocity template!
-------------------------
class name = EntityX
fqn = some.package.EntityX
```

Listing 3.3: Output of the example in 3.2.

5.2). The Ant `build.xml` file maps the outlets to a physical directory. During the generation process the code will be stored in that directory. The listing 3.4 is a snip of a `build.xml` file using the cartridge definition shown in listing 3.1.

3.3.6 AndroMDA Ant Task

Currently AndroMDA is implemented as a custom task for the build tool Ant [4]. Ant is a Java-based build tool and therefor platform independent. It was invented to simplify the build process without the huffiness of the "GNU Make" tool.

Ant **tasks** perform the actual work. Besides the core tasks for low level work like copying, creating directories, zipping etc., optional tasks for more complex work exist. Developers also have the possibility to define their own tasks by extending the Ant framework.

```
<andromda ... >
  ...
  <outlet cartridge="ejb"
          outlet="entity-beans"
          dir="/home/myproject/src/java/ejb/gen/" />
  ...
</andromda>
```

Listing 3.4: Outlet definition example in the AndroMDA ant task.

The whole build process is based on an XML file (often named `build.xml`). There the tasks are defined based on a hierarchical structure. AndroMDA provides a custom task for executing the generator. The properties are listed in table 5.3 in the appendix.

Listing 3.5 shows a snip of an AndroMDA Ant task in a `build.xml` file:

```
<project name="${project.name}" default="jars">
...
<unzip src="CarRentalSystem15.zargo"
       dest="${build.dir}/unzipped"/>

<andromda basedir="${build.dir}/unzipped"
          includes="CarRentalSystem15.xmi"
          lastModifiedCheck="false"
          typeMappings="./src/xml/TypeMapping.xml">

  <outlet cartridge="ejb"
          outlet="entity-beans"
          dir="/home/myproject/src/java/ejb/gen/" />
  ...
</andromda>
...
</project>
```

Listing 3.5: AndroMDA ant task example.

The `<andromda>` task supports nested `<outlet>` tags to customize the mappings for output directories where the generate source code will be placed.

It also supports a nested `<repository>` tag to write extensions to AndroMDA. AndroMDA by default reads models from XMI files. If wanted, AndroMDA could read the model from another format, or source type. This tag would be useful for implementing that sort of extension.

The attribute `transformClassname` specifies the name of the java class that implements the script helper that the Velocity code templates access as `$transform`. The attribute `classname` specifies the name of the java class that implements AndroMDA's meta data repository. One way to write an own meta data repository is by extending the class `MDRepositoryFacade`.

3.4 Enterprise JavaBeans

Enterprise JavaBean [7] is the server-side business layer component specification for the J2EE (Java 2 Enterprise Edition) platform. EJBs enables rapid and simplified development of distributed, transactional, secure and portable Java applications. The developers of EJBs do not have to design and program transactions, security and remote accessability for their own because this is done by the container providers, who are specialists on that area. Instead the component developers use declarative methods, facilitating deployment descriptors to define, how the container should act for the specific components.

Enterprise beans are the J2EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the J2EE server, see figure 3.5. The J2EE specification defines several APIs which a container has to provide to its components.

There are 30 and more J2EE implementations available on the market[7]. AndroMDAs cartridges currently generate components and descriptors for the Open Source application server JBoss [13].

If an application has any of the following requirements the use of enterprise beans might be considered:

- The application must be scalable. To accommodate a growing number of users, it might be necessary to distribute an application's components across multiple machines might be needed. Not only can the enterprise beans of an application run on different machines, but their location will remain transparent to the clients.
- Transactions are required to ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With just a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous[8].

[7] A listing of J2EE application server can be found under http://www.theserverside.com/reviews/matrix.jsp (from September 10th, 2003)

[8] For instance web browsers, handhelds, native java applications, applets etc.

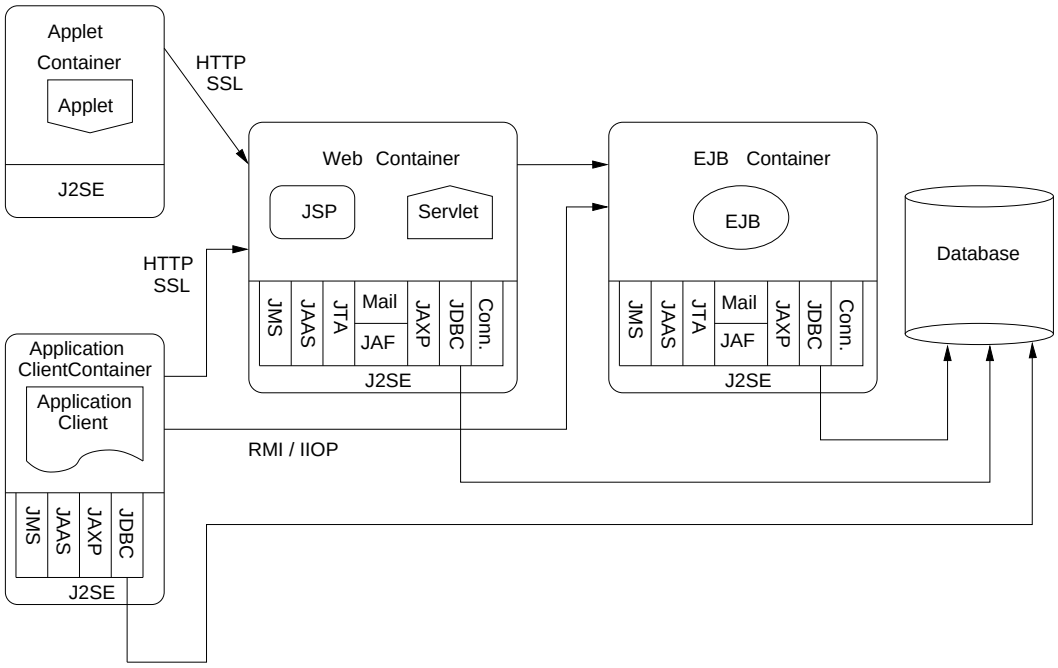


Figure 3.5: J2EE architecture in terms of its containers and APIs [12].

3.4.1 Types of Enterprise Beans

Enterprise Beans are divided in three categories, each for certain purposes.

Enterprise Bean Type	Purpose
Session	Performs a task for a client
Entity	Represents a business entity object that exists in persistent storage
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously

Table 3.6: Types of Enterprise Beans.

Session Beans

A session bean represents a single client inside the J2EE server. To access an application that is deployed on the server, the client invokes the session

bean's methods. The session bean performs work for its client, offloading the client from complexity tasks by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared - it just has one client, in the same way that an interactive session just has one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean is no longer associated with the client.

There are two types of session beans:

Stateful Session Bean: In a stateful session bean, the *instance variables* represent the state of a unique client-bean session. The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears.

Stateless Session Bean: When a client invokes the method of a stateless bean, the beans instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Because stateless session beans can support multiple clients, they can offer better scalability for applications that require a large numbers of clients.

Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients. At times, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage. Therefor, stateless beans may offer better performance than stateful beans.

Entity Beans

An entity bean represents a business object in a persistent storage mechanism. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys and may participate in relationships with other entity beans.

There are two types of persistence for entity beans:

bean-managed: With bean-managed persistence (BMP), the entity bean code contains the calls that access the storage. It is the responsibility of the bean developer to make the entity bean persistent.

container-managed: For container-managed persistence (CMP) beans, the EJB container automatically generates the necessary storage access calls. The developer only declares in a descriptor file, which attributes of the entity bean should be persistent.

Like a table in a relational database, an entity bean may be related to other entity beans. Relationships are implemented differently for entity beans with bean-managed persistence and those with container-managed persistence. With bean-managed persistence, the bean developer has to implement the relationships. With container-managed persistence, the EJB container takes care of the relationships. Relationships in entity beans with container-managed persistence are often referred to as container-managed relationships (CMR).

Message-Driven Beans

A message-driven bean allows J2EE applications to process messages *asynchronously*. It acts as a JMS (Java Message Service) message listener. The messages may be sent by any J2EE component- an application client, another enterprise bean or a Web component -or by a JMS application or system that does not use J2EE technology. Message-driven beans currently process only JMS messages, but in the future they may be used to process other kinds of messages[9].

3.4.2 The Contents of an Enterprise Beans

To develop an enterprise bean, the bean developer must provide the following files:

- **Deployment descriptor:** An XML file that specifies information about the bean such as its persistence type and transaction attributes.

[9]The new EJB 2.1 specification provides a generalization of message-driven beans to support additional messaging types.

- **Interfaces:** The remote and home interfaces are required for remote access. For local access, the local and local home interfaces are required. (Please note that these interfaces are not used by message-driven beans.)
- **Enterprise bean class:** Implements the methods defined in the interfaces.
- **Helper classes:** Other classes needed by the enterprise bean class, such as exception and utility classes.

3.5 XDoclet

XDoclet [24] is a code generation engine. It enables *Attribute-Oriented Programming* for Java. In short, this means that programmers can add more significance to their code by adding meta data (attributes). This is done in special JavaDoc tags. XDoclet parses the source files and generates many artefacts such as XML descriptors and/or source code from it. These files are generated from templates that use the information provided in the source code and its JavaDoc tags.

Currently XDoclet can be used as part of the build process utilizing Jakarta Ant (see sec. 3.3.6) only. Although XDoclet originated as a tool for creating EJBs, it has evolved into a general-purpose code generation engine. XDoclet consists of a core and a constantly growing number of modules. It comes with a set of modules for generation of different kinds of files. Users and contributors can write their own modules (or modify existing ones) if they wish to extend the functionality of XDoclet.

Listing 3.6 shows a method of an enterprise bean class with XDoclet tags for EJBs. The listed tags will cause XDoclet to generate method signatures into the local and remote interface of the bean. In addition XDoclet will create an transaction entry in the `ejb-jar.xml` file for the bean class.

3.6 Struts

The Struts [21] project provides an open source framework for building web applications. The core of the Struts framework is a flexible control layer

