Model-driven Web Engineering with Open Source Technologies

Ismail Faizi

Kongens Lyngby 2014
IMM-M.Sc-2014-03

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

IMM-M.Sc-2014-03

Summary (English)

In Model-driven Software Development (MDSD), software is modeled on a high level of abstraction; from these models the code – or major parts of the code – can be generated fully automatically. Even though there are still some limitations to this approach, MDSD helps to develop software in a faster and more reliable way.

The focus of this project is Web Engineering. Starting from some practical examples, a modeling notation for web applications is developed that allows to model a web application on a high level of abstraction and independently from a specific target platform. The modeling notation allows the generation of the code for the web application in a fully automatic manner. The ultimate goal of this modeling notation is that the code for different target platforms can be generated from the same model. Due to time-limitations, however, the implemented code generator is for a single target platform (Joomla! 2.5 platform). However, the notation is not specific to this target platform. In addition to being independent from a specific target platform, the developed modeling notation and code-generation framework allows for easy integration with existing infrastructure and manually written code.

The modeling notation and code generation framework is developed based on the Eclipse Modeling Framework (EMF). The developed tool platform – named *Welipse* – is made available under an open source license.

In order to make sure that the developed concepts and notation is practically relevant, this project is done in cooperation with Peytz & Co, Copenhagen. The company provided a small but realistic example of a web application for devel-

oping the modeling notation as well as for evaluating it, moreover the company gave guidance on the aspects to cover in order to be practically relevant.

Summary (Danish)

I model-drevet software udvikling (MDSD) er software modelleret på et højt abstraktionsniveau. Fra disse modeller kan koden - eller større dele af koden - genereres fuldautomatisk. Selvom der stadig er nogle begrænsninger med denne tilgang, hjælper MDSD med at udvikle software på en hurtigere og mere pålidelig måde.

Dette projekts fokus er Web Engineering. Ved at starte fra nogle praktiske eksempler, en modelleringsnotation for web-applikationer er udviklet, der gør det muligt at modellere en web-applikation på et højt abstraktionsniveau og uafhængigt af en specifikt platform. Denne modelleringsnotation tillader fuldautomatisk generering af kode for web-applikationen. Det ultimative mål med denne modelleringsnotation er at, koden til forskellige platforme kan blive genereret fra den samme model. På grund af tidsbegrænsning, vil den udviklede kode generator understøtte en enkel platform (Joomla! 2.5 platform). Men modelleringsnotationen er ikke specifik for denne platform. Ud over at være uafhængig af en specifik platform, giver den udviklede modelleringsnotation og kode generator også mulighed for nem integration med eksisterende infrastruktur og manuelt skrevet kode.

Den udviklede modelleringsnotation og kode generator er baseret på Eclipse Modeling Framework (EMF). Det udviklede værktøj - ved navn *Welipse* - er stillet til rådighed under en open source licens.

For at sikre, at de udviklede koncepter samt notationen er praktisk relevant, er dette projekt udviklet i samarbejde med Peytz & Co, København.

Dette firma har leveret et lille, men realistisk eksempel på en web-applikation til udvikling af modelleringsnotationen samt vurdering af denne. Desuden er firmaet kommet med vejledning om de aspekter der skulle dækkes for at det udviklede værktøj kan være praktisk relevant.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science.

The thesis deals with development of modeling notation for web applications. The main focus is Model-Driven Web Engineering (MDWE) where a tool has been developed in order to facilitate MDWE.

The thesis consists of both theoretical aspects of MDWE – and Model-driven Software Development (MDWE) in general – and the realization of these aspects. In the case of the later, development of modeling notation, transformation of model-to-model and model-to-code has been covered. The work on this thesis began on September 2, 2013 and was completed on March 17, 2014. This thesis is credited with 35 ECTS points.

Lyngby, 17-March-2014

[illegible]

Ismail Faizi

Acknowledgements

I would like to thank my supervisor Ekkart Kindler, first for expanding my idea for this thesis and secondly for his constructive ideas which I have learned a lot from. It was both enjoyable and easy to work with him.

My heartfelt gratitude goes to John Christensen, my supervisor at Peytz & Co, and Morten Teisner from Peytz & Co, first and foremost for their cooperation in this project but also for their constructive ideas which, hopefully, have made the results practically relevant.

Thanks to Scott Stanchfield[1], whom I don't know personally, for his great ANTLR 3.x video tutorials which helped me refresh my ANTLR knowledge. I am also grateful to the people behind `tuxfamily.org`[2] for providing great tutorials about GMF.

In the end, I would like to thank everyone that has contributed to this thesis in some way or another. I am foremost grateful to have lived in the era of great researchers whose work has made this thesis possible. In the words of Isaac Newton "If I have seen further than others, it is by standing upon the shoulders of giants".

[1] http://javadude.com/index.html
[2] http://gmfsamples.tuxfamily.org/

Contents

Chapter 1

Introduction

Web applications are becoming more sophisticated, yet the development approaches involved in developing these applications still remain ad-hoc and on a low level of abstraction, i.e. concerned with specific technologies and platforms [KPRR06]. In order to accommodate for the increasingly complex and rapidly evolving nature of web applications, a discipline such as Web Engineering is needed.

A widely accepted approach for developing complex distributed applications is MDSD (Model-Driven Software Development). This approach advocates the use of models as the key artifacts in all phases of development. Usually, each model addresses one concern, independently from the rest of the issues involved in the construction of the software system. The transformations between models enable the automated implementation of a system from the different models defined for it.

One of the best known MDSD initiatives is MDA (Model-Driven Architecture) [MM03], proposed by OMG (Object Management Group) [OMG] an international organisation promoting the theory and practice of object-oriented technology in software development. MDA is a broad conceptual framework that describes an overall approach to software development. The goal of MDA is to separate business and application logic from its underlying platform technology. This way, changes in the underlying platform will not affect existing applications, and business logic can evolve independently from the underlying technology.

Web Engineering is a specific domain in which MDSD can be successfully applied [MRV08]. Today, a number of MDWE (Model-Driven Web Engineering) approaches exist. These approaches provide both methodologies and tools for the design and development of most kinds of web applications [MRV08]. However, these proposals also present some limitations. N. Moreno *et al.* (2008) claims, that these limitations are due to lack of support for modeling further concerns such as architectural styles or distribution and interoperateability of web systems. The reason behind these limitations is the approach being tied to particular architectural styles and technologies. Furthermore, most of these proposals were originally conceived to deal with particular kinds of web applications. A number of current MDWE approaches have adopted MDA principles in order to overcome some of their limitations. Moreover, by doing so, they have become interoperable and compatible with other tools and notations that have adopted MDA principles. Consequently, these tools can exchange data and models; achieving reusability. Reusability is one of the primary goals of OMG beside portability and interoperability [MM03].

Although, the approaches mentioned above provide both methodologies and tools in order to facilitate MDWE, they do not address the needs of web development industry which is unfamiliar with the notion of Model-driven web development; especially those who develop web applications based on open source technologies and platforms. In this thesis, we address this concern.

Our approach is also based on the MDA principles and provides a domain specific notation for modeling web applications on a higher level of abstraction. With this notation, it is possible to model concerns such as *content*, *navigation*, and *presentation*. The strength of our proposal and the underlying tool is the ability to prototype rapidly, and through this, promptly capture the requirements of the application, thus reducing the development time. Furthermore, the extensibility of the tool to target multiple platform is yet another strength. We will support the `Joomla!` platform [OSM] as the default platform and outline the requirements for extending the tool to other platforms such as Drupal [Buy].

We claim, that by applying a domain-specific language, the model-driven approach for developing web applications becomes feasible. This thesis attempts to demonstrate this. In order to accomplish this, this thesis has been done in cooperation with Peytz & Co[1], an IT company specializing in web development. They have, among others, provided a real life example for testing the results

[1]The company is located in Copenhagen: `http://peytz.dk`

of this thesis. In addition, they have driven the development of the modeling notation and the underlying tool.

This report is organized such that a running example, covering all the different aspects of the tool, is presented in the following chapter (chapter 2). The running example is followed by a background chapter presenting the various terms and notions of MDSD and MDWE (chapter 3). In chapters 4 and 5 the design of DSL for web modeling and the requirements specification of the tool has been presented, respectively. A detailed overview of how to use the tool is provided in chapter 6. This chapter is followed by a design chapter (ch. 7) discussing the various design decisions made during the design phase. Then, in chapter 8, the implementation of non-trivial parts of the tool are discussed. The way the tool has been tested is presented in chapter 9. This is followed by reflections chapter (chapter 10) which presents the related work, evaluation, and what is needed in order to extend the tool to other platforms. Chapter 11 gives some concluding remarks and outlines possible future works. Finally, a detailed user guide is provided in appendix A.

CHAPTER 2

Running Example

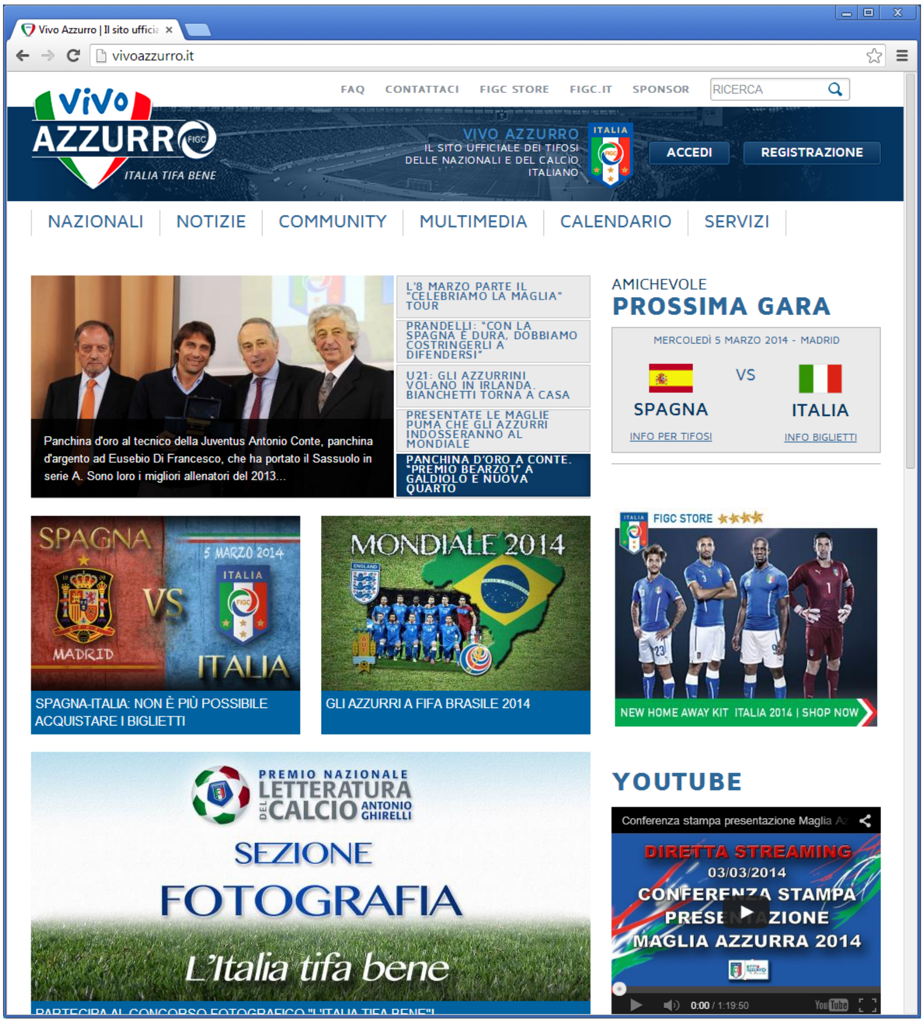
This chapter presents the running example that is used throughout this thesis. As mentioned in the introduction, this example has been provided by Peytz & Co.

The original web pages that the running example is based on, is a web portal[1] developed for the fans of Italian football. The site provides the fans with various kinds of information about the different Italian football *squads*. Figure 2.1 on the following page presents a screenshot of the main page of the website, while Fig. 2.2 on page 7 shows a screenshot of the page presenting the national A team (The Blues).

The national A team is given a lot of attention by providing a lot of news, events and information about the individual players. This information represents the player's biography, former and current clubs, role, and personal data such as name, date, place of birth, height etc. An example of such a page can be seen in Fig. 2.3 on page 8 which presents a player page on the original website.

The fans have the opportunity to register at the site where they can become members of the Vivo Azzurro club or purchase tickets for the upcoming national matches.

[1]http://vivoazzurro.it/



VIVO AZZURRO
IL SITO UFFICIALE DEI TIFOSI DELLE NAZIONALI E DEL CALCIO ITALIANO

Panchina d'oro al tecnico della Juventus Antonio Conte, panchina d'argento ad Eusebio Di Francesco, che ha portato il Sassuolo in serie A. Sono loro i migliori allenatori del 2013...

L'8 MARZO PARTE IL "CELEBRIAMO LA MAGLIA" TOUR

PRANDELLI: "CON LA SPAGNA È DURA, DOBBIAMO COSTRINGERLI A DIFENDERSI"

U21: GLI AZZURRINI VOLANO IN IRLANDA. BIANCHETTI TORNA A CASA

PRESENTATE LE MAGLIE PUMA CHE GLI AZZURRI INDOSSERANNO AL MONDIALE

PANCHINA D'ORO A CONTE. "PREMIO BEARZOT" A GALDIOLO E NUOVA QUARTO

AMICHEVOLE
PROSSIMA GARA
MERCOLEDÌ 5 MARZO 2014 - MADRID
SPAGNA VS ITALIA

SPAGNA-ITALIA: NON È PIÙ POSSIBILE ACQUISTARE I BIGLIETTI

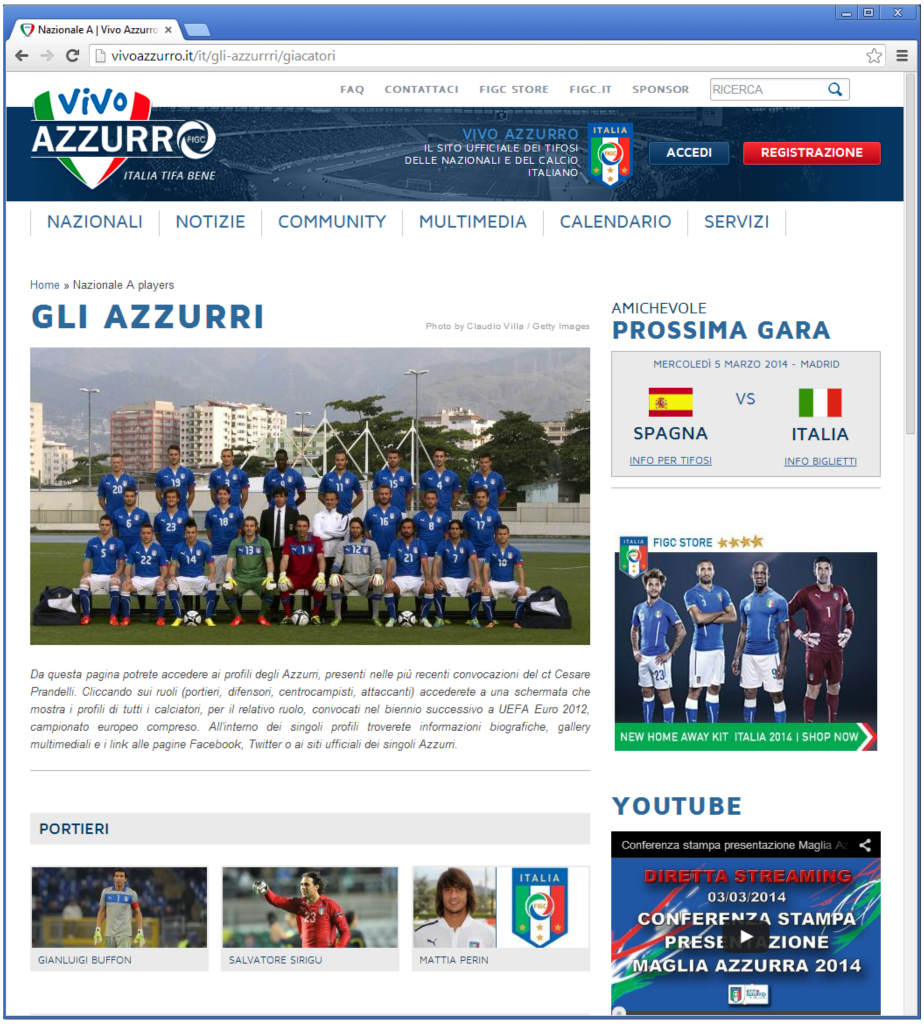
GLI AZZURRI A FIFA BRASILE 2014

NEW HOME AWAY KIT ITALIA 2014 | SHOP NOW

PREMIO NAZIONALE LETTERATURA DEL CALCIO ANTONIO GHIRELLI
SEZIONE FOTOGRAFIA
L'Italia tifa bene
PARTECIPA AL CONCORSO FOTOGRAFICO "L'ITALIA TIFA BENE"

Conferenza stampa presentazione Maglia Az
DIRETTA STREAMING 03/03/2014 CONFERENZA STAMPA PRESENTAZIONE MAGLIA AZZURRA 2014
0:00 / 1:19:50

Figure 2.1: The main page of the *vivoazzurro.it* web application.



Home » Nazionale A players

GLI AZZURRI

Photo by Claudio Villa / Getty Images

Da questa pagina potrete accedere ai profili degli Azzurri, presenti nelle più recenti convocazioni del ct Cesare Prandelli. Cliccando sui ruoli (portieri, difensori, centrocampisti, attaccanti) accederete a una schermata che mostra i profili di tutti i calciatori, per il relativo ruolo, convocati nel biennio successivo a UEFA Euro 2012, campionato europeo compreso. All'interno dei singoli profili troverete informazioni biografiche, gallery multimediali e i link alle pagine Facebook, Twitter o ai siti ufficiali dei singoli Azzurri.

PORTIERI

GIANLUIGI BUFFON | SALVATORE SIRIGU | MATTIA PERIN

AMICHEVOLE

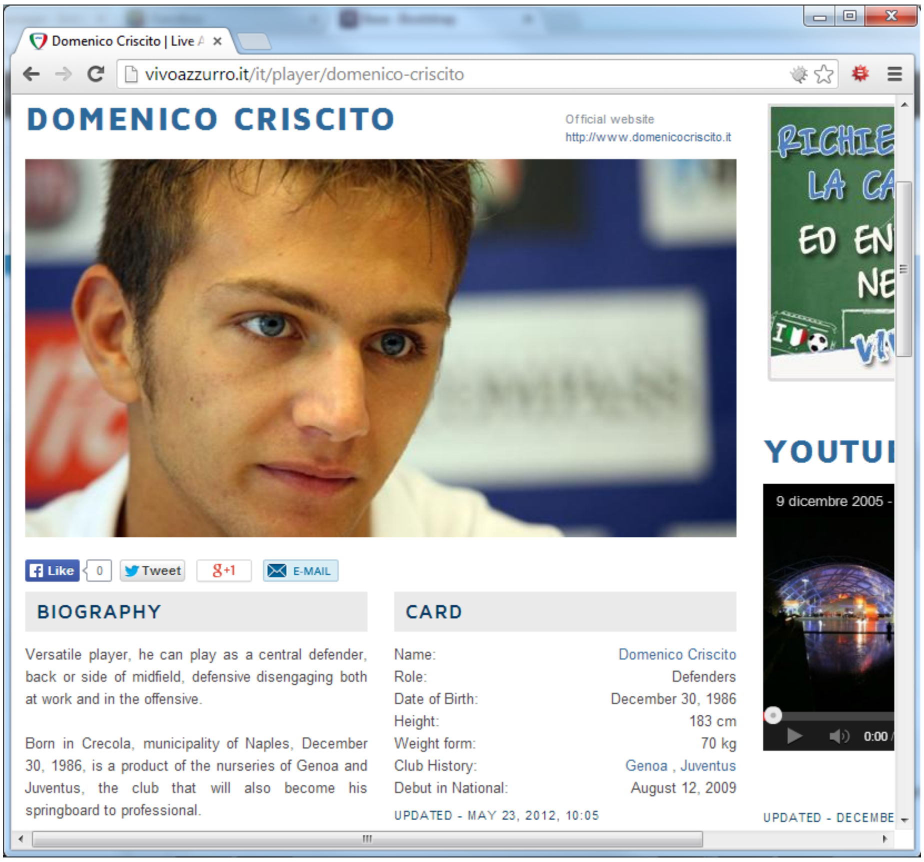
PROSSIMA GARA

MERCOLEDÌ 5 MARZO 2014 - MADRID

SPAGNA VS ITALIA

INFO PER TIFOSI | INFO BIGLIETTI

Figure 2.2: The page of the *vivoazzurro.it* web application that presents the national A team.


Domenico Criscito | Live A
vivoazzurro.it/it/player/domenico-criscito
Like
0
Tweet
g+1
E-MAIL
RICHIE LA CA ED EN NE
YOUTU
9 dicembre 2005 -
0:00

DOMENICO CRISCITO

Official website
http://www.domenicocriscito.it

BIOGRAPHY

Versatile player, he can play as a central defender, back or side of midfield, defensive disengaging both at work and in the offensive.

Born in Crecola, municipality of Naples, December 30, 1986, is a product of the nurseries of Genoa and Juventus, the club that will also become his springboard to professional.

CARD

Name:	Domenico Criscito
Role:	Defenders
Date of Birth:	December 30, 1986
Height:	183 cm
Weight form:	70 kg
Club History:	Genoa , Juventus
Debut in National:	August 12, 2009

UPDATED - MAY 23, 2012, 10:05

UPDATED - DECEMBE

Figure 2.3: The page of the *vivoazzurro.it* web application that presents a player.

2.1 Technical Overview

From a technical point of view, this example is a typical web application. The application consists of a number of pages interconnected with each other through hyperlinks. Figure 2.4 on the next page depicts an extract of some of these pages and the links between them. From this figure, it is obvious that the *Player* page is very significant, since it is interconnected with many pages. Navigation to *external* pages, i.e. pages that belong to other web applications, have been indicated with a red X. As mentioned earlier, the vivoazzurro.it web application is a web portal, thus bringing information together from diverse sources, e.g. different web applications or websites. As seen in the figure, every page is at least connected with one external page.

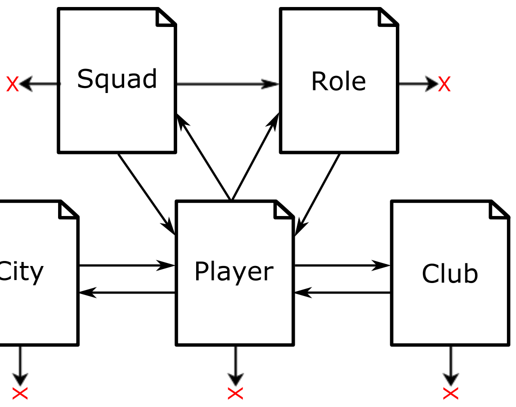


Figure 2.4: The depiction of some pages and the links between them from the *vivoazzurro.it* web application.

Examples of the web applications that the external pages belong to are *FIGC*[2], *twitter*[3] etc.

Every page of the vivoazzurro.it application is rendered using content provided by the database, which means that the application also has a data model.

2.2 Data Model in the Running Example

Figure 2.5 on the following page presents a diagram of the domain model created by us, which closely captures how the vivoazzurro.it application was originally developed. We will be using a part of this model as the data model of the running example.

As one can see from Fig. 2.5 on the next page, the `Player` class is associated with almost all of the other elements. This makes it a significant element in the domain model since, as mentioned earlier, a lot of information is provided about the players; especially about the players of the national A team. The `Player` class represents data such as name, height and biography about a player. Furthermore, it is related to a squad, a club (where the player currently is

[2]The national football federation of Italy: `http://www.figc.it`
[3]`http://twitter.com`

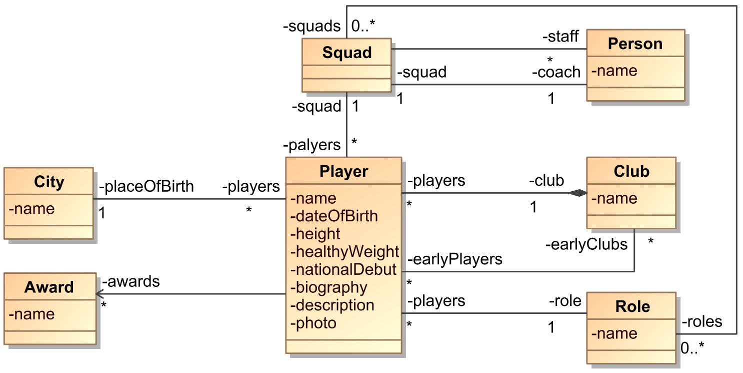


Figure 2.5: Domain model of the *vivoazzurro.it* web application.

playing in), former clubs the player has played in, the position the player plays, the awards won by the player and the city the player was born in.

The class `Squad` models a football squad, e.g. the national A team. It consists of a number of players, a coach and a number of staff. The `Role` class models the role or position a player plays. Every role has an unique name.

In this thesis, we use the Player, Squad and Role classes from the domain model shown in Fig. 2.5 in order to develop a very simple web application. In addition, we also add a `Subscription` class that will contain the e-mail addresses of fans that have subscribed to a player. The resulting data model for our application is shown in Fig. 2.6 on the next page.

2.3 Role of the Running Example

It is important to bear in mind that the example introduced above is used both for developing the domain-specific language for modeling web applications and for evaluating this language. In addition, this example is used to test the underlying tool.

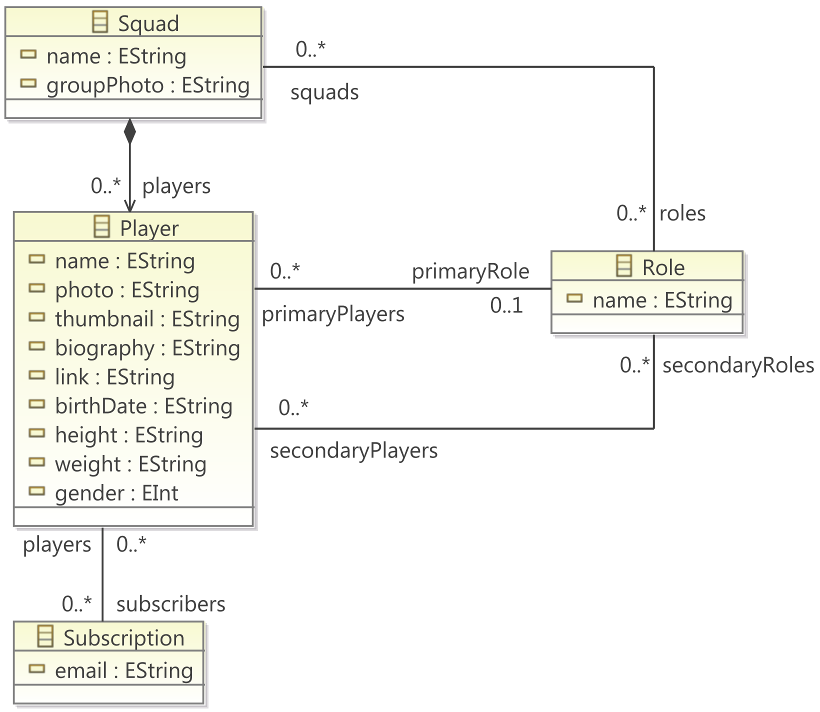


Figure 2.6: Data model of the application that is developed in this thesis.

2.4 Summary

In this chapter, we introduced the running example that is used throughout this thesis. This example is used both to design and evaluate the domain-specific language for modeling web applications.

CHAPTER 3

Background

In this chapter, we present the various concepts related to the field of model-driven software development. Moreover, we present the landscape of web applications and the state of the art of web development. Finally, a brief account on the Joomla! platform and content management systems in general is also presented.

3.1 Conceptual Background

In this section, we describe MDA and the concepts within it. Moreover, we also describe the concept of domain-specific modeling and domain-specific language development. In the following, we only cover those concepts that are essential for the development of a modeling notation and the underlying tool.

3.1.1 MDSD and MDA

As mentioned in the introduction, MDA [MM03] is one of the best known MDSD initiatives. It is proposed and maintained by the OMG (Object Management

Group) [OMG], a consortium that specifies and maintains computer industry specifications for interoperable enterprise applications. Both MDA and its related acronym MDD (Model-Driven Development) are trademarks of the OMG.

As we also mentioned in the introduction, the goal of MDA is to separate business and application logic from its underlying execution platform. A tool that implements the MDA concepts will allow developers to produce models of the application and business logic and also generate code for a target platform by means of transformations. There are two kinds of transformations defined by MDA: M2M (Model-to-Model) and M2T (Model-to-Text). These transformations are further described in the following.

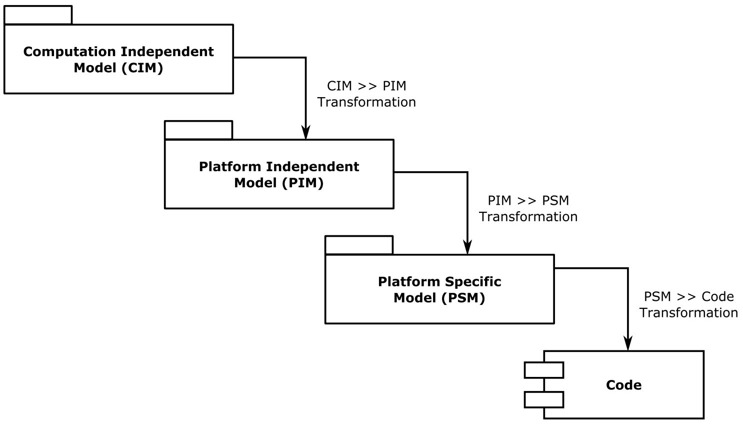
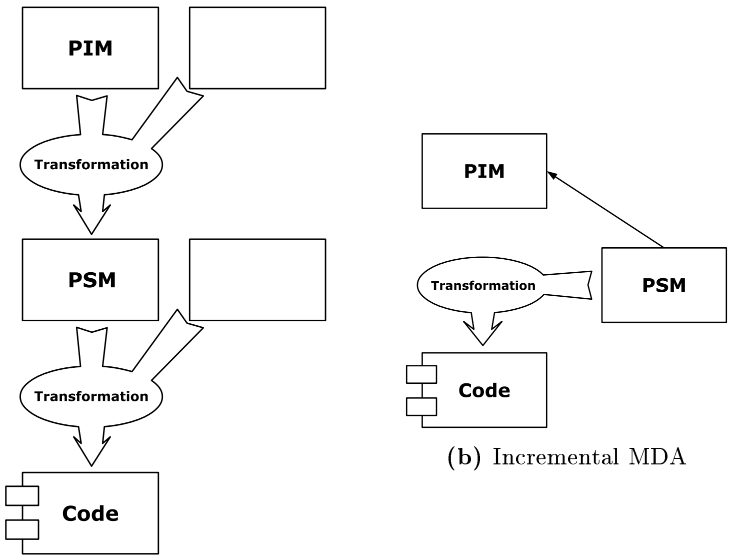

Computation Independent Model (CIM)
CIM >> PIM Transformation
Platform Independent Model (PIM)
PIM >> PSM Transformation
Platform Specific Model (PSM)
PSM >> Code Transformation
Code

Figure 3.1: The MDA Pattern

The MDA framework is organized around three level of abstractions; the so-called computation independent models (CIMs), platform independent models (PIMs) and platform specific models (PSMs) [MM03]. The models in the first abstraction level (CIM) describe the business context and business requirements for the software system. Platform independent models, on the other hand, are specifications of a system in terms of domain concepts and independent of execution platforms. Finally, the PSMs specify how the system uses a particular platform in order to realise the software system. The application's code is also considered a form of PSM, but at the lowest level of abstraction. Figure 3.1 illustrates these abstraction levels. In this figure, the transformation between models on the different level of abstraction is also illustrated. At the lowest level a PSM is transformed into code through M2T transformation. The transformations from CIM to PIM and PIM to PSM, on the other hand, are M2M transformations.



(a) Transformational MDA

(b) Incremental MDA

Figure 3.2: The two approaches for applying the MDA pattern.

There are two different approaches for applying the MDA pattern. These are depicted in Fig. 3.2. In the *transformational* MDA approach, basically, each model from a higher level of abstraction is transformed directly into a model on a lower level of abstraction. This approach is depicted in Fig. 3.2a. Here, a platform independent model and other information are combined by the transformation to produce a platform specific model, which in turn, with combination of other information, is transformed into code. This approach has a serious disadvantage when it comes to keeping the models synchronized; on each level of abstraction the model can be altered by humans. This disadvantage is addressed in *incremental* MDA which is depicted in Fig. 3.2b. In this approach the platform specific model is initialized with the platform independent model. In other words, the PSM is dependent on the PIM and any changes made to the PIM is automatically reconciled by the PSM. The PSM will also contain all the additional information needed in order to transform the PSM to code. In our approach, we have followed the incremental MDA approach.

3.1.2 Meta-Object Facility

MOF (Meta-Object Facility) is an OMG standard that enables metadata management and modeling language definition [OMG06]. It is closely related to UML (Unified Modeling Language). Since the central theme of the MOF approach to metadata management is extensibility, the aim is to provide a framework that supports any kind of metadata, and that allows new kinds to be added as required. In order to achieve this, the MOF has a layered metadata architecture. This architecture is based on the classical four layer metamodeling architecture which is not in the scope of this thesis and will not be discussed further, the interested reader is referred to [OMG06] for more details.

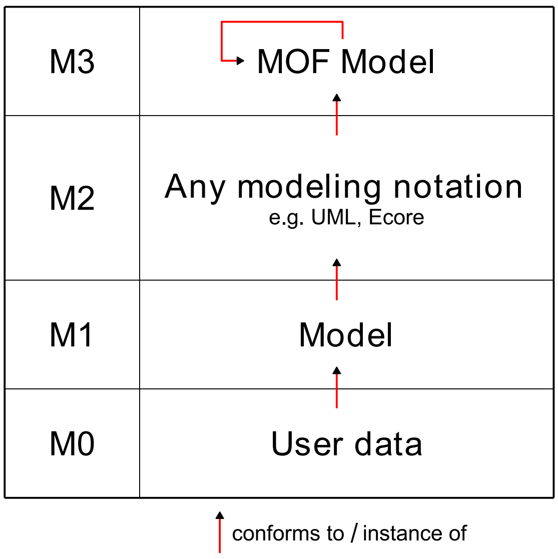
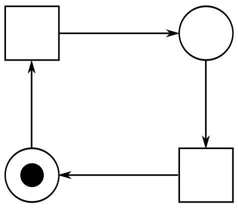


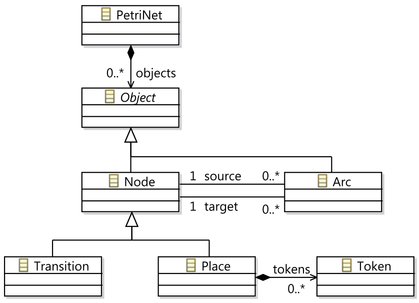
Figure 3.3: The MOF Metadata Architecture

3.1.2.1 The MOF Metadata Architecture

There are four layers in the MOF metadata architecture as mentioned above. These layers are also referred to as meta-levels. The MOF metadata architecture is also known as 3+1 MDA organisation [Béz05]. Figure 3.3 illustrates the MOF metadata architecture. Here, we can see that user data (or information) is at *M0* layer. At the *M1* layer is the model that the user data conforms to (or is instance of). This model in turn conforms to the model (or meta-model) in *M2* layer. At the top most layer, *M3* layer, the MOF's core meta-meta-model (or



(a) A simple Petri net



(b) A meta-model for Petri nets

Figure 3.4: An example of a Petri net (**a**) and a meta-model for Petri nets (**b**).

MOF Model) exists. The meta-models at the M2 layer conform to this model, while this model conforms to itself.

In order to better understand MOF metadata architecture and the layers depicted in Fig. 3.3, consider the Petri net example shown i Fig. 3.4a. In this example all of the concepts of Petri nets have been presented; the two kinds of *nodes*, i.e. *transitions* (the squares) and *places* (the circles), the *arcs* (the arrows) between these nodes and the *tokens* (the black dot). The simple Petri net in Fig. 3.4a is at the M0 layer in the MOF metadata architecture depicted in Fig. 3.3.

In Fig. 3.4b a meta-model (or domain model [Kin09]) for Petri nets is shown as Ecore model which is an EMF[1]-model. This model expresses the concepts of Petri nets as a UML class diagram. However, it must be emphasized that Ecore model is technically different from UML class diagrams, but conceptually means the same [Kin09], i.e. a lot of concepts from UML class diagrams can not be expressed with Ecore model. As seen in Fig. 3.4b, a Petri net consists of many `Objects` which are either a `Node` or an `Arc`. In addition, there are two kinds of nodes, `Transition` and `Place` where places can contain any number of `Tokens`. The simple Petri net in Fig. 3.4a is an instance of the meta-model in Fig. 3.4b. Thus, the meta-model in Fig. 3.4b is at the M1 layer in the MOF metadata architecture depicted in Fig. 3.3.

The Ecore model which is used to express the meta-model shown in Fig. 3.4b is an instance of yet another model; namely the Ecore meta-model. An excerpt of

[1]The Eclipse Modeling Framework (EMF) is one of the framework provided by the Eclipse Modeling Project (EMP) which we utilize extensively in this thesis.

this meta-model is depicted in Fig. 3.5. Notice, that the names of the concepts in this figure begins with a capital letter `E` which stands for Ecore. Furthermore, in Fig. 3.5, `EClass` corresponds to the concept of class in UML class diagrams. This concept can have any number of operations (`EOperation`) and structural features (`EStructuralFeature`). In addition, there are to kinds of structural features of the class concept; namely attributes (`EAttribute`) and references (`EReference`). The concept of operation can have any number of parameters (`EParameter`). The parameters, operations and structural features are all typed elements (`ETypedElement`), i.e. they each have a type. This type is a so-called classifier (`EClassifier`) where it can be either a data type (`EDataType`), e.g. a primitive type such as integer, string, etc., or a class. The Ecore meta-model is at M2 layer in the MOF metadata architecture depicted in Fig. 3.3.

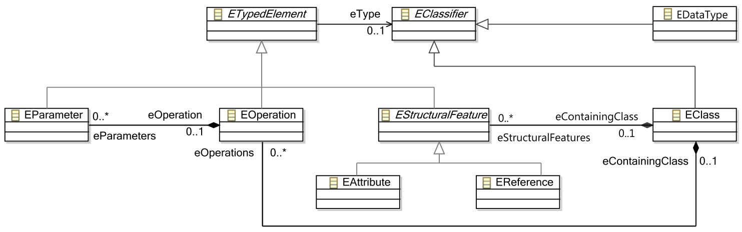


Figure 3.5: An excerpt from the meta-model of Ecore model.

By now, you might think that the Ecore meta-model would also have a meta-model (or a meta-meta-model) which will be at the M3 level. Or, you might think that the MOF Model would be this meta-meta-model that the Ecore meta-model conforms to. Actually, neither of these are the case. This is due to the fact that the meta-levels in the MOF metadata architecture are not fixed [OMG06]. There can be more or less than 4 meta-levels, depending on how the MOF is deployed. As the MOF specification suggest, meta-levels are a convention in order to understand the relationships between different kinds of data and metadata. In above example, we only have 3 meta-levels. This means that the Ecore meta-model conforms to itself, i.e. it is a meta-meta-model. In fact, the Ecore meta-model is quite closely resembled by the so-called EMOF (Essential MOF) [SBPM09, pp. 56].

3.1.2.2 MOF and UML

As mentioned above, Ecore model is technically different from UML class diagram since Ecore model can not express all the concepts of UML class diagram. It was also mentioned that EMOF closely resembles the ECore meta-model. MOF comes also in a more complex version the so-called Complete

MOF (CMOF) which captures the features that EMOF lacks. Both EMOF and CMOF refer to and use the UML standard, i.e. UML Core [OMG06].

3.1.3 Domain-Specific Modeling

In DSM (Domain-Specific Modeling), one designs and develops systems that provides the systematic use of domain-specific languages (DSLs) to represent the various facets of a system. Such languages, unlike general-purpose modeling languages, tend to support higher levels of abstraction. Moreover, DSLs are closer to the problem domain rather than to the implementation domain (the technology), i.e. a DSL follows domain abstractions and semantics. This characteristic of DSLs allows modelers to work directly with domain concepts, e.g. like the Petri net example above. By including the rules of the domain in the language as constraints, the specification of illegal or incorrect models can be disallowed in DSLs.

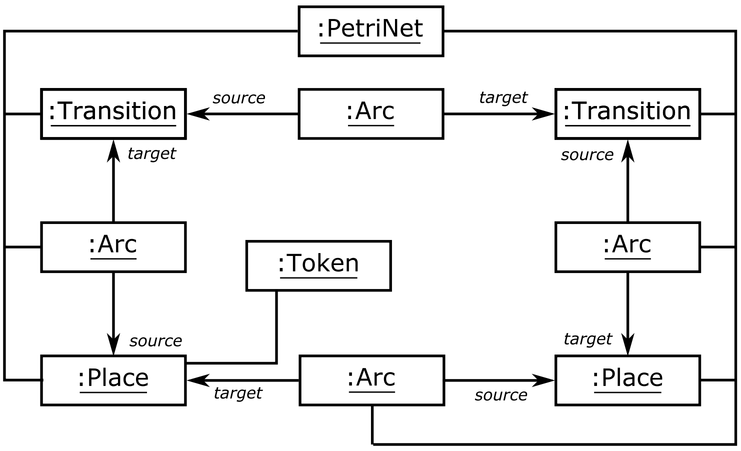


Figure 3.6: The simple Petri net in Fig. 3.4a in abstract syntax.

DSLs are a very important part of DSM. In general, at least two aspects are involved in defining a modeling language: *abstract syntax* and *concrete syntax*—let it be textual or graphical. The domain concepts and rules constitute the abstract syntax, while the notation to represent these concepts and rules constitute the concrete syntax. In the Petrinet example above, the concrete syntax (or the graphical concrete syntax in this case) is shown in Fig. 3.4a, while the corresponding abstract syntax representing the exact Petri net in Fig. 3.4a is shown in Fig. 3.6. Figure 3.6 shows an *object diagram* which shows the objects of the Petri net as instances of classes and the link between them are shown as instances of associations. The classes and associations were discussed above.

The idea of code generation, i.e. automating the creation of executable code from DSM models, is also often included in DSM. Code generation has important benefits in software development. It improves the productivity of developers, since one is free from the manual creation and maintenance of source code. Moreover, it results in quality source code free from defects and errors. Furthermore, by raising the level of abstraction emphasis is put on already familiar terminology and hides unnecessary complexity and implementation-specific details.

A DSM environment may be thought of as a meta-modeling tool, i.e. a modeling tool used to define modeling tool. Using a DSM environment the domain experts only need to specify the domain-specific constructs and rules, and the DSM environment will provide a modeling tool tailored for the target domain. The utilization of a DSM environment significantly lowers the cost of developing tools that support a DSM language. This is due to the fact that program parts such as domain-specific editors, browsers, and components are very costly to build from scratch. Examples of DSM environments include MetaEdit+ [Met] (commercial), Generic Eclipse Modeling System (GEMS) [Fouc] (open source), and Generic Modeling Environment (GME) [oE] (academic). Due to the increasing popularity of DSM, DSM frameworks have been added to existing integrated development environments. Here, we can mention Eclipse Modeling Project (EMP) and Microsoft's DSL Tools for Software Factories. This thesis utilizes a number of frameworks from EMP.

3.1.4 OMG Approach for Defining DSLs

The MDA models, i.e., CIMs, PIMs and PSMs, can be defined using modeling languages. These modeling languages can be specified using a UML- or MOF-compliant language or a language not compliant with OMG standards. However, it is strongly suggested by the OMG that the first choice is the case. This is due to the increasing need for interoperability between notations and tools. Furthermore, this facilitates and improves reuse, since due to interoperability the exchange of data and models is possible.

OMG defines three approaches for defining DSLs [MRV08]. The first approach is to develop a meta-model that is able to represent the domain concepts and rules. Basically, this approach is what we have described above in the DSM section (see section 3.1.3 on the previous page). This approach is the one followed by the CWM (Common Warehouse Metamodel) [Gro03] which is also an OMG product. We have also followed this approach in this thesis.

The second and third approaches for defining DSLs are based on extending UML; lightweight extensions as UML *profiles* and heavyweight extensions by extending the UML meta-model. These approaches are not in the scope of this thesis and will not be elaborated further. The interested reader is referred to [MRV08] for more details.

3.2 State of the Art Web Development

According to Kappel *et al.*, the landscape of web applications is divided into nine different categories [KPRR06]: document-centric, interactive, transactional, workflow-based, collaborative, portal-oriented, social web, ubiquitous, and semantic web. From the early days of web where we had static HTML pages (document-centered web applications) until the more recent web 2.0 applications (ubiquitous web applications). In between we have interactive, semantic web, portal-oriented, collaborative, workflow-based, social web and transaction-oriented web applications. The complexity of these kinds of web application varies a lot and the kinds of concern involved in the development of these applications will directly depend on the type of the application being designed and the project requirements [MRV08]. This shows how versatile the domain of web applications is, and it is very challenging to propose an approach that can capture all different kinds of web applications. Nevertheless, a Model-driven web architectural framework, the so-called WEI, has been provided by N. Moreno *et al.* (2008) [MRV08] that consists of 13 meta-models organized in three main layers; User Interface, Business Logic and Data. In this framework, each model addresses one concern depending on the kind of web application being developed. This framework is the most comprehensive work, known to us, that addresses all concerns regarding the different kinds of web applications. However, this thesis is conceptual and does not provide a concrete tool to support the framework. We will relate to this and other works later in this thesis (see section 10.1 on page 137).

In [KPRR06] the development approaches of web applications are summarized as follows:

- ad-hoc development
- development is based on knowledge and experiences of individual developers
- reuse of existing applications by means of “Copy&Paste”

Furthermore, it is mentioned that these approaches are missing methodic and sufficient documentation of design decisions. We are confident that these approaches are applied even today; especially in the development of web applications based on open source technologies.

3.2.1 User- and Data-centric Web Applications

The development of commercially motivated web applications is centered around the end-user. In the commercially motivated web applications, one endeavours to keep the user on the site as long as possible or encourage the user to purchase a good or service. In order to accomplish this, both the aesthetic aspect of the application and the flow in navigating between pages of the application should go hand in hand. Each single page of the application must provide the user with a reason to stay on the site or return as soon as possible. From the experience of the web development companies such as Peytz & Co, finding the right flow between the pages and the data needed to build these pages is the most crucial phase of the development of the application and must be separated from the aesthetic aspect of the application. Furthermore, the concerns regarding the aesthetic aspect of the application must be addressed after the right flow between the pages have been found. Our approach will aid the developers in finding the right flow between pages and the data needed to build these pages faster by increasing the level of abstraction such that domain concepts such as pages and navigation between them and their content are the primary elements of modeling.

3.3 Content Management Systems (CMSs)

Due to the increasing complexity of web applications, a number of open source web platforms have been provided by the open source community. Some of these are widely used around the world both commercially and non-commercially. One of these open source and widely used web platforms and CMS is *Joomla!*. The Joomla! framework offers three different ways of extending the Joomla! based web application; *component*, *module*, *plugin*. These are commonly call as *extensions*. A component is a complete application that is wrapped within the Joomla! platform. It can be based on the MVC (Model-View-Controller) architecture provided by the Joomla! framework, but is not limited to that. It can also provide modules and plugins. An example of a Joomla! component could be a webshop with a catalog, a customer base, a payment gateway etc. A module is a block of content or a feature from one or multiple components that can be positioned arbitrarily on a web page. This can be, for instance, a login form or a search form. A plugin is a collection of behavior triggered by different events during runtime. These events can be predefined by the Joomla! platform or defined by third-party components. The Joomla! core provides a number of extensions which are part of any installed Joomla! CMS; for instance the *content* component and its related modules and plugins for managing articles or *user* component for managing users. Third-party developers can provide their own extensions in order to address other requirements. Developing such extensions, however, is a time consuming and– due to lack of specialised tools– error-prone task. We would like to address this issue through the MDWE. Moreover, we would like to aid the development process of such extensions by generating most parts of the application.