```
...
* @ejb.interface-method view-type="both"
* @ejb.transaction type="Required"
*/
public java.lang.String addEntityX(EntityXVO vo)
{
...
```

Listing 3.6: Example of XDoclet tags in an EJB class.

based on standard technologies like Java Servlets, JavaBeans, ResourceBundles, and XML (Extensible Markup Language), as well as various Jakarta Commons packages. Struts encourages application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller (MVC) [16] design paradigm.

Struts provides its own controller component and integrates with other technologies to provide the model and the view. For the model, Struts can interact with any standard data access technology, including Enterprise Java Beans, JDBC (Java Database Connectivity) and Object Relational Mapping frameworks. For the view, Struts works well with JavaServer Pages (JSPs), including JavaServer Standard Tag Library (JSTL) and JavaServer Faces (JSF), as well as Velocity Templates, XSL Transformations (XSLT), and other presentation systems.

3.6.1 Struts Controller Components

The controller component in an MVC application has several responsibilities, including receiving input from a client, invoking a business operation, and coordinating the view to return to the client. The controller is implemented by a Java servlet. This servlet is the centralized point of control for the web application. The controller servlet maps user actions into business operations and then helps to select the view to return to the client based on the request and other state information. Figure 3.6 shows the Struts controller mechanism.

3.6.2 Struts Model Components

The Struts framework does not offer much in the way of building model components. Many frameworks and component models are already available for dealing with the business domain of an application, including Enterprise

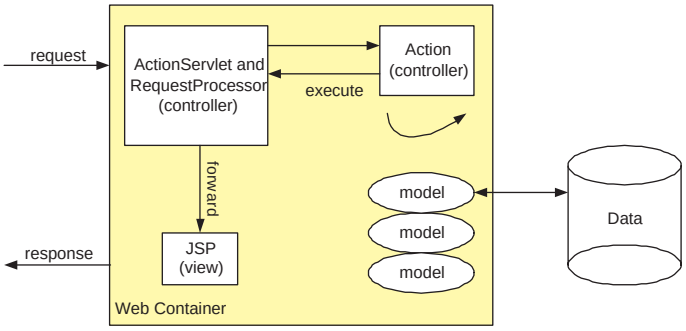


Figure 3.6: Struts controller mechanism.

JavaBeans (EJBs) and Java Data Objects (JDO), or you can use regular JavaBeans and an Object Relational Mapping (ORM) Tool. The Struts framework is not limited to one particular model implementation.

3.6.3 Struts View Components

The framework uses the view components to render dynamic content for the client. Based primarily on Java Server Pages, the components provide support for internationalization, user-input acceptance, validation, and error handling, making it easier for the developer to focus on business requirements.

3.7 J2EE Patterns

The J2EE Patterns [10] represent a collection of J2EE-based solutions to common problems, reflecting the collective expertise and experience of Java technology architects[10]. Patterns describe the use, functionality and relation of objects.

The following patterns became important during the work on the extended cartridges (see sec. 4.2):

Service Locator: The Service Locator provides lookup and lifecycle services to clients, typically the Business Delegates. The Service Locator

[10] A comprehensive collection of J2EE patterns can be found at http://www.theserverside.com/patterns/index.jsp (from September 10th, 2003)

hides the implementation details of the lookup service and the complexity of the lookup process from the clients. It provides a uniform central point for all lookups in the application.

Business Delegate: Business Delegate reduces coupling between tiers and provides an entry point for accessing the services that are provided by another tier. The Delegate may also provide results caching for common requests to improve performance. A Business Delegate typically uses a Service Locator to locate service objects, such as an EJB Home object and JMS Connection factory.

Session Façade: The Session Façade provides coarse-grained services to the clients by hiding the complexities of the business object interactions. The Session Façade may use the Service Locator pattern to locate services.

Value Objects: The Value Object pattern provides best techniques and strategies to exchange data across tiers (that is, across system boundaries). This pattern attempts to reduce the network overhead by minimizing the number of network calls to get data from the business tier.

3.7.1 ValueTreeBuilder

During the work on the cartridges the problem of reflecting the entity graph in the value objects occurred. The first attempt was to add logic into the `getValueObject():EntityVO` method which called each related entity for its value object. This worked fine but couldn't handle circular relations, additionally it is not always necessary to return the whole entity graph. Luckily there were more people dealing with the same misery. Gavin Hughes posted the `ValueTreeBuilder` class (based on an idea by Tim Lee) on the *middlegen-user* mailing list [15].

The tree builder takes a definition for the value object graph, and uses the local interface of the entity to build it.

Usage

To use the `ValueTreeBuilder` follow instructions below:

1. Get the `local` interface of the entity that the tree is to be built for.
2. Create an *include tree* for the tree builder to use.