In our approach, it will be possible to deploy web applications as Joomla! 2.5 components. The reason we have chosen Joomla! is that:

1. we have a good knowledge of the platform and have developed multiple applications for this platform.
2. due to the design of the Joomla! platform, it is possible to automatically generate code for a Joomla! extension.
3. Joomla! platform (more specifically Joomla! CMS) is one of the most wide spread and used open-source CMS system in the world.

More details about Joomla! is provided later in this thesis (see section 8.5.1 on page 117).

3.4 Summary

In this chapter, we introduced various concepts and terms that are used throughout this thesis. We introduced MDSE, MDA and other necessary concepts for developing a domain specific language. Furthermore, the landscape of web applications was outlined and the challenging nature of the domain of web applications was presented. This also covered the state of the art of web development. Lastly, a brief account on content management systems was presented, and arguements regarding the choice of Joomla! as target platform was also presented.

CHAPTER 4

Design of Web DSL

In this chapter, we present the DSL developed by us in order to model web applications on a higher level of abstraction. In addition, we also present the way this DSL has been designed and the major design decisions involved in the process.

4.1 Designing Web Model

Here, we pick up from the definition of the data model in the running example chapter (see section 2.2 on page 9) and model a simple application based on the *vivoazzurro.it* application introduced in the same chapter. The application we will model in the following consists of four pages; namely *Squad*, *Player*, *Role*, and *Player Form*. The first three pages are the ones presented in Fig. 2.4 on page 9, while the purpose of the forth page will become obvious later in this chapter. These pages are actually *page types*, since each one has many different instances. For instance, the screenshot of the page shown in figure 2.3 on page 8 represents an instance of Player page type. In other words for each specific player there exists a page instance of the Player page type. In the following, we will just use page in order to refer to a page type and page instance in order to refer to an instance of a page type.

In order to model the four pages mentioned above, we use the so called *Web model*. Using web model will allow one to create a *Website* which is composed of a number of *Pages*. It is possible to navigate between these pages by using *Links*. Each page is defined with content from the data model by using *Presentation Elements*. These elements are *Text*, *Image*, *List* etc. Moreover, a page also contains *Parameters* through which the content of the presentation elements are provided. The *Parameters* are initialized through the data model during runtime.

The web model is specified in more details in section 4.2 later in this chapter. In the following, we present some of the concepts of web model through examples.

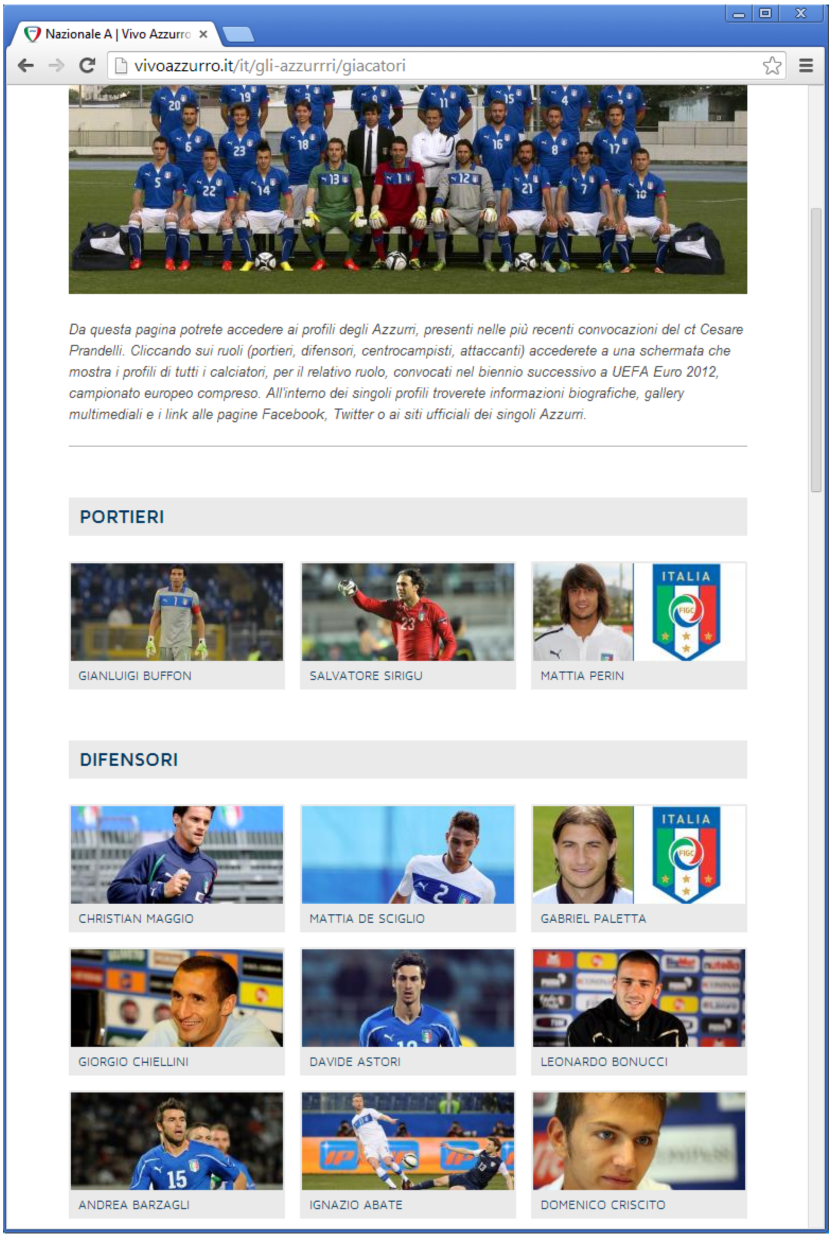
4.1.1 Modeling the Squad Page

Consider the instance of Squad page shown in Fig. 2.2 on page 7. As seen in this figure, the name of the squad (see GLI AZZURRI in the figure) and a group photo (below GLI AZZURRI) of the players and staff of the squad is displayed. Furthermore, in this figure, individual players (see below PORTIERI in the figure) of the squad are displayed. Figure 4.1 on the facing page presents another screenshot of the quad page scrolled down a bit. In this figure the players playing as goalkeepers (see PORTIERI in the figure) and defenders (see DIFENSORI in the figure) are shown. On the Squad page, players playing as midfielders and forwards are also displayed, but these are not visible in the screenshots in Fig. 2.2 and Fig. 4.1. From this, we can see that on the Squad page the players of a squad are listed by their role; two nested lists.

Figure 4.2 on page 28 presents the *graphical concrete syntax* of an example model of the page for a Squad. This figure presents the model of the first page of our simple web application. This graphical concrete syntax is elaborated on in section 4.2.1.2 on page 36. Please disregard the *markings*, e.g. ⑥, for the moment, these are used later in this chapter.

As one can see, the *Page* presented in the Fig. 4.2 models the Squad page where both squad name (see `squad.name` in the figure) and group photo (see `squad.groupPhoto` in the figure) of players and staff of the squad are displayed. Furthermore, players of a squad are listed by their role (see `List player:-Player in role.primaryPlayers` within `List role:Role in squad.roles`). In addition, each player's photo and name, and the name of the player's role is also displayed.

Let us explain some more concepts in Fig. 4.2 on page 28. Notice, that player's name (see `player.name` in the figure) and player's role (see `role.name`) are


Nazionale A | Vivo Azzurro
vivoazzurro.it/it/gli-azzurrri/giacatori

Da questa pagina potrete accedere ai profili degli Azzurri, presenti nelle più recenti convocazioni del ct Cesare Prandelli. Cliccando sui ruoli (portieri, difensori, centrocampisti, attaccanti) accederete a una schermata che mostra i profili di tutti i calciatori, per il relativo ruolo, convocati nel biennio successivo a UEFA Euro 2012, campionato europeo compreso. All'interno dei singoli profili troverete informazioni biografiche, gallery multimediali e i link alle pagine Facebook, Twitter o ai siti ufficiali dei singoli Azzurri.

PORTIERI

GIANLUIGI BUFFON
SALVATORE SIRIGU
MATTIA PERIN

DIFENSORI

CHRISTIAN MAGGIO
MATTIA DE SCIGLIO
GABRIEL PALETTA
GIORGIO CHIELLINI
DAVIDE ASTORI
LEONARDO BONUCCI
ANDREA BARZAGLI
IGNAZIO ABATE
DOMENICO CRISCITO

Figure 4.1: A list of players from the national A team presented on *vivoazzurro.it* web application.

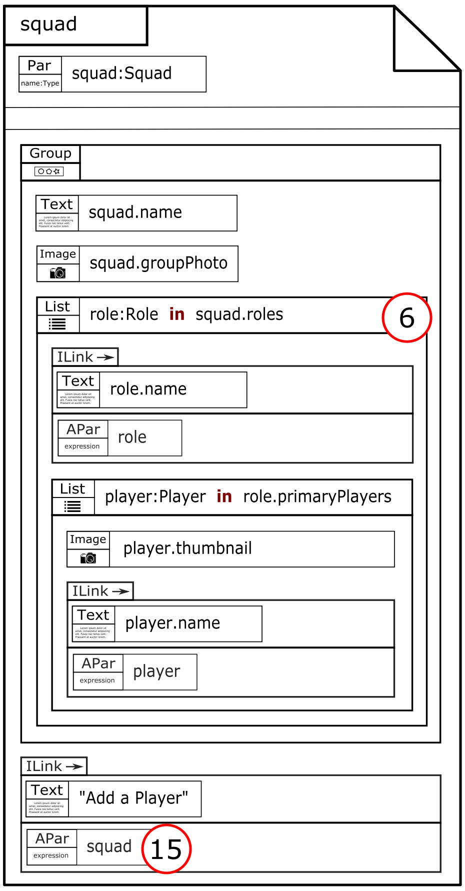


Figure 4.2: Model of the *squad* page in the graphical concrete syntax of web model.

surounded by the *Link* concept (see `ILink` $\rightarrow$), more precisely the *Internal Link* concept. This concept is used to model the navigation between the pages of the application being modeled. In this case, they model navigation to Player and Squad pages, respectively. Later in this chapter, we shall see how this navigation is modeled. As it can be seen from Fig. 4.2 on the facing page, the link concept also contains another concept, which is the so called *actual parameter* (see `player` and `role` in `APar` in the figure). This concept is used to initialize the *parameter* of the page being navigated to. The actual parameter concept amounts to arguments of functions and methods from programming languages. This is elaborated on in section 4.2.1.15 on page 41.

The web model is referencing the data model where the dynamic information of the pages is retrieved. In Fig. 4.2 on the facing page, one can see how this is done; for instance `squad.name` will provide the name of the squad. The `squad` itself is a parameter of type `Squad` (see the data model in Fig. 2.6 on page 11) which is initialized to an instance of this type, i.e. a specific squad, during runtime.

4.1.2 Modeling the Player Page

Consider the screenshot in Fig. 2.3 on page 8 which shows an instance of the actual Player page from the *vivoazzurro.it* application. In this figure, player's personal data such as name (see DOMENICO CRISCITO in the figure), a photo, biography, etc. is displayed. Furthermore, it is also possible to navigate to player's *official website*.

The web model in Fig. 4.3 on the next page presents the *Page* that models a single player view where a player is presented with various personal data displayed; similar to the page instance in Fig. 2.3. Many of the elements in this figure are similar to those in Fig. 4.2 on the facing page. However, there are some new elements in this figure. Notice, that the player's link is surounded by a so-called *external link* (see `XLink` $\rightarrow$). As the name suggests, this concept is used to model the navigation to pages residing in other web applications. In this case, a hyperlink is created to the player's page on the *vivoazzurro.it* application. Furthermore, in Fig. 4.3, a form (see `Form` in the figure) for subscribing to a specific player using e-mail address is also modeled. Such form is not included in the actual page instance shown in Fig. 2.3 on page 8. Its sole purpose is to demonstrate some of the concepts of web model. This will become obvious later in this chapter.

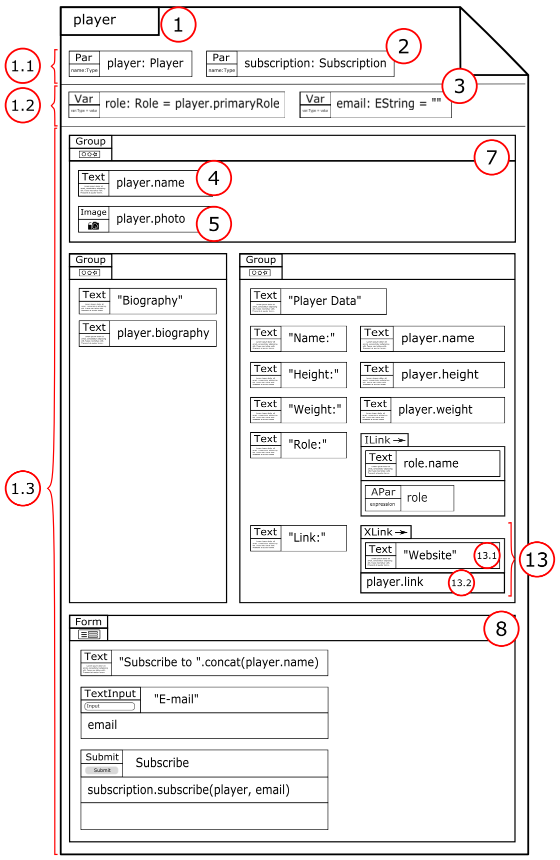


Figure 4.3: Model of the *player* page in the graphical concrete syntax of web model.

4.1.3 Modeling the Role page

Consider the instance of the Role page shown in Fig. 4.4 on the following page. In this figure, players playing as defenders (see DIFENSORI in the figure) are displayed. This list corresponds to the defenders list in Fig. 4.1 on page 27, here, however, more information is displayed about the individual player, i.e. a description of the player is provided.

The web model in Fig. 4.5 on page 33 presents the *Page* that models a single role view where a list of players playing this role (actually list of players playing this role as their primary role) is presented with their name and photo. All the elements in this model should be familiar by now; there is no new element in this model.

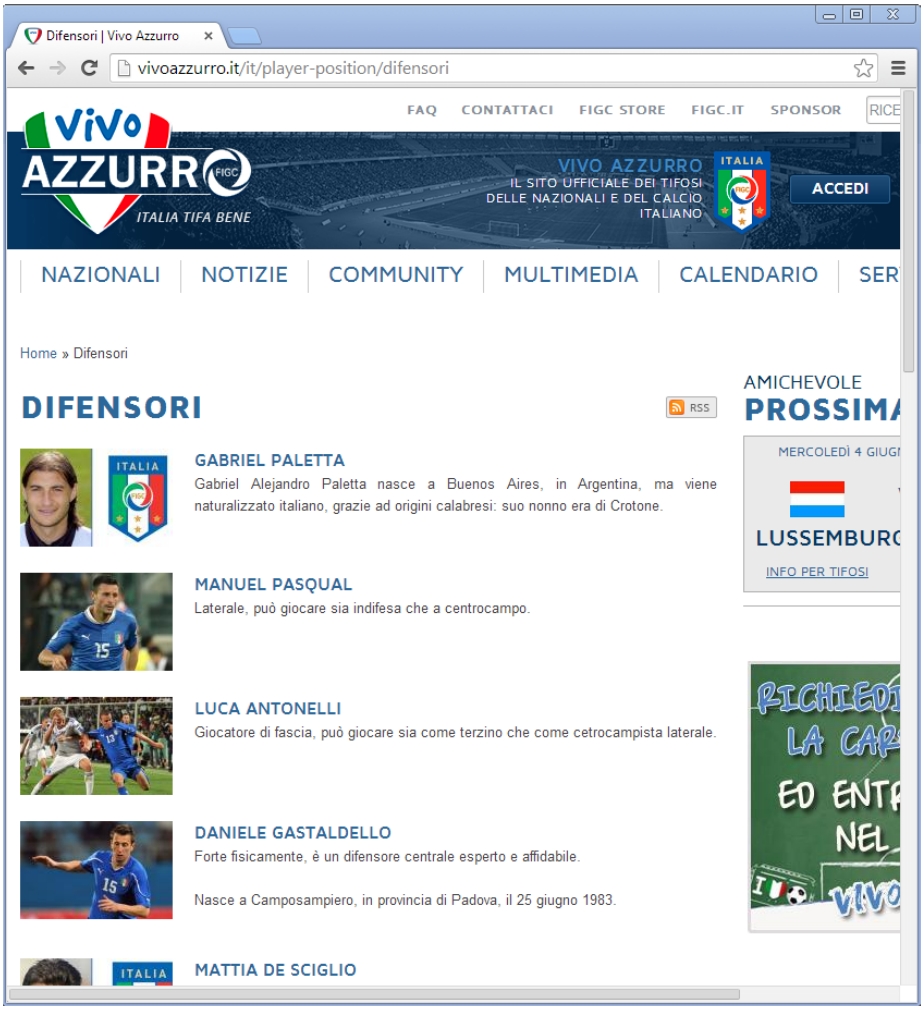
4.1.4 Modeling Form

In order to demonstrate the concept of form and the concepts related to it, we have added the *Player Form* page. The web model in Fig. 4.6 on page 34 presents the *Page* that models this form for creating and modifying an instance of `Player` type (see the data model in Fig. 2.6 on page 11). Notice that in the web model in Fig. 4.6 only player's data such as name, height, gender, primary and secondary roles, and biography are modeled, although, the `Player` type has also other data, e.g. photo. The reason behind this is to keep the model simple but in the same time demonstrate all the concepts. This model can be easily extended with remaining player's data.

4.1.5 Modeling Navigation

The user can navigate between the four pages presented above. These pages are connected with each other through the internal link concept presented in the models above. Figure 4.7 on page 35 shows these connections, i.e. navigation between the four pages. Here it is shown that from Squad page it is possible to navigate to all other three pages. From the Player Form page it is only possible to navigate to the Squad page. Furthermore, it is possible to navigate to the Role page from the Player page and vice versa.

By now, we have a complete model of a simple application that has four pages. It is possible to generate code for this application in a fully automatic manner.



Difensori | Vivo Azzurro

vivoazzurro.it/it/player-position/difensori

FAQ CONTATTACI FIGC STORE FIGC.IT SPONSOR RICE

VIVO AZZURRO ITALIA TIFA BENE

VIVO AZZURRO
IL SITO UFFICIALE DEI TIFOSI DELLE NAZIONALI E DEL CALCIO ITALIANO

ACCEDI

NAZIONALI | NOTIZIE | COMMUNITY | MULTIMEDIA | CALENDARIO | SER

Home » Difensori

DIFENSORI

RSS

GABRIEL PALETTA
Gabriel Alejandro Paletta nasce a Buenos Aires, in Argentina, ma viene naturalizzato italiano, grazie ad origini calabresi: suo nonno era di Crotone.

MANUEL PASQUAL
Laterale, può giocare sia indifesa che a centrocampo.

LUCA ANTONELLI
Giocatore di fascia, può giocare sia come terzino che come cetrocampista laterale.

DANIELE GASTALDELLO
Forte fisicamente, è un difensore centrale esperto e affidabile.

Nasce a Camposampiero, in provincia di Padova, il 25 giugno 1983.

MATTIA DE SCIGLIO

AMICHEVOLE
PROSSIMA

MERCOLEDÌ 4 GIUGI

LUSSEMBURG

INFO PER TIFOSI

Figure 4.4: An instance of the Role page displaying list of players playing as defenders. This screenshot is from the *vivoazzurro.it* web application.

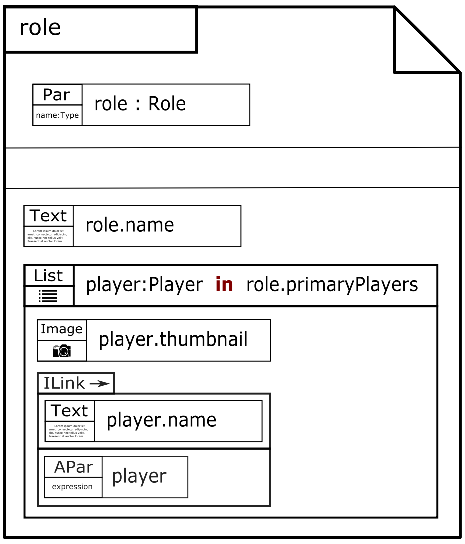


Figure 4.5: Model of the *role* page in the graphical concrete syntax of web model.

This will be covered later in this thesis. In the following, we specify the web model and the concepts within it.

4.2 Specification of Web Model

Web model, which is a platform independent model in terms of MDA, models the concepts within the domain of web applications; we will refer to this as *main concepts.* The main concepts together with other concepts (*expressions and variables*) form the elements of web model. Like any other model, web model has also a meta-model. An overview of this meta-model is presented in Fig. 4.9 on page 50 (main concepts) and Fig. 4.13 on page 54 (expressions and variables). The figures will be elaborated later. Here, we specify the main concepts and expressions and variables concepts.

Main concepts of web model are concepts such as *page*, *link*, *form* etc. Expressions and variables are used to provide the dynamically generated information (content) of various main concepts. In the following, we specify these concepts in more details.

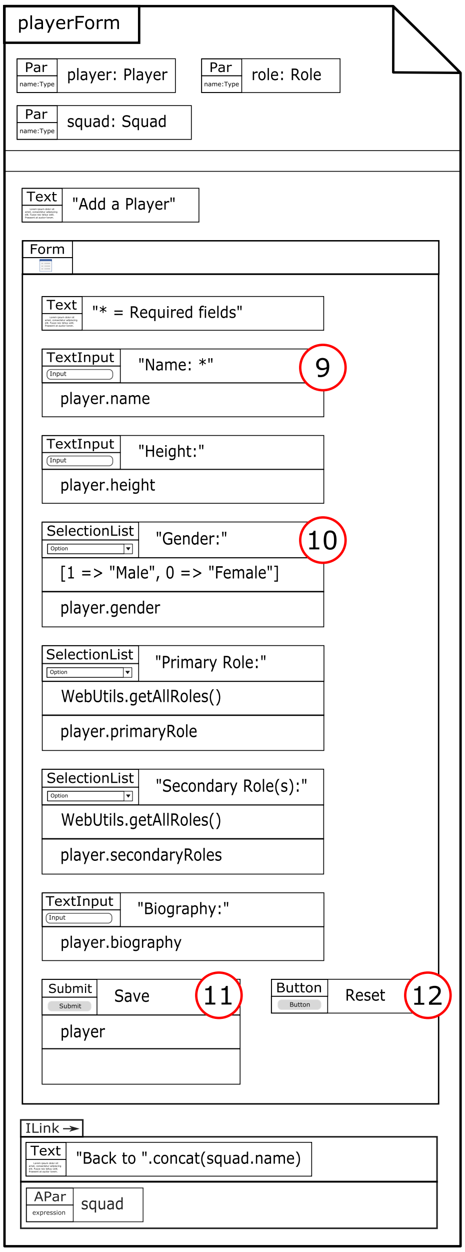

playerForm
Par name:Type player: Player
Par name:Type role: Role
Par name:Type squad: Squad
Text "Add a Player"
Form
Text "* = Required fields"
TextInput Input "Name: *"
player.name
9
TextInput Input "Height:"
player.height
SelectionList Option "Gender:"
[1 => "Male", 0 => "Female"]
player.gender
10
SelectionList Option "Primary Role:"
WebUtils.getAllRoles()
player.primaryRole
SelectionList Option "Secondary Role(s):"
WebUtils.getAllRoles()
player.secondaryRoles
TextInput Input "Biography:"
player.biography
Submit Submit Save
player
11
Button Button Reset
12
ILink
Text "Back to ".concat(squad.name)
APar expression squad

Figure 4.6: Model of the *playerForm* page in the graphical concrete syntax of web model.