```
...
entA = entAHome.findByPrimaryKey(idA);
String it0 = "";
String it1 = "*";
String it2 = "EntityC";
String it3 = "EntityB.*";
String it4 = "EntityB, EntityC";
String it5 = "EntityB.EntityC";

valueA = (EntityAVO)
         ValueTreeBuilder.getValueTree(entA,
         ValueTreeBuilder.createIncludeTree(itX));
...
```

Listing 3.7: `ValueTreeBuilder` include tree examples.

3. Call `ValueTreeBuilder.getValueTree(...)` with the entity and the *include tree*.

The *include tree* specifies the tree of entities you want to be included within the value object. It is `String` using a sort of package notation and wildcards. You can use "*" to get everything reachable from an entity.

Include Tree Examples

The entity - value object graph in figure 4.2 consists of three entities. `EntityA` has a bidirectional n:m relation to `EntityB` and a bidirectional 1:1 relation to `EntityC`. `EntityB` has an unidirectional 1:n (1->n) relation to `EntityC`. The different *include tree* examples in listing 3.7 yield the following:

The `valueA` will always contain its CMP fields that's why it is only explained for `it0`. The builder handles cyclic relationships safely, and build a self-referencing value tree.

`it0` `valueA` will only contain its CMP fields. The collection of `EntityBVOs` will be empty and the `EntityCVO` will be `null`.

`it1` `valueA` will contain the whole graph which is reachable from `EntityA`.

`it2` `valueA` will contain the `EntityCVO` with its CMP fields. `EntityBVO` is omitted because it is not in the include tree.

`it3` `valueA` will contain a collection of `EntityBVO` with its CMP fields. Each `EntityBVO` will contain the whole graph which is reachable from `EntityB` (`EntityBVOs` will contain a collection of `EntityCVOs`, which will also reference back to `EntityAVOs`).

`it4` `valueA` will contain a collection of `EntityBVO` with its CMP fields and the `EntityCVO` with its CMP fields.

`it5` `valueA` will contain collection of `EntityBVO` with its CMP fields. All `EntityBVO` in addition will contain a Collection of `EntityCVOs` with its CMP fields (the `EntityCVOs` will not reference back to the `EntityAVOs`).

For the `ValueTreeBuilder` to work, all entities implement a `getValueObject()` method which returns a value object containing all CMP fields for that entity.

Chapter 4

Results

In this chapter the results will be shown and experiences collected during the work on the extended cartridges will be discussed.

The software products developed on the Institute of Biomedical Engineering - Bioinformatics Group [5] - are multi layered, web-based and data driven applications written in Java programming language and often based on the J2EE platform. Therfor the integration framework was build based on EJB and Struts technology.

AndroMDA provides core functionality for code generation and delivers standard cartridges for EJBs and Struts. The standard cartridges lack several disadvantages which were removed during the work on the extended cartridges. The features added are listed below:

- Clean separation of business layer (EJB) and presentation layer (Struts) through the use of J2EE patterns for business interface, service locator, business delegate and value objects.
- Implementation of "CRUD" (create, read, update and delete) methods for persistent objects in the business and presentation layer.
- A project wizard for creating new projects containing a default model which minimises initial efforts.

The EJBs generated by the extended cartridge currently contain deployment descriptors for the JBoss [13] application server. The EJB cartridge must be adopted if other J2EE implementations should be supported.

The default model is stored in a `*.zargo` file readable for Poseidon CASE tool [8] because Poseidon Community edition if free of charge and used as the default UML Case tool during development.

4.1 Extended Cartridges

The intention of extending the cartridges was to provide a better solution for modelling and programming data driven web applications. The extended cartridge provides the following features:

- Bean-managed persistence Entity Beans
- Value object generation for entities
- Integration of `ValueTreeBuilder` [3.7.1] for building value object graphs
- Standard Struts actions for create, read, update and delete entities ("CRUD" methods)
- Standard `struts-config.xml` file with dynamic form beans
- Standard JSPs for listing, adding, editing and removing entities
- `Service Locator` pattern with `Business Interface` and `Business Delegate` pattern.

This features will allow the programmers to generate quickly a powerful, multi layered, extendable data driven web application based on best practise patterns, standard- and open source technologies.

4.2 Extended EJB Cartridge

The extended EJB cartridge provides all features of the standard cartridge with the extensions described below.

4.2.1 Bean-managed Persistence Entities

The cartridge generates bean-managed persistence entity beans for each class with the stereotype $<<BMPEntity>>$[1]. The cartridge can handle associations between CMP and BMP entities. The code for the relation management is put into an abstract bean classes of the CMP and BMP entities. Like in case of CMP beans there are also implementation classes for BMP beans where business and persistence logic implemented by the developer. This class will be generated only once and will never be touched by the generator again[2].

For n:m and unidirectional 1:n (1–>n) relations the developer must add further information to the model. This is performed by defining a class with the stereotype $<<EntityRelation>>$. The relation class must have dependencies with the stereotype $<<EjbRef>>$ to the corresponding entity classes and an attribute with the stereotype $<<PrimaryKey>>$. The relation class will be implemented as a CMP entity bean with the specified attribute as its `PrimaryKey` and two foreign keys of the corresponding entities[3]. Figure [4.1] shows an example for a unidirectional 1–>n relation between a CMP and BMP bean in the UML model.

For 1:n container-managed relations (CMR) you can define the Tagged value `@andromda.cascade-delete` of the association. This will cause a cascade delete on the n-side if the 1-side is deleted. For instance if you have a "Shop" to "SalesAssistant" (1:n) relation and you remove a shop entity, all "sales assistants" who belong to that shop will also be removed[4].

4.2.2 PrimaryKey Generator Factory

The extended cartridge uses the `Factory` pattern to get a `PrimaryKey`. Listing 4.1 shows a snip of the `create` method in `EntityA` class. The interface, factory and implementation sources are included in the project wizard (see sec. 4.5).

[1] This name should perhaps be reconsidered in terms of PIM and PSM. BMPEntity is too close to the EJB technology.

[2] It is the responsibility of the programmer to implement the persistence logic in the implementation class.

[3] The implementation as CMP entity is the first attempt. Using direct database access via JDBC could perhaps have some advantages.

[4] Cascaded delete isn't implemented yet for $<<BMPEntity>>$.

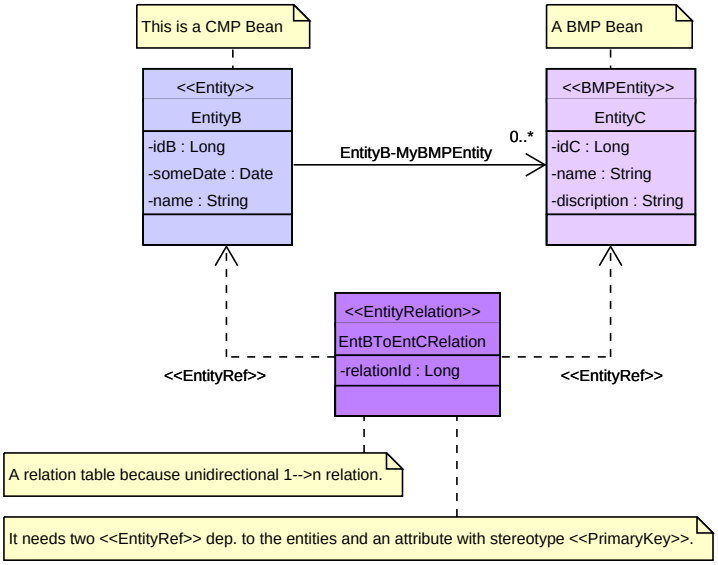


Figure 4.1: Example for bean-managed persistence entities.

```
...
java.lang.Long pk =
  PKGeneratorFactory.getPKGenerator(
  GlobalConstants.KEYGENERATORTYPE).
  getPrimaryKey("EntityA");
...
```

Listing 4.1: PrimaryKey generator usage.

4.2.3 Util Classes

If specified in the XDoclet ant task, XDoclet generates `utilobjects` for enterprise beans. Basically these objects provide static helper methods for looking up `home` and `localhome` interfaces in the Java Naming and Directory Interface (JNDI) repository of the application server. For further information about the XDoclet ant tasks, please visit the XDoclet home page [24]. Listing 4.2 gives an example of an XDoclet task with activated `utilobjects` generation.

The extended ejb-cartridge has also a template to generate `utilobjects` for session beans. An `utilobject` is generated, if a class in the model has the stereotype $<<SessionBeanUtil>>$ and the following tagged values are defined:

```
<project name="${project.name}" default="jars">
 ...
 <ejbdoclet destdir="${ejb.java.dir.gen}"
            ejbspec="2.0"
            force="false">
 ...
   <utilobject cacheHomes="true">
   </utilobject>
 ...
</project>
```

Listing 4.2: XDoclet Ant task example.

`@sbutil.home` The fully qualified class name of the home interface of the session bean.

`@sbutil.remote` The fully qualified class name of the remote interface of the session bean[5].

`@sbutil.jndi` The `jndi-name` under which the session bean ins bound on the application server.

The application developer has to provide a file with JNDI properties for each *<<SessionBeanUtil>>* class. The utility class reads the file from the `classpath` and loads the `InitialContext` with the specified properties.

For instance if you have a util class with the name `SomeSBUtil` then you must place a `SomeSBUtil.properties` file containing the JNDI properties on the `classpath` for the component which uses the util class.

The session beans in the extended ejb-cartridge are using the `utilobjects` to get references of the `localhome` interfaces of the entity beans.

4.2.4 Value Objects

As described in section 3.7, the Value Object pattern provides the best technique and strategy to exchange data across tiers. Usually value objects exist per use case, but for data driven applications it's very likely that there will

[5]Currently the information about the remote interface is only used for documentation but might be used to obtain a remote interface of the bean in the next releases.

be ”CRUD” (Create, Read, Update and/or Delete) use cases for each entity. That's why the value objects in the extended ejb-cartridge are mapped to entities.

A value object class in the model has the stereotype $<<ValueObject>>$ and an association to an $<<Entity>>$ or $<<BMPEntity>>$ class. The generated value object will have set/get methods for all attributes defined in the entity. In addition it will have set/get methods for other value objects, reflecting the relations of the corresponding entity. Figure 4.2 shows an example for modelling value objects. In this example the `EntityAVO` will have set/get methods for the attributes defined in `EntityA` and for its related `EntityCVO` and `EntityBVO`. Entities without a value object, will not be reflected in the value object graph.

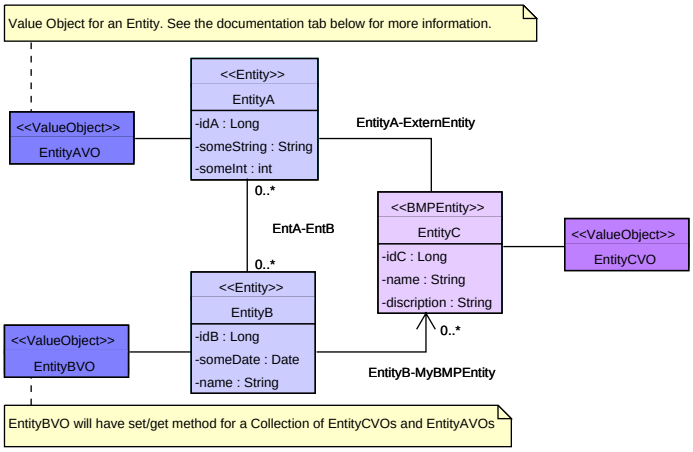


Figure 4.2: Example for value objects.

Entities with value objects will have the following methods in their abstract bean class:

`ejbCreate(EntityVO:vo)` The `ejbCreate` method will set the container-managed persistence fields according to the values in the `vo` and generate a new `PrimaryKey` for the entity. The `ejbCreate` method does not check, whether the `vo` already contains a `PrimaryKey`.

`ejbPostCreate(EntityVO:vo)` The `ejbPostCreate` method performs the relation handling for the entity. This is not straight forward and needs

a more detailed inspection. If the entity has relations which are reflected in the `vo`, then it will get the related value objects form the `vo` and call a helper method `setEntityXVO(vo.getEntityXVO())`.

`setEntityXVO(EntityXVO:vo)` If the `vo` is `null`, then the related entity will also be set to `null`. Otherwise if the `PrimaryKey` of the `vo` is `null` the `create(vo)` method on the `home` interface will be called. If the `PrimaryKey` is not `null`, the method will perform a `findByPrimaryKey()` and set the relation with the found entity[6]. See listing 4.3 for an example[7].

`getValueObject():EntityVO` This method is required by the `ValueTreeBuilder` (see sec. 3.7.1) and returns a new value object with all CMP fields set to the values of the entity. The `ValueTreeBuilder` handles the relations using Java reflection.

`update(EntityVO:vo)` First it sets the CMP fields except for the primary key field, then it utilises the helper methods `setEntityXVO(vo.getEntityXVO())` to set the relations of the entity bean according to the relations in the `vo`.

4.2.5 Services and Standard Operations for Entities

Classes with the stereotype $<<Services>>$ will be implemented as stateless session beans. For all dependencies with the stereotype $<<EntityRef>>$ which point to $<<Entity>>$ or $<<BMPEntity>>$ "CRUD" methods in the abstract session bean class will be generated. In addition the entities have to define a tagged value `@andromda.service.standardoperations` with the value `true`, otherwise no standard operations will be generated.

CRUD methods in the session beans:

`addEntityX(EntityXVO:vo):PrimaryKey` This Method will create a new entity with the given value object `vo` and return the `PrimaryKey` of the newly generated entity. `CreateException` and `NamingException` will be converted to component exception (see sec. 3.3.2).

[6] With this mechanism it is possible to create a bunch of entities with only one value object graph.

[7] `EntityA` to `EntityB` could be a n:m or 1:n relation