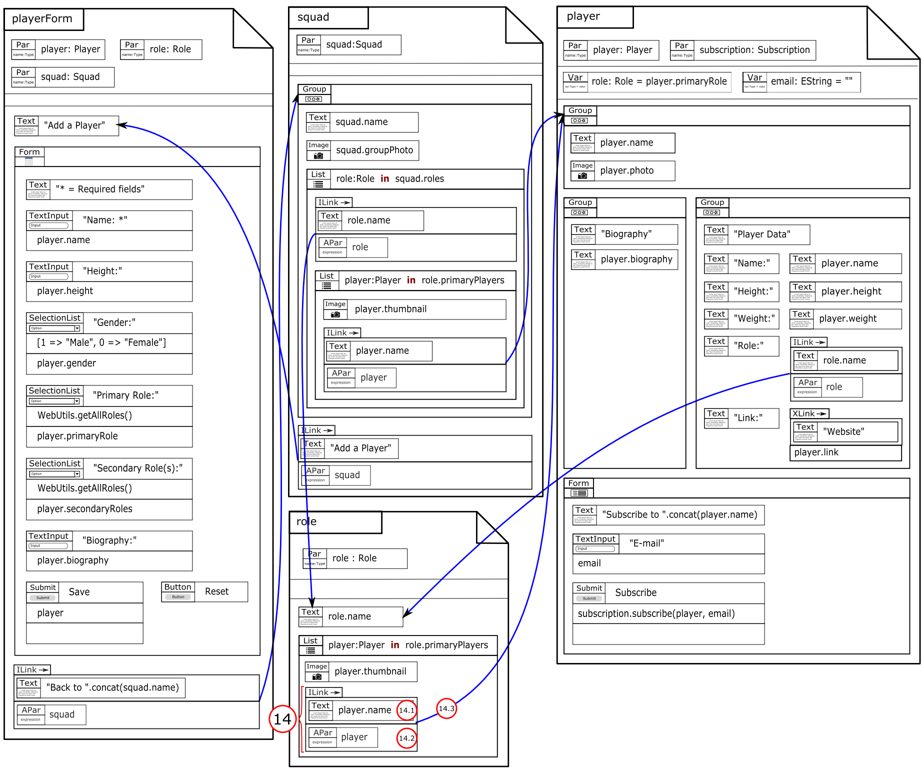


Figure 4.7: Navigation between the four pages in the simple web application.

4.2.1 Main Concepts

As mentioned above, the main concepts of web model are concepts within the domain of web application. These concepts have a graphical concrete syntax which we already have seen in the running example above (see Fig. 4.7 on the preceding page for instance).

4.2.1.1 Website

The concept of *Website* is the root concept in web models. This concept represents a web application. A web application consists of a collection of pages and the navigation between them. One of these pages is marked as the initial page, i.e. the *home* page of the website. The pages of the web application are specified below. The website concept does not have a graphical concrete syntax. It is a container for everything in the application, i.e. a canvas.

4.2.1.2 Page

The *Page* concept represents a type of web page. Examples of this concept is given in the running example above (see Fig. 4.3 on page 30). The graphical concrete syntax of this concept is shown in Fig. 4.3 marked with $\textcircled{1}$. As shown in this figure, a page has a name and is divided into three compartments; marked with $\textcircled{1.1}$, $\textcircled{1.2}$, and $\textcircled{1.3}$ in the figure. These compartments contain the *parameters* (in $\textcircled{1.1}$), *variables* (in $\textcircled{1.2}$), and *page elements* (in $\textcircled{1.3}$), respectively.

A page can have multiple parameters. The purpose of these parameters is to provide access to data. These parameters can be thought of as *formal parameters* of a function/method. For instance the function $f(x) = x^2$ has one parameter (or formal parameter) x. The parameters of the page are initialized through the *actual parameters* of the *Link* concept. In the function call $f(42)$ the value 42 is the argument (or actual parameter) of the function. This value initializes the parameter x in f. Likewise, the actual parameters of a link initialize the parameters of a page. Examples of this was given in the running example in the navigation between the four pages of the application (see figure 4.7). For instance the link from squad page to player page, in figure 4.7, have an actual parameter *player* (a specific player in the list) that will initialize the *player* parameter in the player page (with that specific player). The parameters and the concepts related to it, are elaborated on later.

As mentioned above, beside parameters, a page can have *variables*. Examples of variables are also given in the running example (see `role:Role = player.-primaryRole` in Fig. 4.3). Variables can be used to define values by using the parameters or data types such as string and integer. We will elaborate on variables later in this chapter.

As was mentioned above, the page concept also consists of several elements (page elements) which can be either *presentation* or *navigation* elements. These concepts, unlike parameters and variables, are the visual elements on a page; they display the content (of the page) in various forms. In addition, presentation elements together with navigation elements, unlike parameters and variables, are the main concepts of web models, while parameters and variables are a part of the expressions and variables concepts. Presentation elements are concepts such as *text*, *image*, and *list*, while navigation elements are concepts such as the *internal link* and *external link*. In the following, we first elaborate the page elements. Next, the expressions and variables are specified in more details.

4.2.1.3 Text

The text concept, as the name suggests, models the presentation of text on a page. The text can be either *static* or *dynamic*. The content of static text does not change and is provided during modeling while the content of dynamic text is provided during runtime and changes according to the value of the provided expression. The graphical concrete syntax of this concept is marked with ④ in Fig. 4.3 on page 30. We have already seen plenty of examples of this concept in the running example. For instance in Fig. 4.3, text such as `"Name:"` is static, while text such as `player.name` is dynamic; the first will never change while the second depends on the value of the expression which in this case is the name of a player which is a parameter of this page.

4.2.1.4 Image

The image concept models the presentation of pictures on a page. The graphical concrete syntax of this concept is marked with ⑤ in Fig. 4.3 on page 30. Examples of this concept are also given in the running example. For instance, the player page in Fig. 4.3 contains an example of this concept where a player's photo (`player.photo`) is displayed. The expression `player.photo` specifies the *source* of the image. Like the text concept, images can be *static* or *dynamic*. The source of the dynamic images is provided during runtime while the source of static images is provided during modeling and does not change during runtime. The above example is an example of dynamic image. In the running example,

an example of static image is not given. However, an example of this would be, if we added an image with the following source to the player page:
`"C:\images\banners\players.png"`
This image will be present on any instance of the player page.

The source of the image, can be *referenced* in which case the image will not be a part of the application (it will exist in another application) and will be referenced through a URL. In the default case the images are assumed residing within the application. As an example, we can change the source in above static image to
`"http://www.domain.com/banners/players.png"`
This will result in a static image with referenced source. The other case, i.e. a dynamic image with referenced source, is also possible.

4.2.1.5 List

In order to present a collection of items on a page, the *List* concept is used. The graphical concrete syntax of this concept is marked with ⑥ in Fig. 4.2 on page 28. Various examples of this concept is given in the running example (see Fig. 4.2 and 4.5 on page 33). This concept consists of a *collection* of items of the same type (e.g. `squad.roles`), a *variable* (e.g. `role:Role`), and a number of presentation and navigation elements. Technically, this concept is similar to the *for/while* loop in programming languages such as Java. In the list on the role page in Fig. 4.5 on page 33, we can see that the primary players associated with that role are iterated and their photo (as thumbnail) and name are displayed.

4.2.1.6 Form

Another main concept of web models and a page element, is the concept of *Form*. In order to model user interaction with the application, this concept is used. The graphical concrete syntax of this concept is marked with ⑧ in Fig. 4.3 on page 30. We have seen two examples of this concept in the running example, namely one on the player page (see Fig. 4.3) and one in the player form page (see Fig. 4.6 on page 34). The form concept can contain any page element. In addition, the form concept, consists of several graphical user interface (GUI) elements (*form elements*). In the following, these GUI elements are specified in more details.

4.2.1.7 Text Input

In Fig. 4.6 on page 34, an example of various form elements is given. We have for instance examples of the *Text Input* concept, the *Selection List* concept, the *Submit* concept, and the `Button` concept. The text input concept is used to model the user input as text. The graphical concrete syntax of this concept is marked with ⑨ in Fig. 4.6 on page 34. As seen in this figure, this concept has a *label* (e.g. `"Name:"`) and an expression specifying its *value* (e.g. `player.name`). The legal expressions for *value* is either a variable expression or a structural expression. These kinds of expressions are specified later (see section 4.2.2.2 on page 43).

The text input concept, can be a *text area*, in which case it is a text box that allows input of multiple lines of text, where in the default case it is a text box that allows input of a single line of text. Furthermore, this concept can be a *password*, in which case the characters typed in are masked with * (asterisk symbol).

4.2.1.8 Selection List

The selection list concept is used for presenting the user with several options where one or multiple of these options can be selected by the user. The graphical concrete syntax of this concept is marked with ⑩ in Fig. 4.6 on page 34. Like the text input concept, this concept has a label (e.g. `"Primary Role:"`) and an expression specifying its *value* (e.g. `player.primaryRole`). Furthermore, this concept has also *options* which are provided by an expression, e.g. `WebUtils.getAllRoles()` or `[1 => "Male", 0 => "Female"]`. The type of such expression must be a collection of some sort, i.e. a static list such as `[1 => "Male", 0 => "Female"]` or a dynamic list provided by an operation such as `WebUtils.getAllRoles()`.

The selection list concept can be rendered as either checkboxes, radio buttons or a drop-down list. In the case of radio buttons, only a single option would be selectable.

4.2.1.9 Submit

The concept of submit represents a number of actions that the user can trigger in order to submit the form. The graphical concrete syntax of this concept is marked with ⑪ in Fig. 4.6 on page 34. As seen in this figure, the submit concept has a *value* (e.g. `Save`) which is the text displayed to the user. Furthermore, this concept has a *performer* and a *validator*. The later can be any operation from a class in the data model, e.g. `player.validate(newPlayer)`. The performer, which executes the action, will be specified in the context of *standard* and *custom* submit actions.

4.2.1.10 Custom Submit

The custom submit action is used to model user-defined actions for handling form submit. The graphical concrete syntax of this concept is similar to the submit concept. The *performer* of this concept must be an operation from a class in the data model, e.g. `subscription.subscribe(player, email)`. We have already seen an example of this concept on the player page (see Fig. 4.3 on page 30).

4.2.1.11 Standard Submit – Save, Reset and Cancel

Three standard submit actions are provided in web model, namely *Save*, *Reset*, and *Cancel*. These have the same graphical concrete syntax as the submit concept. However, the value of the submit concept corresponds to the type of submit action, i.e. the value of the submit concept in the case of save submit action is `Save` and similarly in the case of reset and cancel. Furthermore, the *performer* of these submit actions must be an object, e.g. `player`.

4.2.1.12 Button

The final form element is the concept of button. The graphical concrete syntax of this concept is marked with ⑫ in Fig. 4.6 on page 34. This concept, similar to the submit concept, has a value, e.g. `Reset`. Moreover, there are three types of buttons, a regular button, a submit button and reset button; these are the equivalent of three types of buttons specified in HTML (Hyper Text Markup Language). Thus, the submit button and reset button differ from submit concept defined above, since submit concepts such as save, reset or cancel have actions attached to them while submit button or reset button do not.

4.2.1.13 External Link

As mentioned above, there are also navigation elements among page elements. The concept of link, which is a navigation element, models the navigation between pages. We distinguish between two kinds of links, *internal* and *external*. The later one models navigation between pages of two different web applications. We have one example of this concept in the player page in the running example (see Fig. 4.3 on page 30). Here, the external link will redirect the user to the website of the player which is another application[1]. The graphical concrete syntax of the external link concept is marked with 13 in Fig. 4.3 on page 30. As seen in this figure, the external link concept contains a presentation element (marked with 13.1 in the figure) as the source (e.g. a text element) and an expression (marked with 13.2 in the figure) which specifies the target (e.g. `player.link`).

4.2.1.14 Internal Link

The internal link concept, unlike external link, only models navigation between pages that are defined within the same application. This concept consists of a source and a target where both are *presentation* elements within the same page or two different pages. We have seen various examples of this concept in the running example (see figure 4.7 on page 35). The graphical concrete syntax of this concept is marked with 14 14 in Fig. 4.7 on page 35. While the source of this concept is defined the same way as the external link concept (this is marked with 14.1 in Fig. 4.7), the target of this concept is defined by an arrow (see 14.3 in Fig. 4.7 on page 35). This concept also contains actual parameters (see 14.2 in Fig. 4.7) which were described above together with parameters (see section 4.2.1.2 on page 36). The actual parameter concept is specified in the following, while the parameter concept is specified in section 4.2.2.

4.2.1.15 Actual Parameter

The actual parameter concept is used to initialize the parameter of the page being navigated to. The graphical concrete syntax of this concept is marked with 15 in Fig. 4.2 on page 28. This concept contains an expression which must express a parameter or variable in the page. This expression will initialize

[1]This is actually the original page of the player on the *vivoazzurro.it* web application.

a parameter of the same name in the target page, e.g. the actual parameter `player` in Fig. 4.7 on page 35 (see 14.2) will initialize the `player:Player` parameter on the player page.

4.2.1.16 Group

The final as well as one of the main concepts of web models is the *Group* concept. As the name suggests, this concept is used to group page elements. Thus, this concept can contain any page element, also itself. The graphical concrete syntax of this concept is marked with 7 in Fig. 4.3 on page 30. We have seen various examples of this concept in the running example above (see Fig. 4.2 on page 28 for instance).

4.2.1.17 Other Concepts

It is not possible to model any kind of web application using only the concepts specified above. For instance, the concept of *menu* is missing. Such concepts are considered as future work. In the moment, such concepts must be implemented manually or built through the CMS such as Joomla!.

4.2.2 Expressions and Variables

As mentioned above, expressions and variables are used to provide the data (content) of various main concepts. Various elements of the page need some data in order to render an instance of a page, e.g. "same page" for different players. This data is extracted from the data model. In order to express which part of the data model should be accessed and which values a page element should have, the concept of expression is used. This concept makes it possible to express various values by combining explicit values, constants, variables and operations. We have already discussed the purpose of variables and parameters in a page. In the following, we will specify these in more details together with expressions.

4.2.2.1 Parameters and Variables

Unlike expressions, the parameters and variables concepts have a graphical concrete syntax. In figure 4.3 on page 30 the graphical concrete syntax of the parameter concept is marked with ②, while the graphical concrete syntax of the variable concept is marked with ③. From the syntactical point of view both parameters and variables are declared the same way. They both have the same syntax, i.e., `variableName:variableType`. Their difference lies in the initialization. The initialization of parameters and their purpose was presented above; parameters are initialized by the invoker of the page. Variables, on the other hand, are initialized through an expression, i.e., with the following syntax: `variableName:variableType = expression`. An example of this could be `role:Role = player.primaryRole` which is used on the player page (see Fig. 4.3).

4.2.2.2 Expressions

As mentioned above, the expressions concepts do not have a graphical concrete syntax. Instead, they have *textual concrete syntax* (or just syntax). In the following, we specify the various kinds of expressions and their syntax.

Constant Expressions The basic kinds of expressions are constant expressions. These are a constant value of either integer, real, string or boolean. In the running example, we have only examples of string constant, e.g. `"Name:"` in figure 4.2 on page 28. The syntax of string is `"string"`; it is similar to many other programming languages such as Java. Integer and real constants can be expressed as 1, 42, 0.42, or .42. The boolean constants are *true* and $false$. Since we use EMF-models the types of these constants must be specified as `EInt` for integer, `EString` for string, and `EFloat` or `EDouble` for real numbers.

Variable Expressions Another basic expression is the use of a variable or parameter (*variable expression*). For instance in `player.name` (see Fig. 4.2 on page 28) `player` is an example of variable expression which is declared as a parameter in this case. The variable (or parameter) in any valid variable expression must be declared before.

Operation Expressions Operation expressions are yet another kind of expressions. We distinguish between two types of operation expressions, namely *basic operation* and *property operation* expressions. In the following, we specify each of these types of expressions.

Basic Operations The basic operation expressions (or just basic operation) consists of four kinds of operations. These operations are *string*, *arithmetic*, *comparison* and *boolean*. In the following, each of these operations and their operators are specified.

String Operations We have already seen an example of a string operation in the running example. The expression:
`"Subscribe to ".concat(player.name)`
is an example of a string operation expression, in this case the operator is the string *concatenation*. Another string operator is the *length* operator which has the same syntax, e.g. `"some value".length`. The length and the concatenation operators are the only built-in string operators.

Arithmetic Operations In the running example, there are no examples of arithmetic operations. These operations cover the usual mathematical operations, *addition* (e.g. $40+2$), *subtraction* (e.g. $44-2$), *multiplication* (e.g. $21*2$), *division* (e.g. $84/2$). Furthermore, we have arithmetic negation, e.g. -42. All these operators are supported. These operations can be performed both with integer and floating point numbers.

Comparison Operations There are no examples of the comparison operations in the running example either. Like arithmetic operations, these operations cover the usual mathematical operations, *less than* (e.g. $2 < 42$), *greater than* (e.g. $42 > 2$), *less than or equal* (e.g. $2 \leq 42$), *greater than or equal* (e.g. $42 \geq 2$), *equal* (e.g. $42 = 42$), and *not equal* (e.g. $2 \neq 42$). All these operators are supported. However, the signs used for less than or equal, greater than or equal and not equal operators are `<=`, `>=`, and `!=`, respectively, instead of the usual mathematical notation. This is similar to other programming languages such as Java or PHP.

Boolean Operations There are no examples of the boolean operations in the running example either. These operations cover the usual logical operations, *conjunction* (e.g. $P \land Q$) and *disjunction* (e.g. $P \lor Q$). Furthermore, we have boolean negation, e.g. $\neg P$. All these operators are supported. However, the signs used for all three operators are not the usual operator signs shown above. For conjunction &&, disjunction ||, and negation ! is used.

Property Operations In the running example, we have seen examples of expressions like `player.name` (see Fig. 4.2 on page 28), `subscription.subscribe(player, email)` (see Fig. 4.3 on page 30), and `WebUtils.getAllRoles()` (see Fig. 4.6 on page 34). These are examples of property operation expressions. These expressions are used in order to access the properties, i.e. structural features or operations, of classes in the data model. Each of the above examples represent one kind of property operation expression; there are three kinds. In the following, we specify these three kinds in more details.

Structural Expression The `player.name` expression represents an example of a *structural expression* which is one of the three kinds of property operation expressions. This expression expresses calling of structural features, i.e. an attribute or a reference, of a class from the data model. An attribute can be used in forms, e.g. `player.name`. The syntax of structural expression is:
`variable.identifier`,
where `variable` is a variable expression, which we specified above, and `identifier` is the name of the structural feature of a class in the data model.

Classifier Operation The `subscription.subscribe(player, email)` expression represents an example of *classifier operation* which is also a kind of property operation expression. This expression expresses call of an operation from a class in the data model. The syntax of classifier operation is similar to that of structural expression. However, the classifier operation must be specified with parenthesis and might have arguments. For the sake of completeness, we have stated the syntax below.
`variable.identifier([argument1[,argument2[,...]]])`.

Web Utilities The third and last kind of property operation is the so called *web utilities* expressions which is used to call framework operations. An example of this expression, which was also mentioned above, is `WebUtils.getAllRoles()` which expresses call of a framework operation `getAllRoles()` which will retrieve all instances of the class `Role`. The syntax of this expression is `WebUtils.getAll<class-name>()` where `<class-name>` is the name of a class from the data model.

List Expression The *List* expression is the final kind of expression supported by Welipse. We distinguish between two kinds of lists, namely *simple lists* and *associative lists*. We have already seen an example of the later kind (see Fig. 4.6 on page 34). On the player form page in the running example this expression is used as options of the selection list for specifying the gender of a player, i.e. `[1 => "Man", 0 => "Woman"]`[2], is an example of an associative list. The syntax of associative lists is `[key => value]`, while the syntax of simple lists is just `[value]`. The list expression is meant to be used as the options expression of the selection list concept (see section 4.2.1.8 on page 39) in order to define the options of this concept. The simple list expression can be used if no distinction is made between the value of the option displayed and the value of the option stored. For instance, if the type of `gender` attribute of the `Player` class was string, we could use `["Man", "Woman"]` in the form in Fig. 4.6. However, since the type of `gender` attribute is integer, we have used an associative list. This way the *keys* are stored and *values* are displayed in the form.

4.2.2.3 Syntax of Expressions

So far, we have presented the syntax of expressions through examples or informally for each kind of expression. In the following, we present the syntax of expressions in more formal manner.The following context-free grammar defines the syntax of the expressions using the EBNF (Extended BNF) syntactic metalanguage [ISO96].

```
expression = constant
           | variable-use
           | operation
           | classifier-operation
           | structural-expression
           | list-expression
           | webutils-expression
           | '(' expression ')';

constant = ['+'|'-'] number
         | string
         | boolean;

number = integer
       | floating-point-number;

floating-point-number = integer '.' integer
                      | '.' integer
                      | integer '.';
```

[2]This syntax is similar to the one used for defining associative arrays in PHP.

```
string = '"' {letter
             | decimal-digit
             | other-character
             | space-character} '"';

boolean = 'true'
        | 'false';

variable-use = identifier;

identifier = (letter | '_') {letter
                            | '_'
                            | '$'
                            | decimal-digit};

operation = arithmetic-operation
          | comparison-operation
          | boolean-operation
          | string-operation;

arithmetic-operation = expression ('+'
                                 | '-'
                                 | '*'
                                 | '/') expression;

comparison-operation = expression ('<'
                                 | '>'
                                 | '<='
                                 | '>='
                                 | '=='
                                 | '!=') expression;

boolean-operation = ['!'] expression
                  | expression ('&&'
                             | '||') expression;

string-operation = expression '.' 'length'
                 | expression '.' 'concat'
                 '(' expression ')';

classifier-operation = structural-expression '('
                     [expression {',' expression}]
                     ')';
```

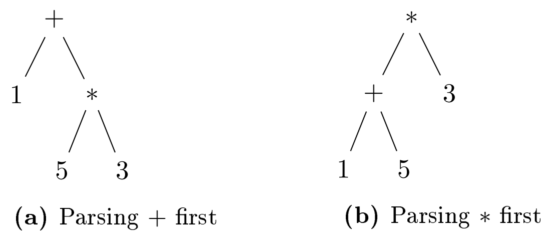


Figure 4.8: Two different parse trees for parsing expression $1 + 5 * 3$ with unknown operators precedence.

```
structural-expression = variable-use '.' identifier;

list-expression = '[' [list-element
                        {',' list-element}]
                  ']';

list-element = expression
             | expression '=>' expression;

webutils-expression = 'WebUtils' '.' identifier '(' ')';
```

The blue color in the above grammar indicates the tokens (terminals) of the expressions language. The gray color, on the other hand, indicates non-terminal symbols defined in [ISO96]. In the above grammar, we have omitted commas for the sake of readability.

Operator Precedence Notice that both arithmetic, comparison, boolean, and string rules are ambiguous as they are presented in the above grammar. These rules do not specify the precedence of the operators. For example, the expression $1 + 5 * 3$ can be parsed in to different way as shown in Fig. 4.8. The correct parse tree for this expression is the one presented in Fig. 4.8b. This is due to the higher precedence of the multiplication operator compared to the addition operator. Table 4.1 on the facing page lists all the operators in order of precedence, with the highest-precedence ones at the top.

In order to force precedence, parentheses may be used which is also presented in the above grammar (see the `expression` rule). For instance, using parentheses we can force the above expression, , i.e. $(1 + 5) * 3 = 18$, to be parsed as shown in Fig. 4.8a.