```
...

// ---- postCreate Helper Methods for relations ----

public void setEntityBVOs(java.util.Collection entityBs)
  throws javax.ejb.FinderException,
         javax.naming.NamingException,
         javax.ejb.CreateException
{
  if (entityBs == null) {
    setEntityBs((Collection)null);
  } else {
    Collection col = new ArrayList();
    java.util.Iterator iter = EntityBs.iterator();
    while(iter.hasNext()) {
      EntityBVO someVO = (EntityBVO)iter.next();
      java.lang.Long id = someVO.getIdB();
      EntityBLocalHome entityBLH =
          EntityBUtil.getLocalHome();
      EntityB entityBRef;
      if (id != null && !id.toString().equals("") &&
          !id.toString().equals("0")) {
        entityBRef = entityBLH.findByPrimaryKey(id);
        entityBRef.setEntityA(
          (EntityA)this.context.getEJBLocalObject());
      } else {
        entityBRef = entityBLH.create(someVO);
        entityBRef.setEntityA(
          (EntityA)this.context.getEJBLocalObject());
      }
    col.add(entityBRef);
    }
    setEntityBs(col);
  }
}
...
```

Listing 4.3: Helper method to set entity relations with value objects.

`findEntityXById(PrimaryKey:pk, String:includeTree):EntityX` Calls `findByPrimaryKey(pk)` on the local home interface, calls the `ValueTreeBuilder` with the found entity and the `includeTree` and returns the Value Object graph. `NamingException` will be converted to component exception.

`findAllEntityXs(String:includeTree)` Calls `findAll()` on the local home interface. Afterwards it calls the `ValueTreeBuilder` for each found entity and returns a `Collection` of value objects. `NamingException` will be converted to component exception.

`removeEntityXById(PrimaryKey:pk)` Calls `remove(pk)` on the local home interface. `RemoveException` and `NamingException` will be converted to component exception.

`updateEntityX(EncityXVO:vo)` Gets the primary key from the `vo`, if the primary key is `null`, a component exception will be thrown. If the `PrimaryKey` is not `null`, `findByPrimaryKey(vo.getIdX())` will be called on the `localhome` interface. Then the `update(vo)` method of the entity is called. `FinderException`, `NamingException` and `CreateException` will be converted to component exception.

4.3 Service Locator and Delegate Patterns

Separating the presentation layer from the business layer is important for maintaining the code. For example it is a bad habit to work directly with EJB classes in the web interface. Any changes in the EJB layer would cause a change in the web layer. Therefor the extended cartridges force the definition of business interfaces, delegates and locator factories.

In this section the modelling and artefacts of these patterns will be explained. Like an $<<Entity>>$ is a persistent and durable component, a $<<Service>>$ models workflows or use cases. These services should be accessible for the next higher layer (presentation layer).

The extended cartridges define a $<<ServiceBusinessInterface>>$ class with dependencies[8] to one or more $<<Service>>$ classes. The business interface will be implemented as a Java interface which defines all business

[8] A dependency is an UML element and is used e.g. in a class diagram similar to associations.

```
...
EntityXVO vo = new EntityXVO();
...
java.lang.Long id =
SomeServiceFactory.getInstance().addEntityX(vo);
...
```

Listing 4.4: Example for a service call.

operations and standard operations of all dependent $<<Service>>$ classes.

In addition to the business interface $<<BusinessDelegate>>$ and $<<ServiceFactory>>$ class must be defined. The delegate class will handle the creation of services and delegate method calls to it. The factory works as a service locator and is responsible for creating the proper delegate classes. Listing 4.4 shows the use of the factory and business interface (i.e. in a Struts action class) for service calls.

4.4 Extended Struts Cartridges

The extended struts cartridge simplifies the generation of Struts actions, dynamic forms, configuration files and JSPs for data driven applications. For each class with the stereotype $<<StandardStrutsAction>>$ which has an association to an entity the cartridge will generate:

Action classes extending the `LookupDispatchAction` class. These classes provide "CRUD" methods for the corresponding entity. A converter method handles the conversion from form to value object and back. The cartridge generates the action mapping for the entity into the `standard-actions.xml` file which will be included into the `struts-config.xml` file as `XML-include`.

`DynaValidatorActionForm` definitions for the corresponding entity into the `standard-forms.xml` file which will be included into the `struts-config.xml` file as `XML-include`.

JSPs to add, edit and remove entities.

`Validator` **definitions** for the dynamic forms. The tagged values `@andromda.struts.validation.depends` and `@andromda.struts.validation.fieldtags` of attributes in the entity will generate entries in