Operator name	Operator Symbol
Boolean negation	`!`
Arithmetic negation	`-`
Multiplication and division	`* /`
Addition and subtraction	`+ -`
Less than, greater than, less than or equal, and greater than or equal	`< > <= >=`
Equal and not equal	`== !=`
Conjunction	`&&`
Disjunction	`||`
String concatenation	`.concat()`
String length	`.length`

Table 4.1: Precedence of operators in the expressions language. The operator with the highest precedence is presented at the top of the table.

4.3 A DSL for Modeling Web Applications

The concepts described above form a DSL for web applications. The meta-model of this DSL is presented in Fig. 4.9 on the following page (main concepts) and Fig. 4.13 on page 54 (expressions and variables). In the following, we discuss these figures very briefly and provide more details about them in section 4.4 on page 55. We begin with Fig. 4.9 on the next page. Notice, that in this figure the `VariableInitialization`, `VariableDeclaration`, `Parameter`, and `Expression` classes, which are also shaded in the figure, belong to expressions and variables. Their details are presented in figure 4.13 on page 54 which we will touch upon in a little while.

Figure 4.9 should be read downward from the class `Website` which represents the website concept. This figure (Fig. 4.9) is divided into three figures where each figure presents an excerpt of the meta-model of web model. In the following, we use these figures in order to discuss the meta-model of web model.

As seen in Fig. 4.10 on page 51, the Website class has an attribute called `name` representing the name of the application, e.g. *vivoazzurro*. Furthermore, the Website class contains zero or more pages. In addition, it also references a page as the home page of the website.

The concept of page is represented by the `Page` class (see Fig. 4.10). Like Website class, the Page class has also an attribute `name` representing the name of the page. Furthermore, the Page class contains zero or more parameters, vari-

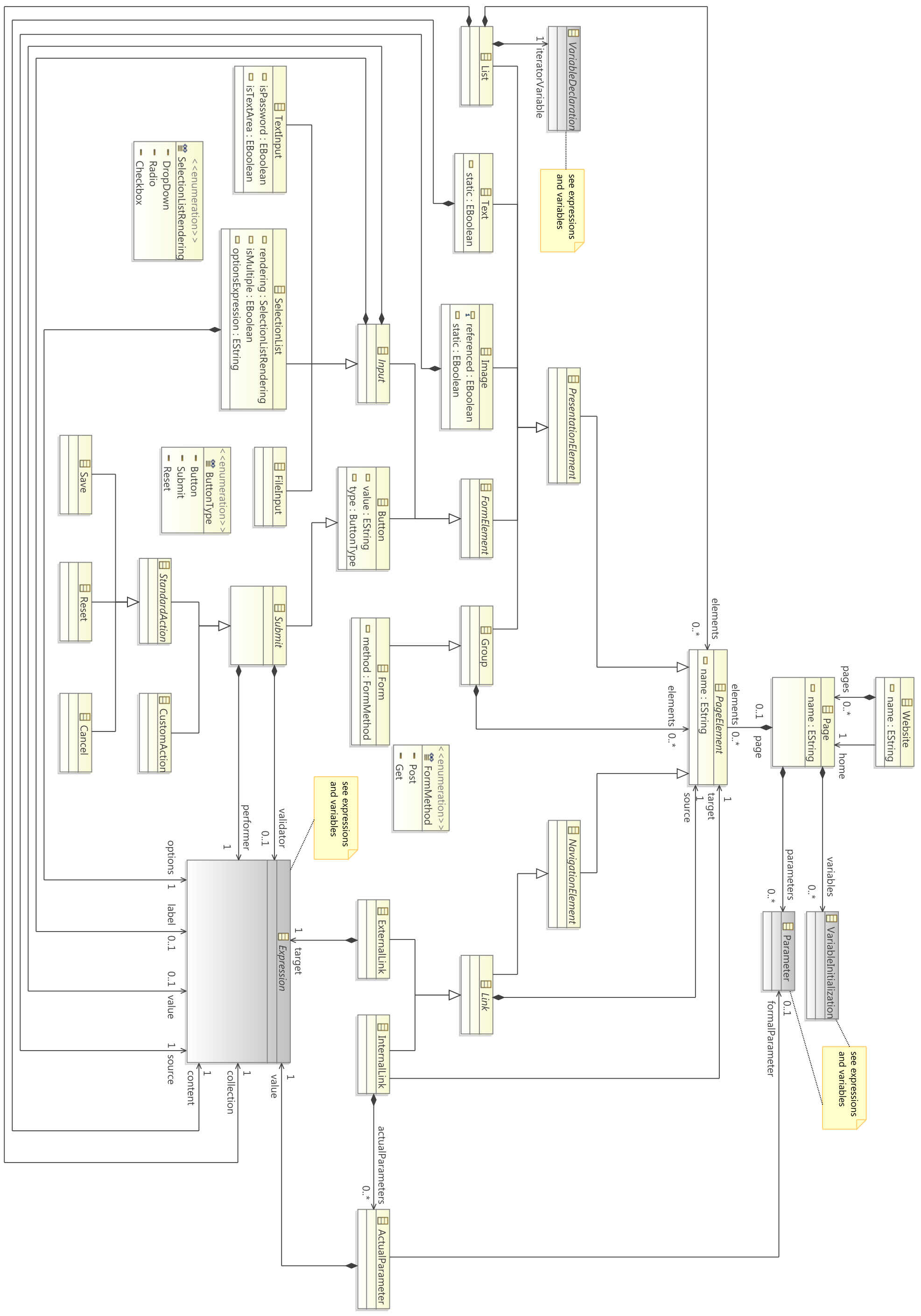


Figure 4.9: Overview of meta-model of web model where only details of main concepts are presented.

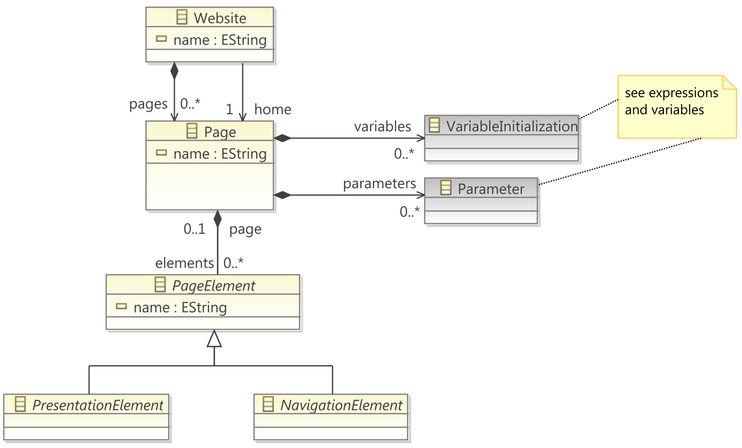


Figure 4.10: An excerpt of meta-model of web model where only details of `Website`, `Page`, `PageElement`, `PresentationElement`, and `NavigationElement` classes are presented.

ables and elements, where parameters are of type `Parameter`, variables of type `VariableInitialization`, and elements of type `PageElement`. The two former will be discussed later, while PageElement class is discussed in the following.

The PageElement class is specialised as `PresentationElement` and `NavigationElement` classes. Moreover, the PageElement class references the page containing it. In addition, the PageElement class has also an attribute `name` which is inherited by the various page elements through the PresentationElement and NavigationElement classes. In the following, we discuss the PresentationElement class using Fig. 4.11 on the following page and later discuss the NavigationElement class using Fig. 4.12 on page 53.

The PresentationElement class is representing the presentation elements of the page concept. This class is specialised as `List`, `Text`, `Image`, `FormElement` and `Group` classes, directly. The FormElement class is specialised further as `Input` and `Button` classes, where the Input class is, in turn, specialised as `TextInput`, `SelectionList`, and `FileInput` classes. The Button class is specialised as `Submit` class which represents the submit concept (see section 4.2.1.9 above). The Submit class is further specialised as `StandardAction` and `CustomAction` classes. These classes represent the standard submit concept (see section 4.2.1.11 on page 40) and custom submit concept (see section 4.2.1.10 on page 40), respectively. The StandardAction concept is further specialised as `Save`, `Reset`, and `Cancel` classes. As seen in Fig. 4.11 on the following page, Image, Text, List, Input, SelectionList and Submit classes reference the `Expression` class

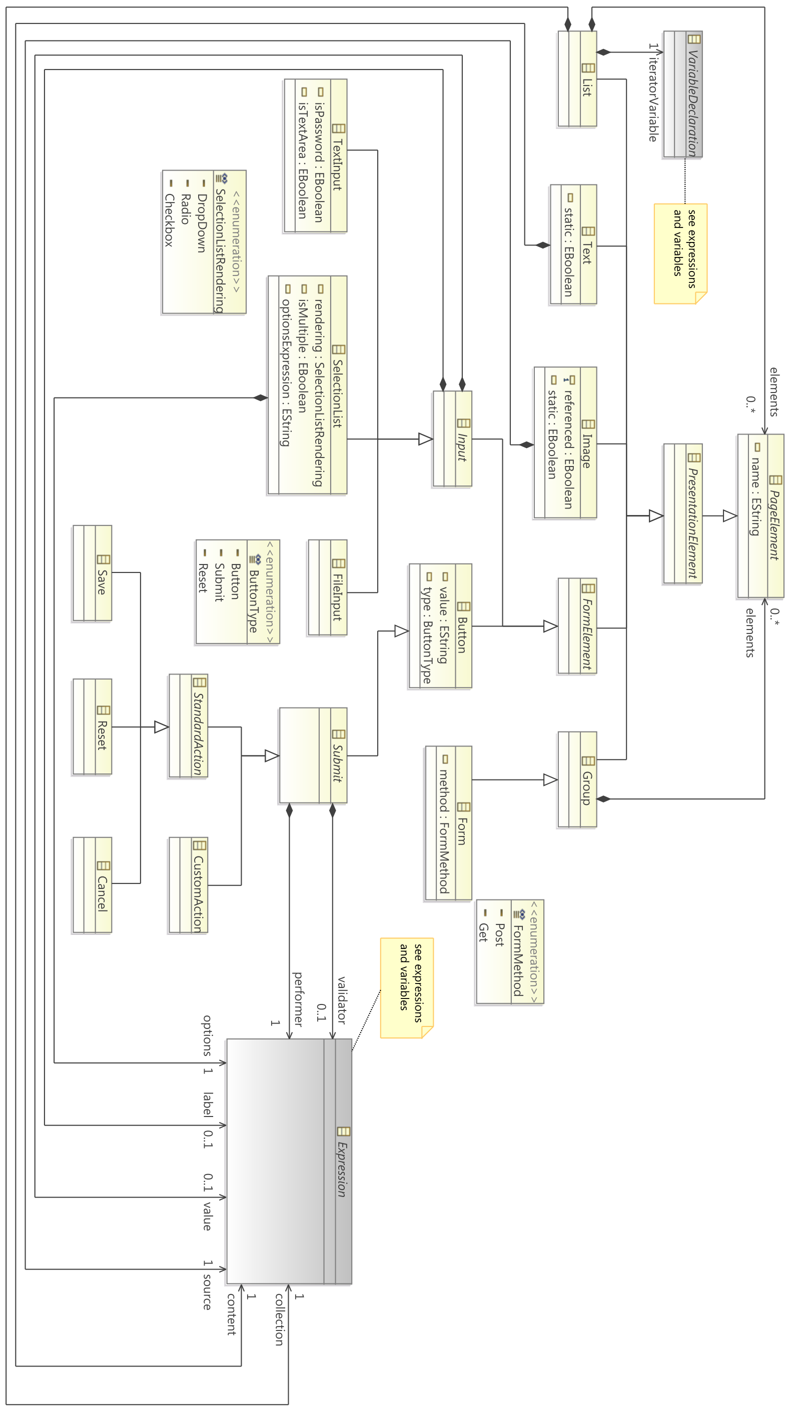


Figure 4.11: An excerpt of meta-model of web model where only details of presentation elements are presented.

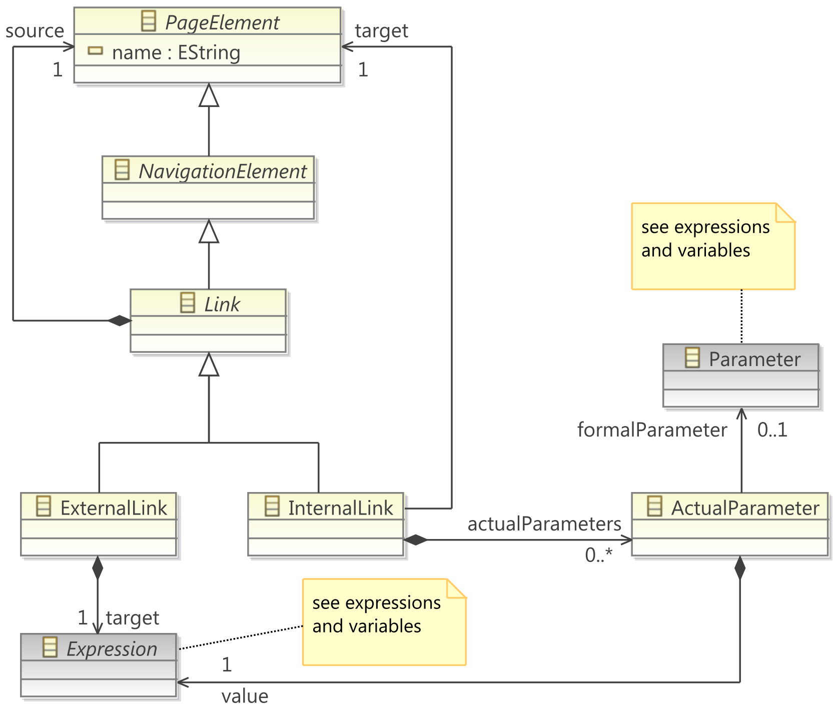


Figure 4.12: An excerpt of meta-model of web model where only details of navigation elements are presented.

which belongs to expressions and variables.

Consider Fig. 4.12 which we discuss in the following. The NavigationElement class is representing the navigation elements of the page concept. This class is specialised as `Link` class which, in turn, is specialised as `ExternalLink` and `InternalLink` classes. Notice that these classes have page element as *source* (by inheriting the source reference from Link class), but have different targets, i.e. the InternalLink class has a reference `target` of type PageElement, while the ExternalLink class has a reference `target` of type Expression. This was specified in sections 4.2.1.14 on page 41 and 4.2.1.13 on page 41, respectively.

We now discuss the expressions and variables in the meta-model of web model. Consider Fig. 4.13 on the following page and notice that in this figure the `E-Classifier`, `EStructuralFeature`, and `EOperation` classes, as indicated, belongs to the meta-model of Ecore model. For more information about these classes the reader is referred to [SBPM09].

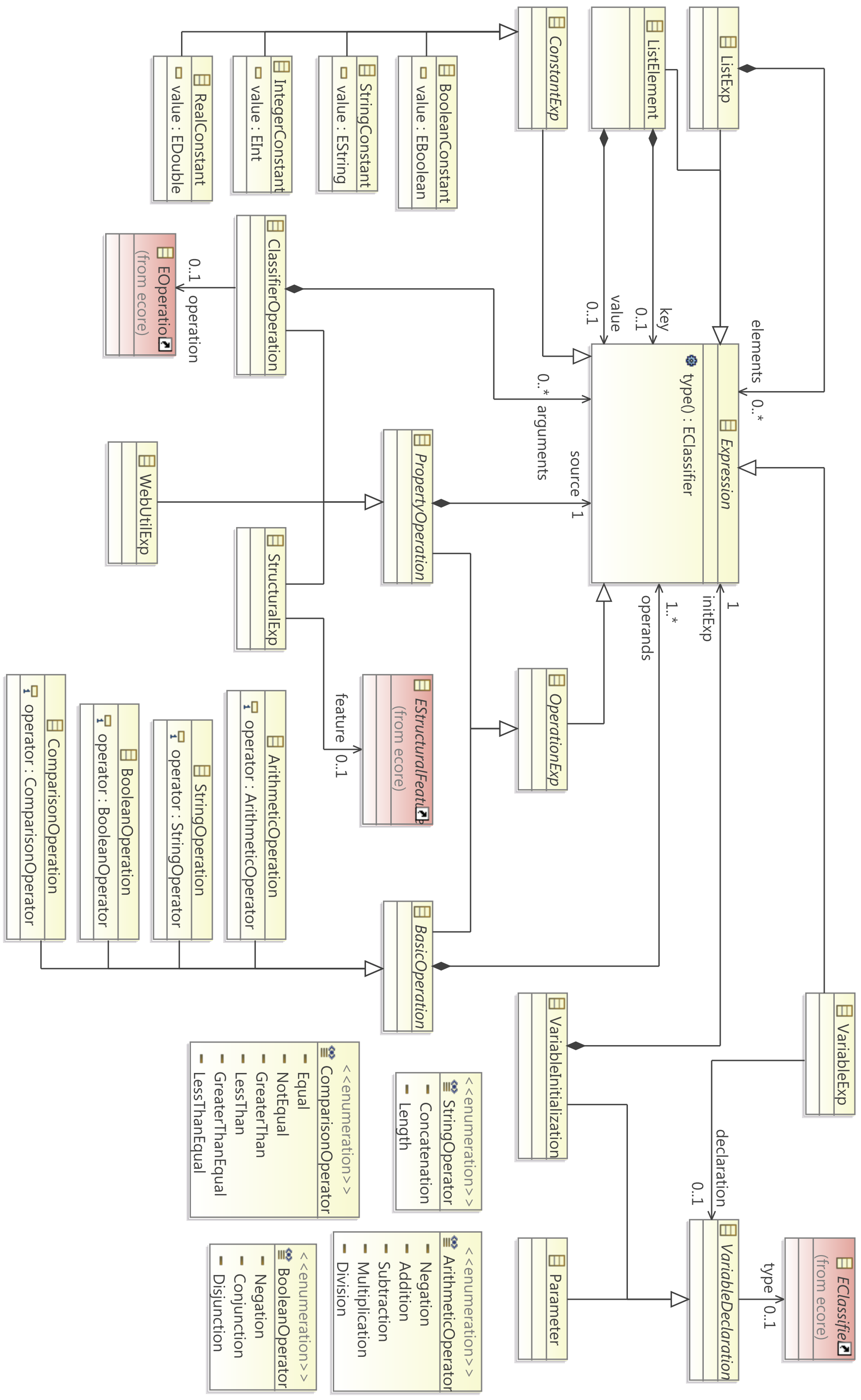


Figure 4.13: Overview of meta-model for web model where only details of expressions and variables are presented.

As seen in Fig. 4.13 on the preceding page, the Expression class is specialised as `ConstantExp` (representing constant expression), `OperationExp` (representing operation expression), `VariableExp` (representing variable expression), `List-Exp` (representing list expression) and `ListElement` (representing an element of a simple list or associative list) classes. The ConstantExp and OperationExp classes are specialised further; ConstantExp class as `BooleanConstant`, `String-Constant`, `IntegerConstant`, and `RealConstant` classes, while OperationExp class as `PropertyOperation` and `BasicOperation` classes. The PropertyOperation and BasicOperation classes are further specialised. The later one is specialised as `ArithmeticOperation`, `StringOperation`, `BooleanOperation`, and `Comparison-Operation`. The PropertyOperation class, on the other hand, is specialised as `ClassifierOperation`, `StructuralExp` (representing structural expression), `WebUtilExp` (representing web utilities expression) classes.

Consider the `VariableDeclaration` class in Fig. 4.13 on the facing page. This class is specialised as `Parameter` and `VariableInitialization` classes. The VariableInitialization class has a reference `initExp` representing the initial expression of the variable concept. Notice, that the type of parameters and variables is an EClassifier.

From above discussion and the meta-model in Fig. 4.9 and Fig. 4.13, it should be clear how the DSL for modeling web application is designed. In the following, we discuss this design in more details by presenting major design decisions and the related arguments.

4.4 Discussion of Web DSL Design

In this section, the design decisions regarding the abstract syntax of main concepts of web model are presented. In the following, we will mainly refer to Fig. 4.9 on page 50 which presents an overview of the abstract syntax of these concepts.

4.4.1 Text and Image as Static and Dynamic Concepts

As we mentioned above (see sections 4.2.1.3 on page 37 and 4.2.1.4 on page 37), both text and image concepts can either be static or dynamic. The idea behind this distinction is to determine whether the content of the text (and the source of the image) is provided by a constant expression, e.g. “`Name:`“, or by an expression that must be evaluated each time during runtime, e.g. `player.name`.

By knowing this fact, the evaluation of expression can be optimized. There are two possible approaches for modeling this in the abstract syntax of text and image. We begin by presenting the approach we have chosen. Then, an alternative approach is presented and compared to our approach.

Notice, that in figure 4.9 both `Text` and `Image` classes have an attribute `static` of type `EBoolean`. Another thing to notice about this attribute, but which is not obvious in the figure, is the fact that it is a derived attribute. The value of this attribute is derived based on the type of expression as the content of the text (and the source of the image). In the case of constant expression the value of this attribute will be *true* meaning that the text (and image) is static. In all other cases, the value of this attribute will be *false* meaning that the text (and image) is dynamic.

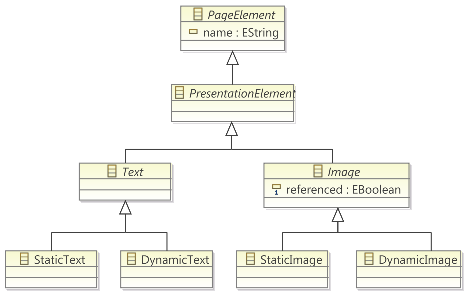


Figure 4.14: The alternative approach to the design issue regarding the static and dynamic Text and Image concepts.

An alternative approach to this design issue is using inheritance, i.e., by making `Text` and `Image` classes abstract and specialising them using two subclasses, namely one for the static case and one for the dynamic case. This is depicted in figure 4.14 where the Text class is subclassed by `StaticText` and `DynamicText` (and the Image class is subclassed by `StaticImage` and `DynamicImage`).

The alternative approach presented above, is not flexible and from the modeling perspective both `StaticText` and `DynamicText` are the same; similarly for `StaticImage` and `DynamicImage`. Furthermore, this approach will also result in four classes, as seen in figure 4.14, and thus four different graphical notations would be required compared to the two in our case, which is preferred.

4.4.1.1 Elements of List Concept

As mentioned above (see sections 4.2.1.16, 4.2.1.6, and 4.2.1.5), the concepts group, form and list can contain any page element, also themselves. This is also depicted by the *elements* composition of the `Group` class in figure 4.9 on page 50. This composition is also inherited by the `Form` class which represents the form concept. However, the `List` class, which represents the list concept, has its own composition named *elements* and of the same type. It would have been natural that the List class also was a subclass of the Group class and thus inherited this composition from it. This is depicted in figure 4.15. The reason behind this is that the list concept was developed before the group and the form concepts, and since we did not refactor the implementation, it remained as it is.

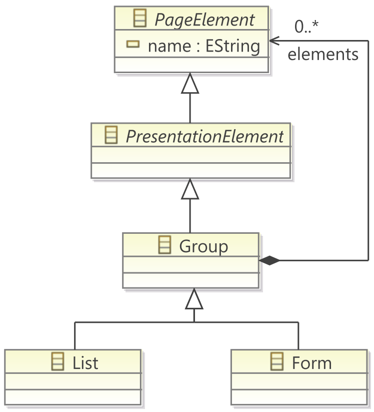


Figure 4.15: A better design of the abstract syntax of the List concept.

4.4.1.2 Scope of Parameters and Variables

In the following, we consider the `role` page from the running example (see Fig. 4.5 on page 33) and add to it, among other things, a list that displays the player's secondary roles. This is depicted in figure 4.16 on page 59. Notice, that the *iterator variable* `role` in the new list masks the parameter `role` of the page. In this example, it is not possible to access the parameter `role` within the new list due to the naming conflict, even though it has scope within the new list. This is also the case for the variables of the page. The issue here is how to address the naming conflict. There are two approaches for addressing this issue. In the following, we discuss these approaches in more details and present the approach chosen by us.

One approach for addressing this issue is by using *namespaces*. For instance, in order to make the parameter `role` accessible in the new list, we could express it like `page.role`, where `page` is the namespace here.