```
<form-validation>
  ...
  <formset>
    ...
    <form name="EntityXForm">
      ...
      <field property="name" depends="required,mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^[a-zA-Z]*$</var-value>
        </var>
      </field>
      ...
    </form>
    ...
  </formset>
  ...
<form-validation>
```

Listing 4.5: Validator example.

the `standard-validation.xml` file. Listing 4.5 shows the output for the tagged values in table 4.1 where `XXX=<var>...</var>`.

Standard navigation menu to find all and add new entities. The `MainMenu` Servlet reads a menu definition file (in xml format) and handles the menu interaction for the web interface. The source for the `MainMenu` Servlet is included in the project wizard.

Each $<<StandardStrutsAction>>$ class has to have a dependency to a $<<ServiceFactory>>$ class. The actions will use the factory like in case of listing 4.4.

Tag	Value
`@andromda.struts.validation.depends`	`required,mask`
`@andromda.struts.validation.fieldtags`	`XXX`

Table 4.1: `Validator` tags example for an attribute of an entity.

4.5 Project Wizard

The project wizard helps to create a new project and can be downloaded from the CVS repository. The wizard is actually an ant task, which reads the properties defined in the `wizard.properties` and performs some copying and renaming. Before calling the wizard, the `wizard.properties` file has to be edited. To run the wizard, a command window (or shell) must be opened. After changing to the wizard directory the ant task can be called:

```
ant -f wizard.xml -Dproject.home=<PROJECT_DIRECTORY>
```

This will create the default project in the specified directory. The project consists of several directories described in table 5.5, an Ant `bild.xml` file and a JBuilder (an IDE from Borland [14]) project file. Additionally a default UML model in the uml-source directory for Poseidon UML CASE tool [8] will be created. It has predefined stereotypes and contains diagrams for domain-, value object-, util- and standard struts action classes. It is recommended to start with this model and run through the generation process with the predefined classes.

To run the generator change to the project directory and call:

```
ant
```

The command window will output several information about the generation process. Fist the model is read from the specified XMI file, then AndroMDA locates the cartridges on the classpath and generates the code (marked with XDoclet). Then the XDoclet code generation task is called. At the end follows compilation, copying, archiving and deployment. The task logic and dependencies are defined in the `build.xml` file.

To list all ant tasks which are defined in the `build.xml` file call:

```
ant -projecthelp
```

This will output a list of ant tasks for the `build.xml` in the project directory (see table 5.4 in the appendix).