Another approach, which is the one applied by us, is to use *shadowing*. For instance, in order to make the parameter `role` accessible in the new list, they must have different names in order to avoid shadowing. This approach has the advantage of making the code generation very straight forward; at least for languages that support shadowing. For instance, the PHP[3] code generated for the role page would be something similar to:

```
<?php
    ...
    $role = ...;
    ...
?>
...
<?php
    foreach($player->secondaryRoles as $role)
    {
        echo $role->name;
    }
?>
...
```

Since also in PHP, the iterator variable will mask the page parameter due to variable shadowing in the PHP programming language, our approach of shadowing will work straight forward. The approach with namespaces, however, would require a special mechanism in order to work, and thus complicate the code generation. For the sake of simplicity, we have not applied the namespaces approach.

4.5 Summary

In this chapter, we designed and specified the DSL for modeling web applications on a higher level of abstraction. To this end, we introduced the so-called *web model*. Furthermore, this model was demonstrated through modeling of some

[3]The code generator, that targets the Joomla! platform, generates mainly PHP code, since the Joomla! platform is based on the PHP programming language.

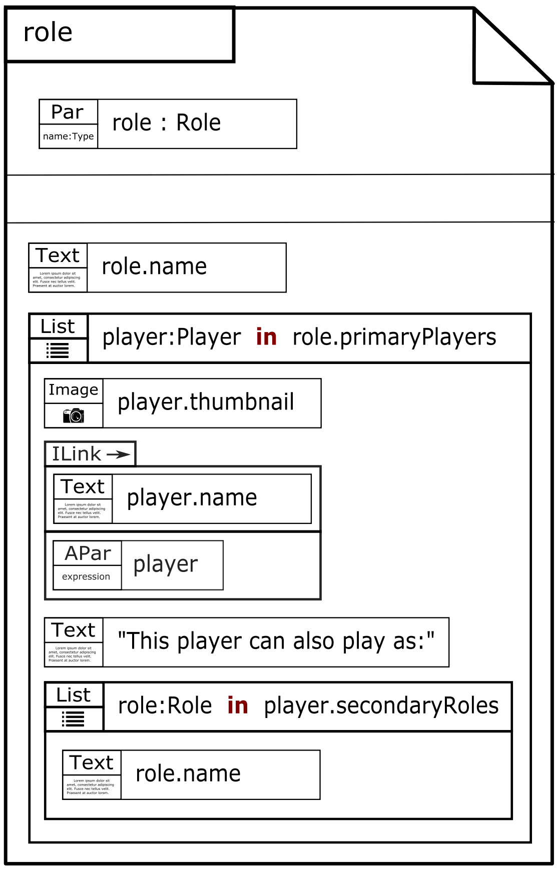


Figure 4.16: The role page from the running example extended with player's secondary roles.

pages from the running example. In addition, design decisions regarding the design of the DSL was also discussed.

Chapter 5

Analysis

In this chapter, we present requirements analysis of the end product, which we have named *Welipse* (Eclipse Tools for Web Engineering). Using this tool, it is possible to model web applications and generate code for them targeting the Joomla! platform. In the following, first, we describe and classify the users of Welipse. Next, we present the objectives of the tool. Finally, we give an overview of the development process with Welipse and following this, at the end of this chapter, the functional and non-functional requirements are specified.

5.1 Classification of Users

In the running example (see chapter 2 on page 5), an obvious process for developing web application can be traced. This process begins with the definition of the data model together with the web model in order to model the data and the pages of a web application and the flow between them. This information is then used in the Joomla! generator model to generate the code for the application which can be deployed on a running Joomla! CMS. There are different users involved in the various steps in the development process. This process is elaborated on later in this chapter. In the following, we classify the users involved in this process and discuss their roles.

The main users of the tool are employees, e.g. requirements analysts and developers, at Peytz & Co or any other web development company and the clients of these companies, i.e. the owner of the web application that will be developed by the company using Welipse. Furthermore, the users of the web application are also users of the tool, but in an indirect way; they are affected by the result produced by the tool. We will be referring to this class of users as end-users. Both the client and the end-users do not have direct influence on the requirements of the tool. However, they are affected by the result produced by the tool. Thus, they are in the equation and should be considered.

After the deployment of the web application, the end-users will be directly involved in the development process; they will be able to use the application. The customers (the owner of the application), will be involved in most parts of the development process such as defining the data (data model) and specifying various pages of the application and their content (web model). Generally, they will also be involved in testing the application. Testing the application, however, is not a part of the development process with Welipse; there are no mechanisms for devising and conducting tests using Welipse. This aspect of web application development has not been considered due to the scope of this thesis. However, testing the usability aspect as well as the flow between pages of the application is possible with Welipse.

The role of the development companies can be further divided into two different kind of roles. These are requirements analysts who will use the tool for requirements specifications and the developers who will benefit from the tool in the development process. The requirements of these two groups are not in conflict but due to time-limitation we can only focus on one of these. From our discussion with Peytz & Co, we agreed to address the requirements of the requirements analysts (in the case of Peytz & Co the project managers), since they believe that the benefits of using the product in requirement specification phase is far greater than using it in the development phase. Requirements analysts will in cooperation with clients, use Welipse to specify the data model (also domain model) and the various pages in the application and the content and navigation between them. They will also be able to generate running application from these models in order to rapidly develop a prototype of the application.

5.2 Objectives

As we also mentioned in the introduction, the strength of our proposal and the underlying tool is the ability to prototype rapidly. Thus, the main objective of the end product is to capture the requirements of the application being developed in an enhanced way and in a much shorter time. Beside being efficient, the end product should also be reliable such that it can be deployed by analysts in real life projects, i.e. to develop real life applications.

5.3 Requirements

In the following we address the main requirements of the tool (both functional and non-functional). We begin with an overview of the development process with Welipse.

5.3.1 Overview of Development Process

As mentioned above, an obvious process for developing web application using Welipse can be traced in the running example (see chapter 2 on page 5). An overview of this process is depicted in figure 5.1. As shown in this figure, the process begins with the definition of *data model*.

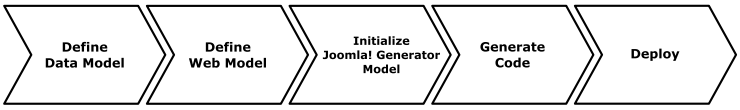


Figure 5.1: Overview of development process with Welipse.

The next step in the development process is the definition of the *web model* for specifying the various pages within the application, their content and the navigation between them. The data model and the web model form the PIMs in terms of MDA, i.e. they do not depend on any specific platform.

The web model (and indirectly the data model) is used to initialize the *Joomla! generator model* which contains the configuration needed in order to generate a web application targeting the Joomla! platform. The Joomla! generator model can be, among others, configured with *initial data* of the application and custom cascading style sheets (CSS) for describing the look and formatting of the application.

Once the Joomla! generator model is configured, a complete Joomla! 2.5 component can be generated automatically. This is the fourth step in the development process with Welipse (see figure 5.1). The resulting *code*, can be manually extended or directly deployed to a Joomla! CMS. The *deployment* of the generated code, completes the development process with Welipse.

The rest of this section specifies the above mentioned steps and the artifacts involved in these steps in more details.

5.3.2 Specification of Non-Functional Requirements

In the following, each non-functional requirement of the tool is discussed individually. To begin with, we outline the *basic criteria* for selecting various technologies for the implementation of Welipse.

5.3.2.1 Technology of the Tool

Since one of the main goals of this thesis is to provide an open source tool, the technology used to build the tool must also be open source. To this end, we have utilized the Eclipse platform together with EMF (Eclipse Modeling Framework). Another reason for choosing Eclipse and EMF is due to the fact that these technologies are familiar to us. The choice of subsequent technologies for developing various parts of the tool has been based on three *basic criteria*. These criteria are:

- The technology must be open source.
- The technology must be integrable with the Eclipse Platform and IDE.
- The technology must be integrable with EMF.

5.3.2.2 Usability

Since one of the objectives of Welipse is to be deployed in real life projects, it must be usable by requirements engineer and/or project managers as they are the primary users of the tool. To this end, we will provide a graphical editor for the DSL that will be developed. The usability of this editor is, however, limited to the underlying technology it is developed with, since we will not develop this editor from scratch but will generate it.

5.3.2.3 Performance

The main area concerning performance in Welipse is code generation. The approximal time for generating code must be under a minute. This is, however, only the case for models of relative size; below 10 pages in the web model and the same size of data model, i.e. below 10 classes.

5.3.2.4 Maintainability

There are three different kinds of maintainability, namely the maintainability of the tool, the maintainability of the models created and used by the tool and the maintainability of the automatically generated code. We specify these in the following.

The maintainability of the tool is determined by its overall architecture. One possibility of such overall architecture is to separate main components of the tool from each other. We will use this architecture by developing each component as an Eclipse *plugin*.

The maintainability of the models is also determined by their overall architecture. One possibility of such overall architecture is decoupling core concepts (CIMs/PIMs) from the platform-specific extensions (PSMs) totally. The advantage of this architecture is that, CIMs/PIMs and PSMs can be maintained individually. However, these models can get out of sync with each other very easily. To remedy this disadvantage, we will provide an automatic way of reconciling the PSMs with CIMs/PIMs.

The maintainability of the generated code is determined by whether it is possible to insert manually written code and regenerate the code without losing the manually written code. We will provide a way to extend the generated code by manually written code without these being lost in the process of regeneration.

5.3.2.5 Extensibility

The extensibility of the tool is determined by the overall architecture of the tool. By designing the tool based on the above mentioned architecture, it is possible to extend Welipse by adding more platforms such as Joomla!; the default platform in Welipse.

5.3.3 Specification of Functional Requirements

From the overview of the development process with Welipse shown in figure 5.1 on page 63, some main artifacts can be identified. These are: data model, web model, Joomla! generator model, and the generated code. In addition to these artifacts, Welipse supports functionality such as import of *initial data* and customer CSS files of the application. Furthermore, Welipse provides a *code generator* targeting the Joomla! platform. In the following, we specify these in more detail.

5.3.3.1 Data Model

The data models are defined using EMF-models. We will not provide more details about these models, but instead refer the reader to [SBPM09] for more details about these models and the EMF-technology in general. However, we specify the extent of their use as data model. An example of an EMF-model is given in figure 2.6 on page 11. As seen in this figure, the classes, attributes and references are supported. In addition, operations are also supported. No other concept, beside these concepts, is supported, e.g. inheritance.

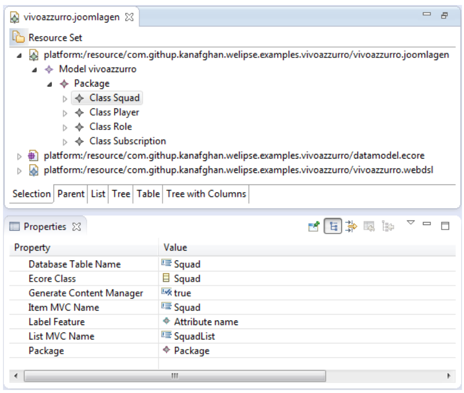
5.3.3.2 Web Model

We have already discussed this model in details in chapter 4 on page 25. Web model, which is a platform independent model in terms of MDA, is a core part of Welipse and models the concepts within the domain of web applications. For specification of this model, we refer the reader to section 4.2 on page 33.

5.3.3.3 Joomla! Generator Model

Welipse will support the Joomla! platform as the default platform one can generate the application code for. To this end, Welipse will provide a code generator that generates codes for the Joomla! 2.5 platform. This generator will produce codes for a complete Joomla! 2.5 component. More details on Joomla! and Joomla! 2.5 component are provided in section 8.5.1 on page 117. In the following, we specify the various configurations, i.e. the customization of generated code, supported by Welipse.

Joomla! Generator Model in the Running Example Consider the simple application we have developed so far in chapters 2 on page 5 and 4 on page 25. In order to generate code for this application, we need to initialize a *Joomla! generator model* from the web (see Fig. 4.7 on page 35) and data (see Fig. 2.6 on page 11) models. This model is a so called *generator model* which is used to generate the actual code for the application.



```
platform:/resource/com.github.kanafghan.welipse.examples.vivoazzurro/vivoazzurro.joomlagen
    Model vivoazzurro
        Package
            Class Squad
            Class Player
            Class Role
            Class Subscription
platform:/resource/com.github.kanafghan.welipse.examples.vivoazzurro/datamodel.ecore
platform:/resource/com.github.kanafghan.welipse.examples.vivoazzurro/vivoazzurro.webdsl
```

Property	Value
Database Table Name	Squad
Ecore Class	Squad
Generate Content Manager	true
Item MVC Name	Squad
Label Feature	Attribute name
List MVC Name	SquadList
Package	Package

Figure 5.2: The screenshot of the Joomla! Generator Model for the application in the running example.

Figure 5.2 presents the Joomla! generator model for our simple application. Here, we can see that the data model of our application is presented using a `Package` that contains the Squad, Player, Role and Subscription classes presented as `Class`. In the properties view, we can see the various properties of a `Class`, for instance the property *Database Table Name* represents the name of the database table that will be generated for this class. By changing these properties one customises the generated code. The default values of these properties, however, are reasonable for immediate code generation in most cases. In the following, we specify all the different properties of the Joomla! generator model.

Specification of Joomla! Generator Model An initial version of a Joomla! generator model, unlike data model and web model which are created manually, is created automatically by providing the corresponding web and data models. Then it can be customized. For instance, the Joomla! generator model presented in Fig. 5.2 is created automatically from the corresponding data model (see Fig. 2.6 on page 11) and web model (see Fig. 4.7 on page 35). In other words, the Joomla! generator model will wrap the data and web models and, by providing various properties, facilitate the means for customizing the generated application. Like data and web model, the Joomla! generator model has also a meta-model. An overview of this meta-model is presented in Fig. 5.3 on page 71.

On the top level (application level), the properties of Joomla! generator model are divided into three different categories, namely *Database*, *General* and *Manifest*. The manifest related properties are the **extension name** in Joomla! (the application name which is initialized from the web model), **creation date** of the application, information about the application author such as **name**, **e-mail** and **website**, **copyright** information, **license**, **version**, and **description** of the application. Furthermore, the *Joomla! version* and *extension type* properties are also manifest related. The **Joomla! Version** property specifies the version of the Joomla! that the generated code targets. Since Welipse only supports Joomla! 2.5 component, the only possible value of this property is *J25*. The **Extension Type** property specifies the type of Joomla! extension that will be generated. Since Welipse only supports Joomla! 2.5 component, the only possible value of this property is *component*. The manifest related properties are used in the generation of the Joomla! component manifest.

General Properties The Joomla! generator model also provides general properties such as *component name*, *custom CSS files*, *initial data*, *use bootstrap*, and *minified bootstrap*. These are specified in the following.

The **Component Name** property is derived from the *extension name* property and is not changeable. It is used in the code generation process and represents the generated Joomla! component name.

The **Custom CSS Files** property, as the name suggests, provides the mean for adding custom CSS (Cascading Style Sheets) files to the generated application. This is done by entering the absolute path of the CSS file. Multiple CSS files can be added by separating the paths with a semicolon (;).

By setting the **Initial Data** property, it is possible to initialize the application with data. In other words, this data, which is provided as a zip file, is used to initialize the generated database tables. This property is specified in more details in section 5.3.3.4 on page 72.

The **Use Bootstrap** property specifies whether to use the Bootstrap[1] framework in the generated application. The default value of this property is *true* which means the Bootstrap framework will be added to the application.

[1]Bootstrap is a front-end framework for developing responsive, mobile first web applications. For more information please refer to `http://www.getbootstrap.com`

The **Minified Bootstrap** property specifies whether to use the minified version of Bootstrap framework. The default value of this property is *false* which means that the minified version of Bootstrap framework will not be added to the application. This property is only applicable if the *use bootstrap* property is enabled.

Database Related Properties On the application level, the Joomla! generator model only provides one database related property, namely **Database Table Prefix**. This property specifies the prefix used in the names of the generated database tables; these tables are generated from the data model. The default value is `#__` followed by application name and suffixed with an addition underscore (_), e.g. `#__vivoazzurro_` where *vivoazzurro* is the name of the application. This is the standard prefix used in Joomla! development. This way identical table names from different extensions will not conflict with each other.

Customizing Database Generation As shown in Fig. 5.2 on page 67, the Joomla! generator model provides properties for customizing the generated database tables, e.g. the name of the generated database table corresponding to a class in the data model. Such properties are provided both on class, attribute and reference level. These properties are specified in the following.

The **Database Table Name** property, as the name suggests, specifies the name of the database table corresponding to a class in the data model. In the example in figure 5.2 the class Player is given a *Player* database table name. The Database Table Name property is initialized from the name of the class; it is a provided for each class in the data model, i.e. it is a class level property.

The **Database Column Type** property, as the name suggests, this property specifies the type of the database column that corresponds to an attribute of a class in the data model. Possible values are `INT`, `FLOAT`, `DOUBLE`, `VARCHAR`, `TEXT`, `DATE` and `TIMESTAMP`. This property is initialized by guessing the value through the type of the attribute, e.g. an attribute of type `String` will result in `VARCHAR` as value. This property is provided for each attribute of each class in the data model; it is an attribute level property.

The **Nullable** property specifies whether the database column representing an attribute or a reference of a class can have `NULL` value. The default value of this property is `true` which means that the data column can have NULL value.

Customizing Joomla! Component The Joomla! generator model also provides various properties for customizing the generated Joomla! component. These properties are provided on several levels, i.e. on class level, attribute of a class and reference of a class level. These properties are specifies in more details in section 8.5.1.2 on page 120.

Other Customization Sometimes, one would like to specify the database table prefix property for each class in the data model. This is not possible in what was specified above. Furthermore, one would also like to map a class in the data model to an existing table. For instance, Joomla! provides a nice user (`users` table) and category (`categories` table) framework which can be used by mapping related classes from the data model to these tables. Such customization is considered future work.

Meta-model for Joomla! Generator Model In order to develop the Joomla! generator model with the above specification, we use the meta-model presented in figure 5.3 on the facing page. In this figure, the Joomla! generator model is represented by `JoomlaGenModel` class. The attributes of this class form the top level properties specified above. Furthermore, the JoomlaGenModel class has a reference `datamodel` to the `GenPackage` class. From this reference, we can see that each Joomla! generator model can only have one data model and that the data model must be contained in a single Ecore package. The JoomlaGenModel has also a reference `webmodel` to the `Website` class from web model. As one can see from this reference, a Joomla! generator model must have one (and only one) web model. Moreover, the JoomlaGenClass has also a reference `databaseTables` to the `DatabaseTable` class. This reference is a derived reference which derives the list of database tables of the application from the data model. The DatabaseTable class, as it is shown in the figure, has three references, namely `genClass`, `foreignKeys`, and `columns`; these references are to the GenClass, GenReference and GenAttribute classes, respectively. These three references are used to construct the database table corresponding to a class from data model. The attributes of GenClass, GenReference and GenAttribute classes represent the properties related to the customization of the generated database; these were specified above.

More information about the meta-model of Joomla! generator model is provided in section 7.1 on page 87.

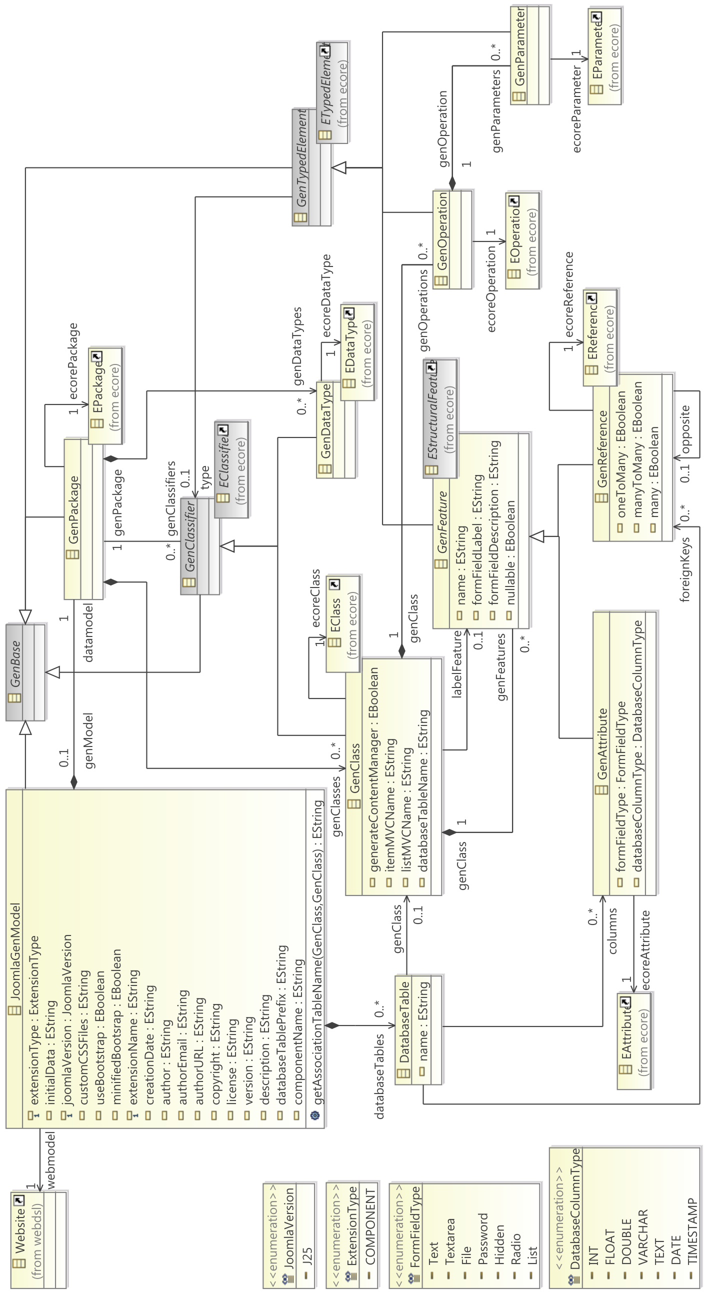


Figure 5.3: Overview of the meta-model of Joomla! Generator Model.

5.3.3.4 Data Import

It is possible to populate the database of an application by importing the data from CSV (Comma Separated Values) files. Each CSV file with the same name as a class from the data model will be used to populate the corresponding table in the database. Each column in the CSV file must represent an attribute of the corresponding class and thus a field in the corresponding database table. Each row in the CSV file is then representing a database record of the corresponding table.

For instance, consider the Player class in the data model in the running example (see Fig. 2.6 on page 11). This class has nine attributes and the corresponding database table will also contain these nine attributes as columns. The CSV file containing the data for this table must consist of these nine columns. This is shown in the following excerpt of the CSV file containing the players data.

```
name;photo;...;weight;gender
Marco Parolo;parolo__xpx.jpg;...;82 kg;1
Manuel Pasqual;Pasqualprofilo_.jpg;...74 kg;1
.
.
.
```

If the application has images, e.g. player's photo, these should be put in a folder named `img`. This folder and the CSV files must be archived as a `.zip` file and the absolute path of this file must be provided as the value of the *Initial Data* property of the Joomla! generator model. This property is specified above (see section 5.3.3.3 on page 66).

5.3.3.5 Graphical Editors

We will provide a simple graphical editor for working with web models. This editor will support all the graphical concrete syntax of web DSL concepts specified in chapter 4 on page 25.

5.4 Summary

In this chapter, we presented the implementation of Welipse. First, the main components of the tool were discussed. Next, the organisation of the implementation as Eclipse plug-in projects was presented. Finally, the implementation of each non-trivial part of the tool was discussed in more details; expressions language, code generation, etc.

CHAPTER 6

Use of Welipse

In the following, we present the development process with Welipse. This process also outlines the main use cases of Welipse. We will describe this process using the running example presented in chapter 2 on page 5 and Fig. 6.1 on the following page. As depicted in Fig. 5.1 on page 63 in the analysis chapter, the development process with Welipse is broken down into five steps: definition of the *data model*, definition of the *web model*, initialization and customization of *Joomla! generator model*, generation of *code*, and *deployment*. In the following, we elaborate on these steps.