```
<server>
  ...
   <attribute name="URLs">
deploy/,file:///C:/projects/SomeProject/build/
   </attribute>
  ...
</server>
```

Listing 4.6: Example for an URL entry of the `jboss-service.xml` file.

4.5.1 Project Directory Structure

Table 5.5 in the appendix lists the project directory structure and its meaning.

The `build` directory contains subdirectories with the extracted EJBs and Web application. During development it is useful to let the JBoss `URLDeploymentScanner` watch the `build` directory, if the `ejb-jar.xml` and `web.xml` files change, JBoss will start a redeployment. Therefor in the `jboss-service.xml` file in the `conf` directory the projects `build` directory must be added to the URL list. Listing 4.6 gives an example how the URLs attribute could look like.

It is useful for web development to modify the JSPs without redeployment. Just edit the JSPs in the `build` directory and reload the browser. The changes can bee seen without redeploying the whole web application

Chapter 5

Discussion

This chapter will sum up the experiences which have been made during the work with AndroMDA and point out some general aspects for generation driven software development.

AndroMDA with the extended cartridges provide an extendable and powerful integration framework based on standard technologies (UML, XMI, EJB, JSP, Servlets etc.) and Open Source products (Ant, XDoclet, Velocity, Struts, JBoss). It allows fast design and prototyping for data driven web applications and speeds up the development process as well as provides better maintainability, robustness and consistency. By the use of the "project wizard" (see sec. 4.5), the initial efforts are decreased. The "UML Template Model" provides a good starting point for modelling and development.

The integration of generative software development has several drawbacks at the beginning of its introduction: programmers are faced with new technologies; school enrolment and training for developers who have little experience with code generators, modelling and UML CASE tools ; restructuring the development team; and more. Benefits will only turn out at a later date: when the team is used to the new techniques used for development; applications have to be ported to new upcoming technologies; architecture design changes have to be made on existing code because of performance issues; and more.

5.1 Pros and Cons for Code Generators

Code generators give programmers the potential for faster development and generally shorten the time to market for an application. The maintainability, consistence and robustness increases the more code is generated. Design patterns can be easily converted into the generator templates, adopted and extended for the projects requirements (if the code generator is not a closed framework). In case of AndroMDA the templates and even the core generator are open source and therefor extendable without any limitations.

Project managers should consider an initial cost for establishing code generators. Some programmers must be familiar with the generator templates and the core technology to adopt them for the project needs. The other programmers will basically focus on business integration without any technical and architectural design challenges, which may reduce their motivation.

5.1.1 MDA Approach

Currently MDA is strongly promoted and more implementations and tools are available claiming to support the MDA architecture.

It is a fact that there are different lifetimes for the models of business processes (i.e B2B etc.), the software integrations and the implementation technologies (J2EE, .Net, CORBA etc.). To decouple these levels of modelling and the approach to define transformations between the models types (which actually describe the same but with different views) is an important step towards reusability, portability (platform independence) and interoperability. These are the key goals of the MDA approach, but if they can be reached is a question of the acceptance of the market. Especially the interchange of models between different MDA solutions (although XMI technology) will be challenging. For instance the XMI DTD for UML 1.3 standard exists since 1999 but interchanging the models between several CASE tool providers is a elaborate work.

5.2 AndroMDA is Work in Progress

AndroMDA provides a so called ”MDA light” solution. Most of the PSM is part of the templates which directly generate the implementation for of the

PSM. In other words, the PSM is omitted in AndroMDA and the implementation is generated directly from a PIM with tagged metadata.

The current version (2.0.3 final) provides a stable, extendable code generator framework for rapid software development. It is based on open source technologies and free of charge. The new features of release 2.0.3 final and future plans for release 3.0 are listed and discussed on TheServerSide.com[1].

5.3 Cartridge Management and Model Transformations

During cartridge development, a big lack of reusability was identified. This might be solved in release 3.0. after the introduction of model transformation before code generation (PIM to PSM). Putting generation logic into the Velocity templates hinders reusability because the lack of generalization for the templates (i.e. EJB 2.0 template extends EJB 1.1 and reuses functionality).

Developing cartridges is an error prone work. The "edit-templates -> generate test code -> compile test code -> deploy test application" process takes too much time. Each project will require minor changes for the code templates as requirements differ from project to project. Cartridge reusability will play a key role in the future context for AndroMDA.

The more cartridge will be developed, the more modelling restrictions will occur and the PIM will change more to a graphical representation for the implemented technology. Either there will be restrictions for the PIM which will complicate cartridge development or additional modelling elements for each cartridge will force the PIM to be more like an implementation model.

For each problem domain a metamodel should exist which describes the PIM. This model can be transformed to a PSM with transformation definitions. These definitions are not specified and will cause dependencies to the MDA tool providers.

[1] Follow the discussion about AndroMDA
http://www.theserverside.com/home/thread.jsp?thread_id=20801 (from September 10th, 2003)

5.4 AndroMDA in the Bioinformatics Domain

Bioinformatics covers many different domain categories from a technical point of view: data acquisition (i.e. DNA sequencing roboters); data validation (experimental results); data analysis and postprocessing (simulations, drug research etc.);

All these domains are data driven. Storing and retrieving biological data will be basic requirements for many applications in the Bioinformatics domain. The integration framework with AndroMDA and the extended cartridges frees the programmers from standard design considerations and let them focus on business requirements.

Appendix

XML elements in `andromda-cartridge.xml`

Element	Description
`<cartridge>`	The cartridge tag is used to delimit the cartridge descriptor and give a name to the cartridge.
`<property>`	The `<property>` tag is used to define a set of key-value pairs for a cartridge. These properties define which aspects of the target application's architecture are generated by this cartridge. In the future, these properties will be used for automatic aspect-to-cartridge mappings (not yet implemented).
`<stereotype>`	This tag is used to declare the stereotypes that trigger the core module to use this cartridge. Example: The EJB cartridge declares the $<<Entity>>$ stereotype because entities may be represented by entity beans.
`<outlet>`	This tag is used to declare a logical outlet name that can later be mapped to a physical directory using the `<andromda>` task in the `build.xml` script.
`<template>`	The `<template>` tag is used to describe the set of templates that will be used to generate source code.

Table 5.1: The XML elements used in the `andromda-cartridge.xml` file [2].

XML attributes for the <template> tag in `andromda-cartridge.xml`

Attribute	Description	Required
`stereotype`	Specifies the name of the stereotype that should trigger the use of this template.	Yes
`sheet`	Specifies the path (relative to the cartridge root) for a template (*.vsl) file to use for code generation.	Yes
`output-Pattern`	Specifies a pattern in java.text.MessageFormat syntax. You can use this pattern to tell AndroMDA how to construct output file names. The pattern can consist of any ordinary printable characters as well as some predefined placeholders for things that AndroMDA already knows about: {0} stands for the package directory of the class. {1} stands for the class name. {2} stands for the base part of the *.vsl file name. {3} stands for the name of the stereotype.	Yes
`outlet`	Specifies the logical name of the outlet where the cartridge will write the output files caused by this template.	Yes
`overWrite`	Specifies whether the files already contained in `<outlet>` should be overwritten when AndroMDA runs a second time.	Yes
`generate-EmptyFiles`	Specifies whether files should be generated even if the template did not produce any output. This can be used by the cartridge developer to decide if a certain file should be generated based on the information in the model. Note: If this property is set to "false", the template produces no output, if `overWrite` is set to "true", and an existing file is found (probably generated by a previous run), then this file is removed.	No, default is "true"
`transform-Classname`	Specifies a name of a java class that can be used to help the code generation templates. In some cases the velocity scripting language is not enough for what you would like to do in your code generation template. You can solve this problem by writing a java class to extend the `$transform` object in the template. One way to write your own helper is to extend the class `SimpleOOHelper`.	No

Table 5.2: Attributes of the `<template>` tag in the `andromda-cartridge.xml` file [2].

AndroMDA Ant Task Attributes

Attribute	Description	Required
`basedir`	Specifies the path to the directory location of the case tools' XMI files.	Yes
`includes`	This is the standard Ant `includes` attribute. Specifies any files or directories with XMI files that AndroMDA should try to process.	Yes
`excludes`	This is the standard Ant `excludes` attribute. Specifies any files or directories with XMI files that AndroMDA should not try to process.	No
`modelURL`	It can be used instead of the `includes` and `excludes` attributes to point directly at the location of your XMI file. It is useful if the model file is in a JAR and should not be unzipped. Unlike include it can only be used to process one XMI file.	Yes
`lastModifiedCheck`	This turns on or off the ability to check the last modified date on files in order to determine whether or not they need to be re-rendered or not. The value of this attribute can be "true, false, yes, no". By default, it is true, meaning that the last modified date should be checked and if the original `.xml` file has not changed the output file is not created again. This accelerates the build process because files that have not changed will not get reprocessed.	No, defaults to "true"
`velocityPropertiesFile`	This is the path to the `velocity.properties` file. It is an optional argument and by default is set to find the properties file in the same directory that the JVM was started in.	No

Table 5.3: Attributes of the AndroMDA Ant Task [2].

Ant tasks and directory structure of the default project

Ant task	Description
build	Builds ejb.jar and web.war
build-nogen	Builds ejb.jar and web.war without Xdoclet tasks
clean-action	Removes generated struts action files
clean-all	Removes all generated files
clean-build	Removes all jars, wars, ears and class files
clean-common	Removes generated common java files
clean-config	Removes generated config files
clean-ejb-config	Removes generated EJB config files
clean-ejb-java	Removes generated EJB java files
clean-html	Removes generated html files
clean-jsps	Removes generated JSPs
clean-test	Removes generated test java fils
deploy	Verifies and deploys generated ear in JBoss
deploy-nogen	Verifies and deploys generated ear in JBoss without XDoclet tasks
junit	Init testdb and performs all JUnit regression tests.
redeploy-ejbs	redeploys EJBs and WebApp

Table 5.4: Ant tasks for the `build.xml` file in the default project.

Directory	Description
./documentation	project documentation
./lib	lib directory
./resources	resource files come here
./src	the source directory
./src/app	application sources
./src/cartridges	cartridges directory for the project, contains the ejb-, struts-, java- and simple-example-cartridges
./src/java	java source files come here, the subdirectories contain ”generated” and ”manual” directories
./src/web	directory for the web application sources
./src/uml	directory for UML model source files
./src/xml	directory for XML source files
./build	directory for the extracted EJBs and web application
./toInstall	directory for the jar (EJB archive), war (web archive) and ear (enterprise application archive).

Table 5.5: Default project directory structure.

List of Modelling Elements

Table 5.6 gives a summary of the defined stereotypes for model elements and its purpose and meaning.

EJB Cartridge		
Stereotype	**Element**	**Meaning**
BMPEntity	class	For bean-managed persistence Entity Beans
EntityRelation	class	For n:m or unidirectional 1->n [BMP-CMP] or [BMP-BMP] relations
ValueObject	class	Value object for an BMP or CMP entity
SessionBeanUtil	class	Helper class for looking up and caching a `home` interface
ServiceBusinessInterface	class	Interface implemented by the delegate class; used by the upper layers
BusinessDelegate	class	Implements the `ServiceBusinessInterface` and delegates method calls to the underlying session beans
TemplateService-Factory	class	Creates and caches instances of delegate classes
Struts Cartridge		
Stereotype	**Element**	**Meaning**
StandardStrutsAction	class	Struts action class containing ”CRUD” methods for an entity
Java Cartridge		
Stereotype	**Element**	**Meaning**
GlobalConstants	class	Project specific class, holding constants
Exception	class	Java `Exception` class

Table 5.6: Stereotypes for the extended cartridges.

Table 5.7 gives a summary of the defined tagged values for model elements and its purpose and meaning.

EJB Cartridge	
Tagged Value / Element	**Meaning**
`@ejb.persistence` / class	The name of the table for the entity (i.e. `table-name="ENTITY_XXX"`)
`@ejb.persistence` / attribute	The name of the column for the attribute of an entity (i.e. `column-name="ATT_XXX"`)
`@andromda.service.-standardoperations` / class	Session beans will contain "CRUD" methods for entities which have this tag set to `"true"`.
`@andromda.-cascade-delete` / association between CMP entities	Deletes the related entities if entity is removed
`@sbutil.home` / SessionBeanUtil class	Fully qualified class name for the `home` interface of the session bean
`@sbutil.remote` / SessionBeanUtil class	Fully qualified class name for the remote interface of the session bean
`@sbutil.jndi` / SessionBeanUtil class	The JNDI-name under which the session bean is bound on the application server
Struts Cartridge	
Tagged Value / Element	**Meaning**
`@andromda.struts.-validation.required` / attribute	Declares which validation rules are required for the attribute of an entity
`@andromda.struts.-validation.fieldtags` / attribute	Declares which field tags the validation rule should contain (needs the required tag above).

continued on next page

continued from previous page	
Tagged Value / Element	**Meaning**
`@andromda.viewtype` / association	If the value is `"ComboBox"` the `Edit/Add`-JSP will contain a `ComboBox` for selecting a related entity. If not specified the `Edit/Add`-JSP will contain a `TextField`[2].
`@andromda.entity-Relation` / association	The association needs this tag if a relation to an BMP entity needs a $<<EntityRelation>>$. The value must be the `class name` of the $<<EntityRelation>>$[3].

Table 5.7: Tagged values for the extended cartridges.

[2]The related entity must have an attribute `"name"` to work with `ComboBox`
[3]This is not an ideal solution and there might be changes in the future!

Bibliography

[1] Web resource for microarray databases .
http://www.biologie.ens.fr/en/genetiqu/puces/bddeng.html
September 10th, 2003

[2] AndroMDA introduced by Matthias B.
http://www.andromda.com
September 10th, 2003

[3] Andromdas car rental example. Included in the binary and source distribution.

[4] Ant.
http://ant.apache.org
September 10th, 2003

[5] Bioinformatics Group - Institute of Biomedical Engineering - Graz University of Technology.
http://www.genome.tugraz.at/
September 10th, 2003

[6] Document Access Page on OMG.
http://www.omg.org/technology/documents/index.htm
September 10th, 2003

[7] Enterprise JavaBeans Technology.
http://java.sun.com/products/ejb/
September 10th, 2003

[8] Gentleware.
http://www.gentleware.com/
September 10th, 2003

[9] Hibernate.
http://hibernate.bluemars.net/
September 10th, 2003

[10] J2EE Patterns.
http://java.sun.com/blueprints/patterns/index.html
September 10th, 2003

[11] Java.
http://www.java.sun.com
September 10th, 2003

[12] Java Server Programming, J2EE 1.3 Edition.
http://www.wrox.com
September 10th, 2003

[13] JBoss.
http://www.jboss.org
September 10th, 2003

[14] JBuilder IDE from Borland.
http://www.borland.com
September 10th, 2003

[15] Middlegen.
http://boss.bekk.no/boss/middlegen/
September 10th, 2003

[16] Model-View-Controller.
http://java.sun.com/blueprints/guidelines/
September 10th, 2003

[17] Netbeans.
http://www.netbeans.org/
September 10th, 2003

[18] OMG Object Management Group.
http://www.omg.org/
September 10th, 2003

[19] OMG's UML Unified Modeling Language^{TM}.
http://www.omg.org/uml
September 10th, 2003

[20] OMG's Unified Modeling Language (UML).
http://www.omg.org/gettingstarted/what_is_uml.htm
September 10th, 2003

[21] Struts.
http://jakarta.apache.org/struts/
September 10th, 2003

[22] The NCBI Handbook.
http://www.ncbi.nlm.nih.gov/books
September 10th, 2003

[23] Velocity.
http://jakarta.apache.org/velocity/
September 10th, 2003

[24] XDoclet.
http://www.xdoclet.org/
September 10th, 2003

[25] Merkle B. Gemachte Vorschriften. iX 4/2003, S 114-117.

[26] Ball CA, Sherlock G, Parkinson H, Rocca-Sera P, Brooksbank C, Causton HC, Cavalieri D, Gaasterland T, Hingamp P, Holstege F, Ringwald M, Spellman P, Stoeckert CJ Jr, Stewart JE, Taylor R, Brazma A, and Quackenbush J. Microarray Gene Expression Data (MGED) Society. *Science*, 298:539–539, 2002.

[27] Roos DS. COMPUTATIONAL BIOLOGY: Bioinformatics–Trying to Swim in a Sea of Data. *Science*, 291(5507):1260–1261, 2001.

[28] Venter JC et. al. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

[29] CRICK F. Central Dogma of Molecular Biology. *Nature*, 227:561–563, 1970.

[30] Miller J and Mukerji J. MDA Guide Version 1.0, May 2003. Document Number: omg/2003-05-01.

[31] Czarnecki K and Eisenecker UW. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, June 2000.
http://www.generative-programming.org/
September 10th, 2003

[32] Schmaranz K. Ausgewählte Kapitel der Softwareentwicklung - Programming for Large Libraries.
http://courses.iicm.edu/ak_swent/AKSWE_Skriptum.pdf
September 10th, 2003

[33] Bohlen M and Starke G. MDA Entzaubert. Objektspektrum, March 2003.
http://www.objektspektrum.de
September 10th, 2003

[34] Schena M, Shalon D, Davis RW, and Brown PO. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270:467–470, 1995. PM: 7569999.

[35] Völter M Stahl T. Moderne zeiten. IX 03/2003 S. 101, March 2003. http://www.heise.de/ix/

September 10th, 2003

[36] Markus V. Code Generation: A survey of Concepts, Techniques and Tools. www.voelter.de/data/presentations/ProgramGeneration.zip

September 10th, 2003