6.1 Data Model

The first step in the development process with Welipse is modeling the application data. This is done using the so called *data model*. In Welipse, EMF-models are used for this purpose. More precisely, the Ecore model is used in Welipse as data model. An example of such a model is given in Fig. 2.6 on page 11. For more information about the Ecore model and EMF-technology in general, the reader is referred to [SBPM09]. It must be emphasized that not all the concepts of Ecore are supported. Only EClass, EAttribute, EReference, and EOperation concepts are supported.

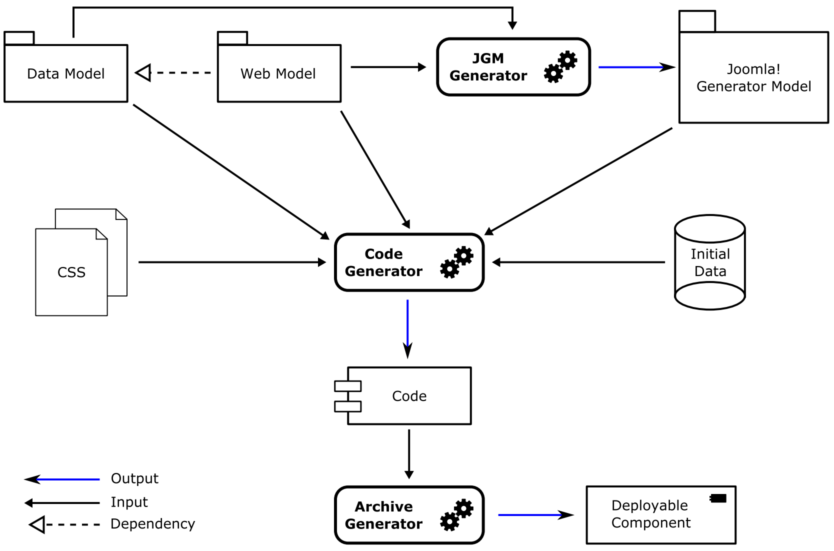


Figure 6.1: An overview of the different artifacts and their usage in the development process with Welipse.

For information about creation and definition of data models, please refer to the user guide in appendix A on page 143.

6.2 Web Model

The second step in the development process is modeling the actual web application, the pages comprising the application, and the navigation between these pages. Furthermore, the content of these pages are also defined in the web models. Examples of such pages are presented in Fig. 4.2 on page 28 (squad page), Fig. 4.3 on page 30 (player page), Fig. 4.5 on page 33 (role page), and Fig. 4.6 on page 34 (player form page). As seen, in these examples, the concepts of web application, such as *text*, and *image* or used together with elements from the data model in order to define the content of the various pages.

For information about creation and definition of web models, please refer to the user guide in appendix A on page 143.

6.3 Joomla! Generator Model

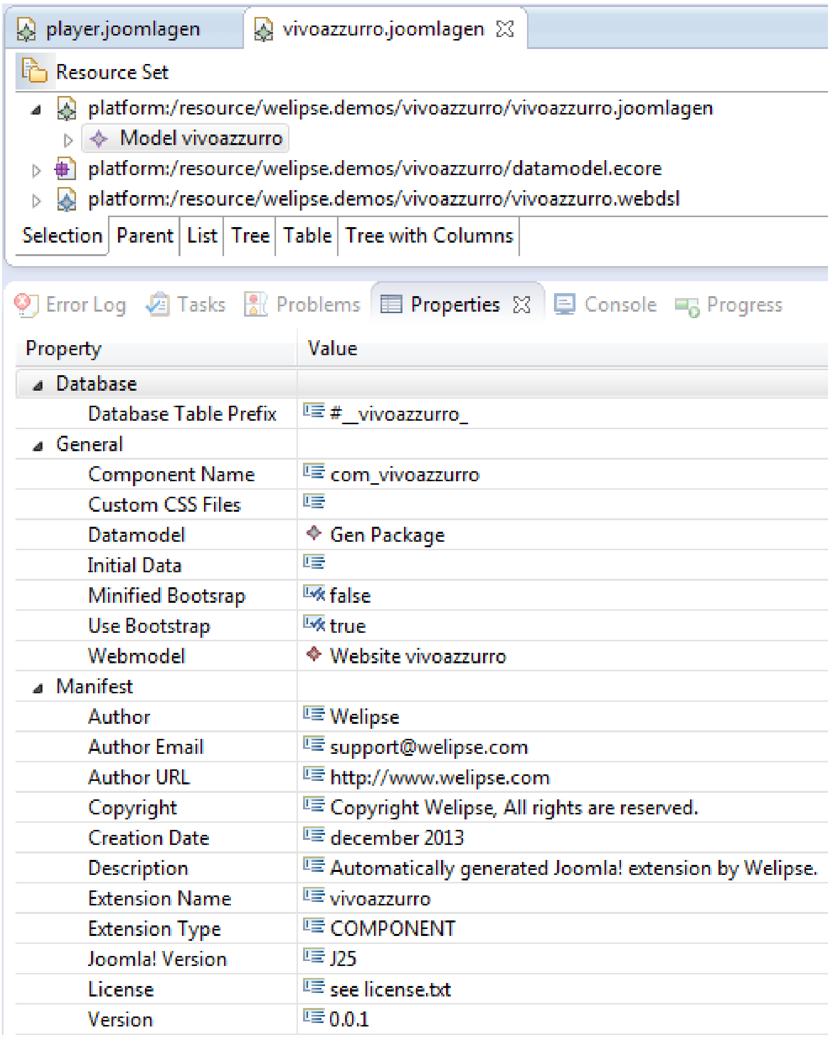
The third step in the development process with Welipse is the refinement of the data and web models through a platform specific model. The default generator model within Welipse is the Joomla! generator model which targets the Joomla! platform. This model can be automatically initialized (or generated) from the data and web models and can be manually changed to configure the generated code. This is also depicted in Fig. 6.1 where the JGM Generator (Joomla! Generator Model Generator) takes as input the data and web models and produces the corresponding Joomla! generator model.

In order to generate code for the simple application with the data and web models shown in Fig. 2.6 on page 11 and Fig. 4.7 on page 35, respectively, we need to initialize a Joomla! generator model from these models. A screenshot of this model is presented in Fig. 6.2 on the next page showing the properties on the application level. The Joomla! generator model is initialized automatically from the web and data models. Thus, many of the properties of Joomla! generator model are initialized with values from these models. By changing these properties, we can customize the generated code. The default values of these properties, however, are reasonable for immediate code generation in most cases.

In chapter 5 on page 61 (see section 5.3.3.3 on page 66), a screenshot of the Joomla! generator model showing the class level properties is shown (see Fig. 5.2 on page 67). In Fig. 6.3 on page 79, on the other hand, the attribute level properties (see Fig. 6.3a on page 79) and the reference level properties (see Fig. 6.3b on page 79) are shown. Information about specific configuration options of these properties and the Joomla! generator model in general is presented in section 5.3.3.3 on page 66.

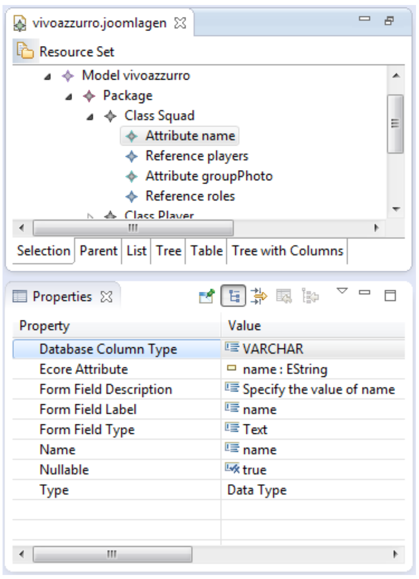
Using the Joomla! generator model, it is, among others, possible to import *initial data* and custom CSS files.

For more information about initialization, reconciliation and configuration of Joomla! generator model, please refer to the user guide in appendix A on page 143.


player.joomlagen
vivoazzurro.joomlagen
Resource Set
platform:/resource/welipse.demos/vivoazzurro/vivoazzurro.joomlagen
Model vivoazzurro
platform:/resource/welipse.demos/vivoazzurro/datamodel.ecore
platform:/resource/welipse.demos/vivoazzurro/vivoazzurro.webdsl
Selection
Parent
List
Tree
Table
Tree with Columns
Error Log
Tasks
Problems
Properties
Console
Progress

Property	Value
Database	
Database Table Prefix	#__vivoazzurro_
General	
Component Name	com_vivoazzurro
Custom CSS Files	
Datamodel	Gen Package
Initial Data	
Minified Bootsrap	false
Use Bootstrap	true
Webmodel	Website vivoazzurro
Manifest	
Author	Welipse
Author Email	support@welipse.com
Author URL	http://www.welipse.com
Copyright	Copyright Welipse, All rights are reserved.
Creation Date	december 2013
Description	Automatically generated Joomla! extension by Welipse.
Extension Name	vivoazzurro
Extension Type	COMPONENT
Joomla! Version	J25
License	see license.txt
Version	0.0.1

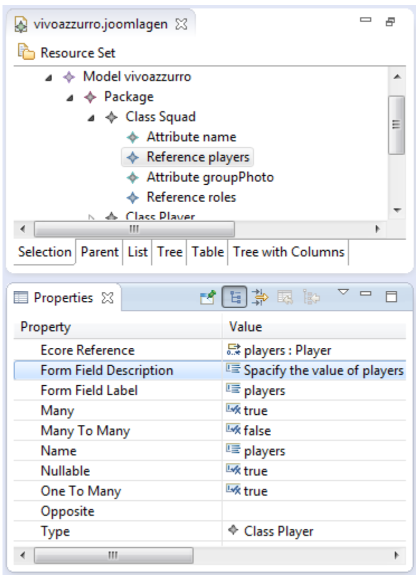
Figure 6.2: The screenshot of the Joomla! generator model for the application in the running example. This figure presents the properties on the application level.



```
◢ Model vivoazzurro
  ◢ Package
    ◢ Class Squad
        Attribute name
        Reference players
        Attribute groupPhoto
        Reference roles
      Class Player
```

Property	Value
Database Column Type	VARCHAR
Ecore Attribute	name : EString
Form Field Description	Specify the value of name
Form Field Label	name
Form Field Type	Text
Name	name
Nullable	true
Type	Data Type

(a) Properties of `name` attribute of Squad class.

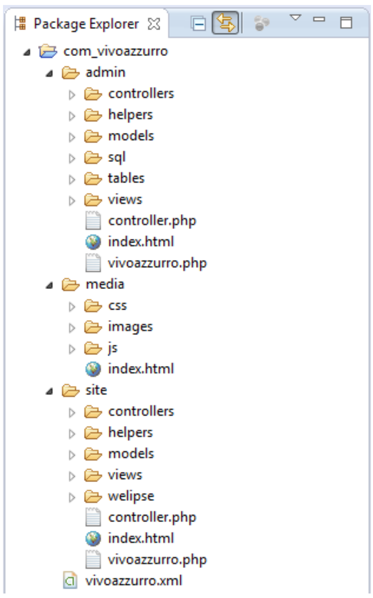


```
◢ Model vivoazzurro
  ◢ Package
    ◢ Class Squad
        Attribute name
        Reference players
        Attribute groupPhoto
        Reference roles
      Class Player
```

Property	Value
Ecore Reference	players : Player
Form Field Description	Spacify the value of players
Form Field Label	players
Many	true
Many To Many	false
Name	players
Nullable	true
One To Many	true
Opposite	
Type	Class Player

(b) Properties of `players` reference of Squad class.

Figure 6.3: Screenshots of the Joomla! generator model for application in the running example. This figure presents the properties of `name` attribute and `players` reference of the Squad class.



```
Package Explorer
com_vivoazzurro
  admin
    controllers
    helpers
    models
    sql
    tables
    views
    controller.php
    index.html
    vivoazzurro.php
  media
    css
    images
    js
    index.html
  site
    controllers
    helpers
    models
    views
    welipse
    controller.php
    index.html
    vivoazzurro.php
  vivoazzurro.xml
```

Figure 6.4: The a screenshot of the generated code for the Joomla! 2.5 component.

6.4 Code Generation

Once the Joomla! generator model is initialized and customized, it is possible to generate code for the web application targeting the chosen platform. As depicted in Fig. 6.1, the Code Generator takes as input all three models together with initial data of the application and custom CSS files and produces the corresponding code.

Code generation is the fourth step in the development process. In the case of Joomla! platform a complete Joomla! 2.5 component as Eclipse project is generated which is ready to be installed on a running Joomla! CMS (content management system). However, depending on the data model, the generated code might need manual implementation before deploying it, e.g. operations of classes in the data model must be implemented before they can be executed.

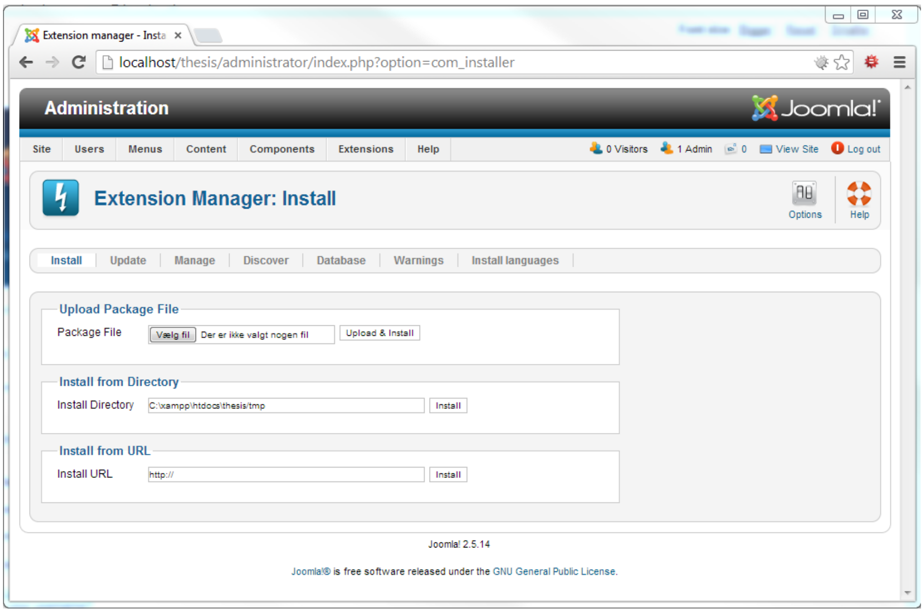
Figure 6.4 presents a screenshot of the generated code for the running example; a Joomla! component implementing the four pages shown above.

For more information about Joomla! and Joomla! component, please refer to section 8.5.1 on page 117. Information about code generation using the Joomla! generator model is presented in the user guide in appendix A on page 143. There, it is also possible to get the generated code for the running example in order to deploy it.

6.5 Deployment

The final step of the development process with Welipse is deployment of the generated application. As shown in Fig. 6.1, the Archive Generator takes the generated code and produces a deployable component which can be installed on a running Joomla! CMS.

The above generated code can be packaged into a zip archive and installed through the *Extension Manager* of a running Joomla! CMS (Content Management System). Figure 6.5 presents a screenshot of Joomla!'s extension manager. The "Upload Package File", as seen in this figure, option can be used in order to install the generated code as a zip archive.



Upload Package File

Package File [Vælg fil] Der er ikke valgt nogen fil [Upload & Install]

Install from Directory

Install Directory C:\xampp\htdocs\thesis\tmp [Install]

Install from URL

Install URL http:// [Install]

Joomla! 2.5.14

Joomla!® is free software released under the GNU General Public License.

Figure 6.5: A screenshot of the Extension Manager of the Joomla! platform.

6.5.1 Running the Application

Once the generated Joomla! 2.5 component is installed on a Joomla! CMS the component can be accessed through the following URL:

```
<joomla-base>/?option=com_vivoazzurro&view=squad&squad=1
```

where `<joomla-base>` is the base URL where the Joomla! CMS is installed, e.g. `http://localhost/thesis` or `http://www.example.com`. The above URL will present the squad page with the specific squad instance that is identified with number 1. More about this is presented in section 8.5.1 on page 117. A screenshot of the squad page, accessed by above URL, is shown in Fig. 6.6 on the next page.

Figure 6.7 on page 84, Fig. 6.8 on page 85, and Fig. 6.9 on page 86 present a screenshot of the player, role and player form pages, respectively. These screenshots are from the generated application.

6.6 Summary

In this chapter, we presented the development process with Welipse, thereby outlining the main use cases of the tool. The development process with Welipse can be summarized as: create the data model, create the web model, initialize and customize the generator model (e.g. Joomla! generator model), generate code and deploy the generated code.

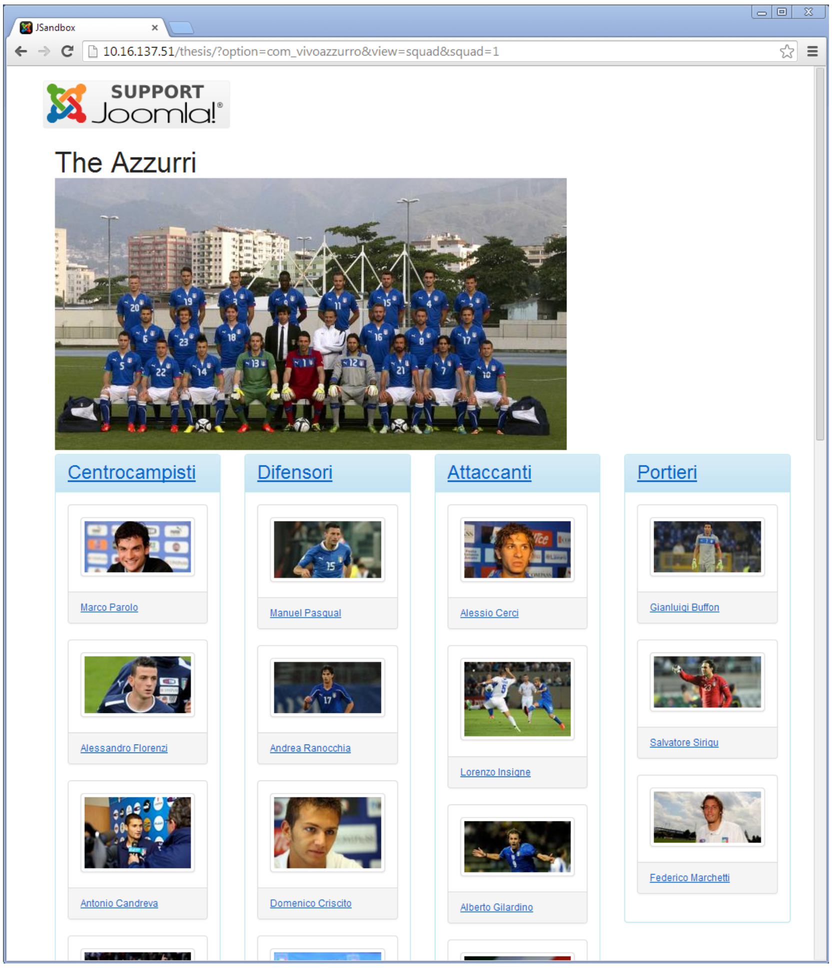
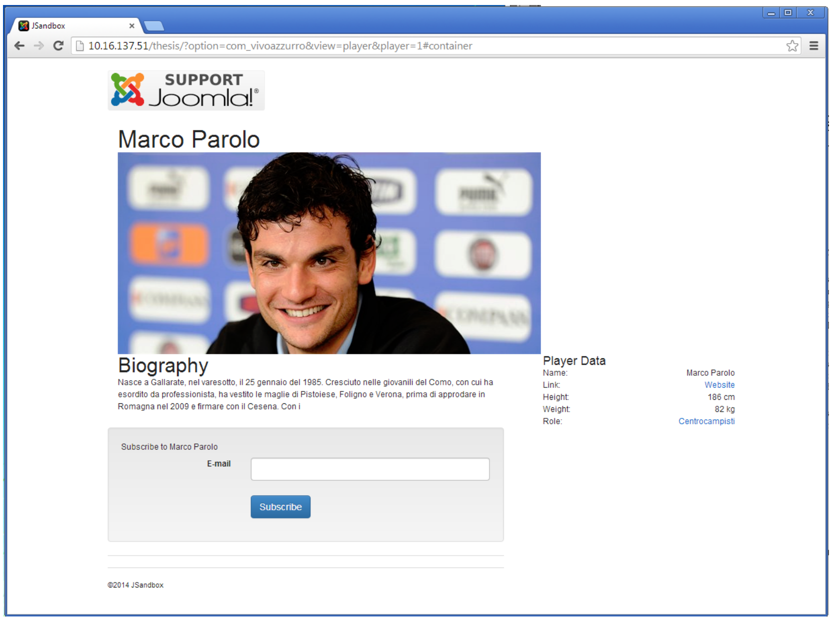


Figure 6.6: A screenshot of the Squad page from the generated application.


JSandbox
10.16.137.51/thesis/?option=com_vivoazzurro&view=player&player=1#container
SUPPORT Joomla!®

Marco Parolo

Biography

Nasce a Gallarate, nel varesotto, il 25 gennaio del 1985. Cresciuto nelle giovanili del Como, con cui ha esordito da professionista, ha vestito le maglie di Pistoiese, Foligno e Verona, prima di approdare in Romagna nel 2009 e firmare con il Cesena. Con i

Player Data

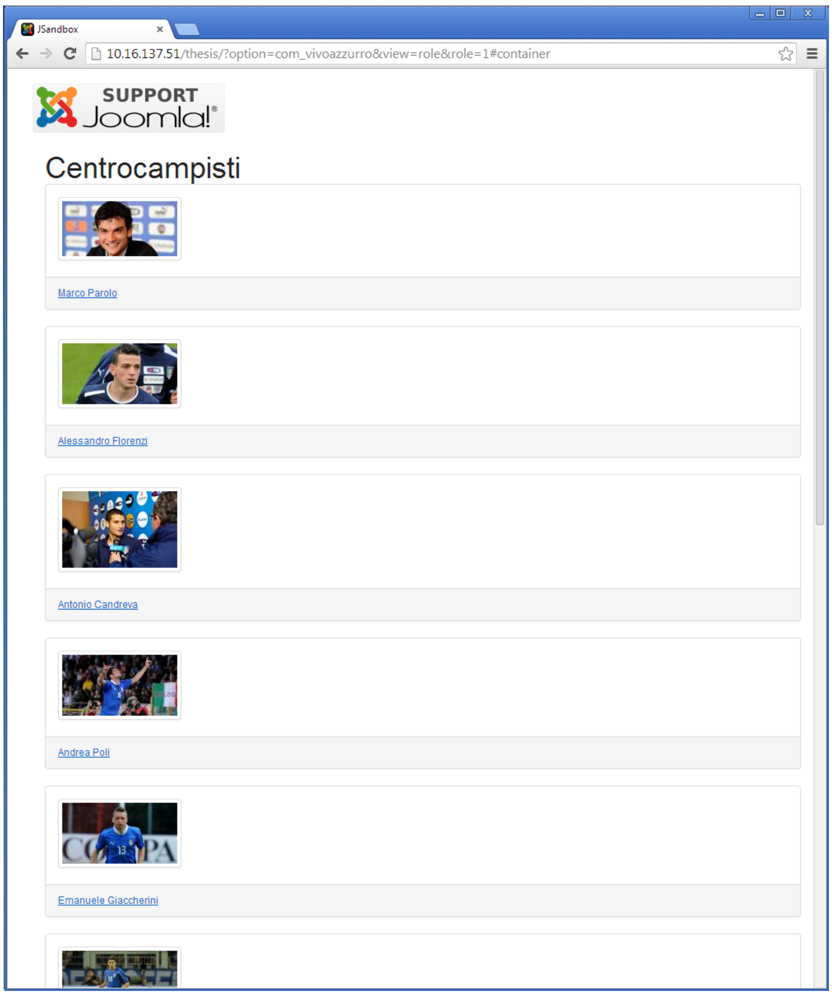
Name:	Marco Parolo
Link:	Website
Height:	186 cm
Weight:	82 kg
Role:	Centrocampisti

Subscribe to Marco Parolo

E-mail

©2014 JSandbox

Figure 6.7: A screenshot of the Player page from the generated application.


JSandbox
10.16.137.51/thesis/?option=com_vivoazzurro&view=role&role=1#container
SUPPORT Joomla!

Centrocampisti

Marco Parolo

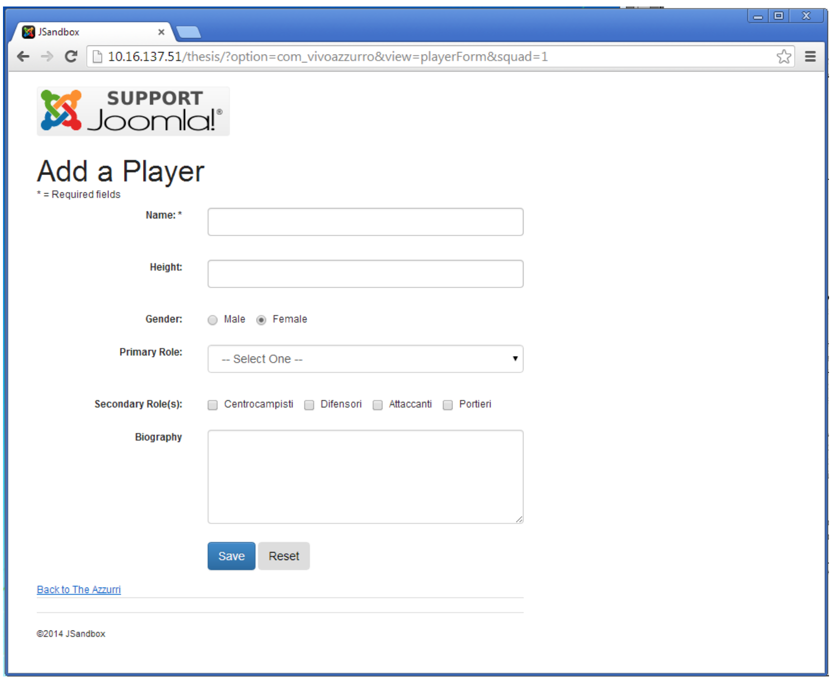
Alessandro Florenzi

Antonio Candreva

Andrea Poli

Emanuele Giaccherini

Figure 6.8: A screenshot of the Role page from the generated application.



10.16.137.51/thesis/?option=com_vivoazzurro&view=playerForm&squad=1

Add a Player

* = Required fields

Name: *	
Height:	
Gender:	Male [✓ Female]
Primary Role:	-- Select One --
Secondary Role(s):	Centrocampisti Difensori Attaccanti Portieri
Biography	

Save Reset

Back to The Azzurri

©2014 JSandbox

Figure 6.9: A screenshot of the Player Form page from the generated application.

Chapter 7

Design of Welipse

In the following, we present all major design decisions and the related arguments. We begin by elaborating on the design of Joomla! generator model. Next, mapping objects to tables is discussed.

7.1 Joomla! Generator Model

So far, we have seen multiple examples of the Joomla! generator model (see Fig. 5.2 on page 67, Fig. 6.2 on page 78, Fig. 6.3a on page 79, and Fig. 6.3b on page 79). The meta-model for this model is discussed in chapter 5 on page 61 (see Fig. 5.3 on page 71). In the following, we discuss the design of this meta-model.

The way Joomla! generator model is designed is inspired from the design of EMF generator model; the so called *GenModel* [SBPM09]. Thus, the way an instance of Joomla! generator model is constructed is similar to the construction of a GenModel. More on this topic is presented in the following chapter (see section 8.4 on page 113). Here, we focus on the meta-model of Joomla! generator model.

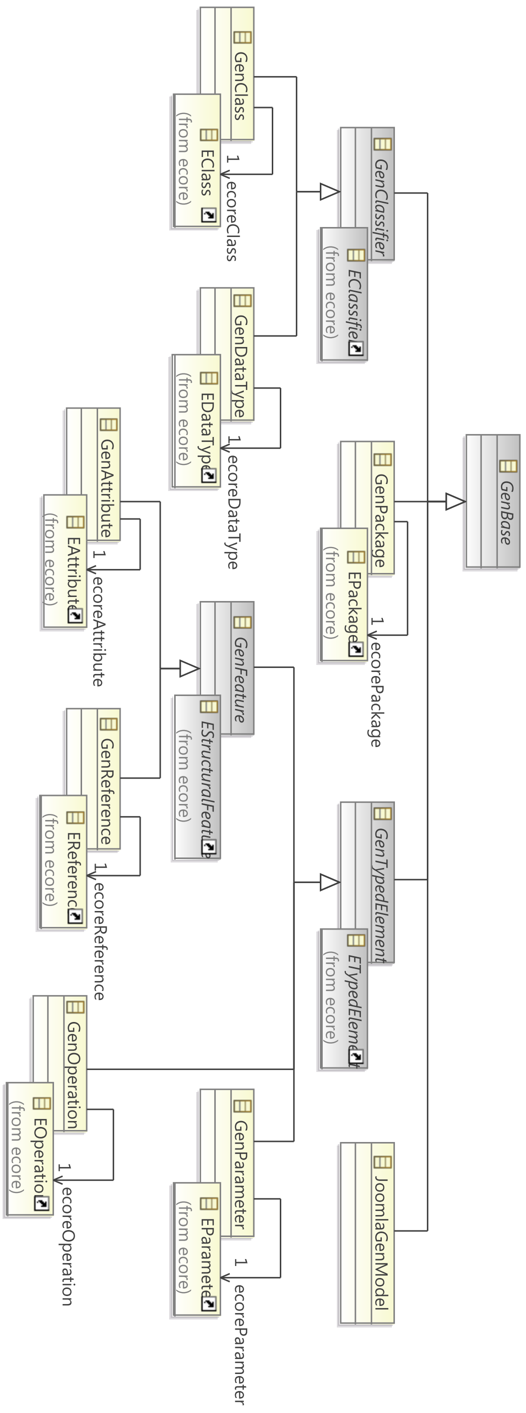


Figure 7.1: Overview of hierarchy in the meta-model of Joomla! generator model.

Figure 7.1 on the facing page presents an overview of the hierarchy in the meta-model of Joomla! generator model. In this figure, we can see both the Joomla! generator model classes and the corresponding Ecore model classes wrapped by them. The Joomla! generator model classes decorate the Ecore model classes in the same way the EMF generator model classes decorate Ecore model classes [SBPM09]. In order to compare EMF generator model with Joomla! generator model, we use Fig. 7.2 on the next page which presents an overview of hierarchy in the meta-model of EMF generator model. Despite some differences, the pattern is clear by looking at Fig. 7.1 and Fig. 7.2. Where the two models differ is, among other things, the `GenFeature` class. In the meta-model of the Joomla! generator model, this class is abstract and has two subclasses for decorating the EAttribute and EReference classes from the Ecore model. The GenAttribute and GenReference classes, which are not needed in EMF generator model, are , however, necessary for the construction and generation of database tables in the Joomla! generator model. These classes are mapped to columns and foreign keys of database tables, respectively. Construction and generation of database tables are elaborated on in section 8.4.3 on page 116.

Another difference between the meta-model of Joomla! generator model and the meta-model of EMF generator model is the absence of many classes in the meta-model of Joomla! generator model, e.g. the `GenEnum` class. These are not yet supported by the Joomla! code generator, but can be added in the future if needed.

The advantage of this approach, as mentioned in [SBPM09], is that the Ecore meta-model (which is also the meta-model for our data model) does not change and can remain independent of any information that is only relevant for code generation. Furthermore, this approach addresses the *incremental MDA* approach for applying the MDA pattern, which was discussed in section 3.1.1 on page 13.

The disadvantage with this approach, as also mentioned in [SBPM09], is that the two models (data model and Joomla! generator model) might get out of sync if the web and/or the data models are changed. To handle this, the generator model elements are able to reconcile themselves with changes to their corresponding Ecore elements [SBPM09]. This reconciliation is done automatically, and the user does not need to worry about it. The reconciliation process for the Joomla! generator model is elaborated on in section 8.4.2 on page 114.

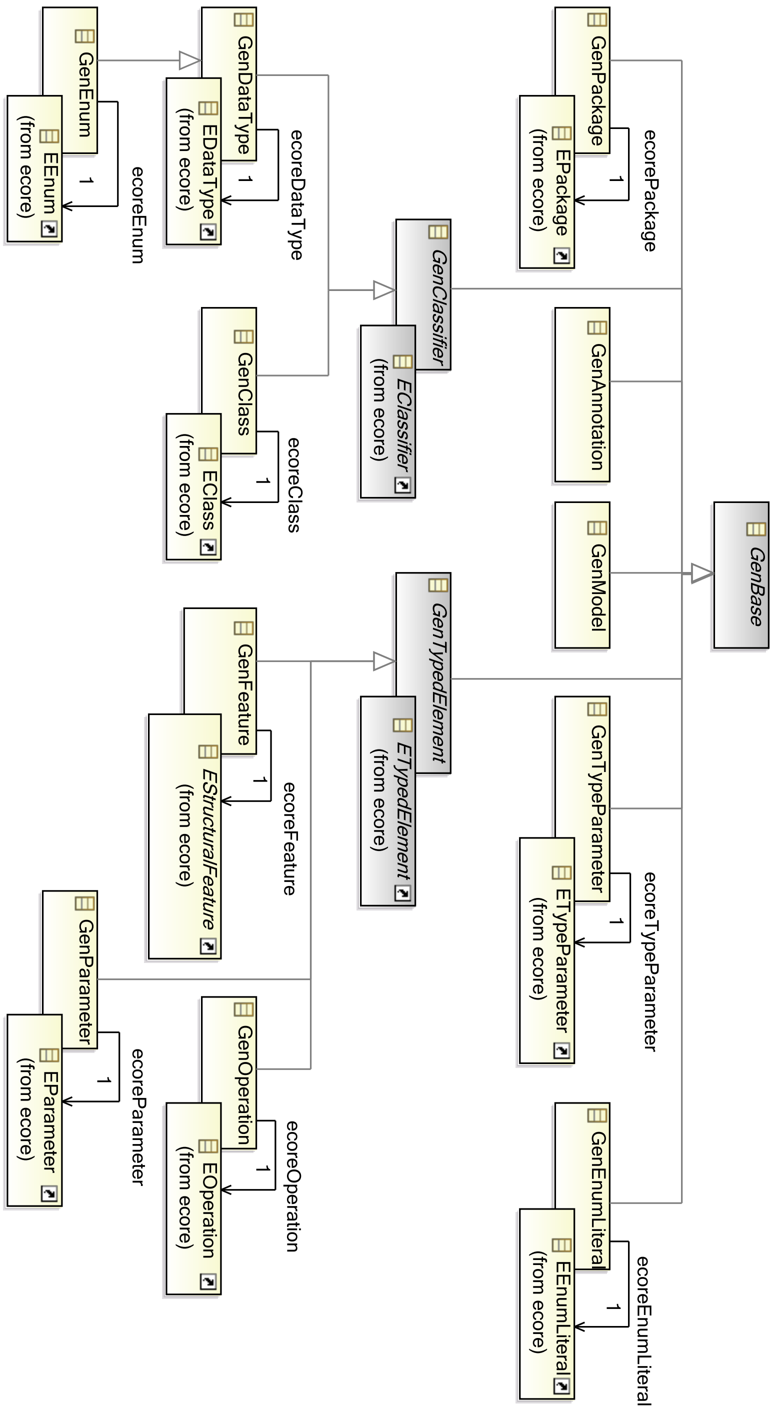


Figure 7.2: Overview of hierarchy in the meta-model for EMF GenModel.

7.1.1 Initial Data Import

As mentioned in section 5.3.3.3 on page 66, the Joomla! generator model provides a way to import the initial data of the modeled application; the *Initial Data* property of the Joomla! generator model. This is specified in section 5.3.3.4 on page 72. In the following, we discuss the design of the approach used to import the data.

The initial data of the application must be provided in a specific way in order to be interpreted and imported properly by the code generator. Although, the initial data must be provided during customization of the Joomla! generator model and before the code generation, we recommend to run the code generator before defining the initial data. This issue is discussed in more details in the user guide (see section A.7.1 on page 164). Here, we discuss the rationale behind the design of this feature.

From the discussion with Peytz & Co, a common method and format for importing the initial data of the application could not be find. According to Peytz & Co, the data provided from clients comes in various forms and formats which must be manipulated in one way or another before it can be used in development. The approach we have chosen is a simple approach (with regards to implementation) but nevertheless very strict. Due to time-limitation of this thesis, we have implemented this approach. The ideal solution, however, will be a sort of graphical user interface that provides means for importing data in different formats, e.g. SQL, CSV, EXCEL, etc. and facilitates features for mapping the data to classes in the data model.

The import of initial data of the application is considered a major feature of Welipse, since the data (real life data) during prototyping is crucial for the development of web application, cf. Peytz & Co. This feature is an obvious future work item.

7.2 Mapping Objects to Tables

In the meta-model of Joomla! generator model (see Fig. 5.3 on page 71) the `JoomlaGeneratorModel` class has an association `databaseTables` which represents the database tables that are derived from the GenClasses in the Joomla! generator model. In this section, we discuss the approaches used for mapping objects to tables in general.

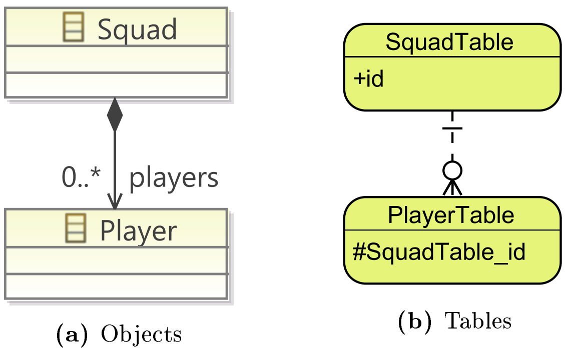


Figure 7.3: An example of the *foreign key association* pattern.

As mentioned by Keller (1997) [Kel97], object-oriented programming and the relational model are two different paradigms of programming. In order to store objects in relational databases, concepts of object-oriented programming must be mapped to relational table structures. These concepts are, cf. [Kel97]: aggregation, inheritance and polymorphism, associations between classes, and data types.

In the following, we only present the different patterns for mapping associations between classes to relational tables. For a more thorough exposition of all the patterns for mapping the different concepts of object-oriented programming to relational tables, the read is referred to [Kel97].

7.2.1 Associations of $0 : n$ Multiplicity

Consider the data model in the running example (see figure 2.6 on page 11). In figure 7.3a an excerpt from this model, namely the `Squad` and `Player` classes and the association between these classes, are shown. For the sake of simplicity the attributes of both classes have been omitted.

In order to map the scenario presented in Fig. 7.3a to relational tables, two tables are created, e.g. `SquadTable` for the `Squad` class and `PlayerTable` for the `Player` class. The association between the two classes are represented by a *foreign key* in the `PlayerTable`. To this end, a *primary key* is added to `SquadTable`. This mapping is depicted in Fig. 7.3b where the plus (+) symbol represents a primary key and the foreign key is represented by a hash (#) symbol. Furthermore, this figure represents the relationship between the two tables (or *entities*) which in this case is a $0 : n$ relationship between the squad and the player tables. This, however, should not be added to the table schema, but since we follow the notation in [Kel97], this is added here anyway.

The mapping in Fig. 7.3 on the facing page is an example of a *foreign key association* pattern, cf. [Kel97, pp. 21-22]. The general structure of this mapping is depicted in Fig. 7.4. This pattern maps an association of multiplicity $0 : n$ between two classes to relational tables. Notice that this pattern is also used in the case of $1 : 1$ or $0 : 1$ multiplicities; these are special cases in the $0 : n$ multiplicity. Furthermore, the association in Fig. 7.4 also represents a composition (like the one in Fig. 7.3a on the facing page).

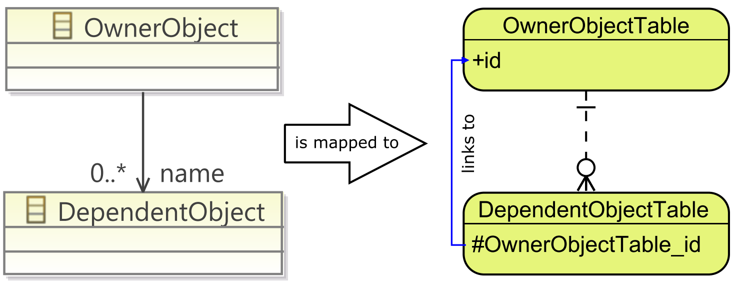
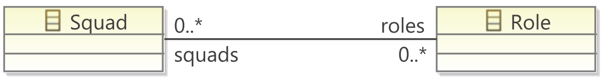


Figure 7.4: The general structure of the *foreign key association* pattern.

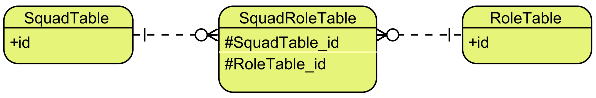
7.2.2 Associations of $m : n$ Multiplicity

In Fig. 7.5a on the next page an excerpt of the data model in the running example is shown. In this figure, the `Squad` and `Role` classes and the *bidirectional* association between them are presented. Like previous section, we will map this scenario to relational tables. In this case, similar to the case with foreign key association pattern, a table for each of the classes is created, e.g. `SquadTable` for the `Squad` class and `RoleTable` for the `Role` class. The association, in this scenario, is represented by a new table having two foreign keys. To this end, a primary key is added to the `SquadTable` and `RoleTable`. This mapping is depicted in Fig. 7.5b on the following page.

The mapping in figure 7.5 on the next page is an example of a *association table* pattern, cf. [Kel97, pp. 23-24]. The general structure of this mapping is depicted in Fig. 7.6 on the following page. This pattern maps a $m : n$ association between two classes to relational tables.



(a) Objects



(b) Tables

Figure 7.5: An example of the *association table* pattern.

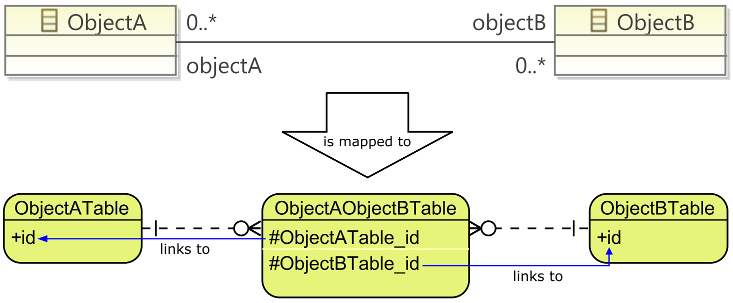


Figure 7.6: The general structure of the *association table* pattern.

7.2.2.1 Mapping Other Concepts

As mentioned above, the concepts of object-oriented paradigm that must be mapped to relational tables are aggregation, inheritance and polymorphism, associations between classes, and data types. Above, we presented two patterns for mapping associations to relational tables. Mapping of the remaining concepts, e.g. inheritance which is more interesting, is not covered here, since these concepts are not supported in the data model, e.g. inheritance. However, in the case of data types, as also mentioned in section 5.3.3.3 on page 66, they are simply mapped based on the type of the class attribute, e.g. the data type `int` (or more precisely `EInt`) is mapped to integer (or more precisely the `INT` data type in MySQL).

Extending Welipse with more patterns that address mapping of concepts mentioned above is yet another future work item.

7.3 Introducing the Database Generator Model

As mentioned in section 5.3.3.3 on page 66 and section 7.2 on page 91 above, the patterns for mapping objects to tables are applied in the Joomla! generator model. This means that these patterns must be applied to each PSM that is added to Welipse, e.g. Drupal (Drupal generator model). This has huge implication on the maintainability of Welipse, not to mention the redundant implementation of these patterns. To address this issue, we propose the introduction of yet another model; namely the *Database Generator Model*.

The database generator model is a PIM, i.e. it is independent of any platform such as MySQL, PostgreSQL, MSSQL etc. The patterns for mapping objects to tables together with any other platform independent properties are applied in this model. For instance, one of these platform independent properties would be to choose a specific pattern for mapping object inheritance to tables.

The introduction of new model also means the extension of the overall development process. Furthermore, new artifacts and features are also introduced along the way. In order to demonstrate how the introduction of the database generator model extend the development process with Welipse, we have extended Fig. 6.1 on page 76 in chapter 6 on page 75. The result is shown in Fig. 7.7 on the following page. As seen in this figure, the `DBGM Generator` (database generator model generator) takes the data model as input and produces the `Database Generator Model`. Actually, it will be an initialization the same way as the

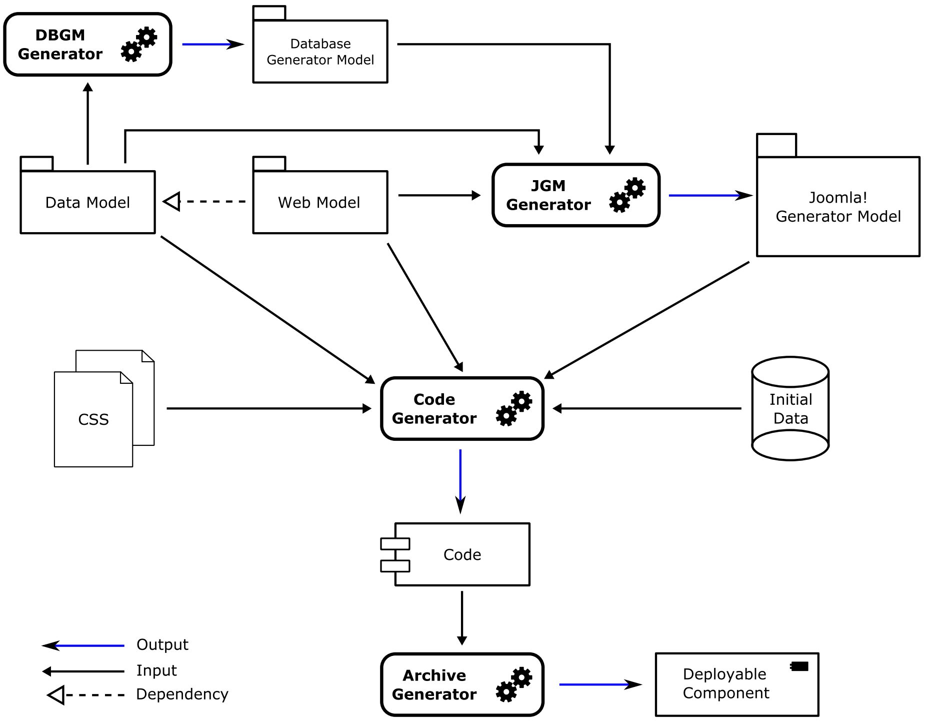


Figure 7.7: An overview of the different artifacts and their usage in the development process with Welipse when the Database generator model is introduced.

Joomla! generator model is initialized from data and web models. Notice, that in Fig. 7.7, the JGM Generator now also takes the Database Generator Model as input. This way, the database generator model can be customized such that it targets a specific platform, e.g. MySQL.

The reason behind not adding the database generator model to Welipse is, again, due to the time-limitation of this thesis. Future work must add this model before extending Welipse with yet another PSM.

7.4 Summary

In this chapter, we discussed all major design decisions and the related arguments with respect to the design of the tool. First, we elaborated on the design of Joomla! generator model. Next, mapping objects to tables was discussed. Finally, it was mentioned that the separation of database tables generation from Joomla! generator model is a better approach; using the so-called Database generator model.

CHAPTER 8

Implementation of Welipse

In this chapter, we present the implementation of non-trivial parts of Welipse. First, we present the main components of Welipse through a component diagram. Next, we present an overview of Eclipse projects that contain the implementation of various parts of Welipse. Finally, each the implementation of non-trivial parts of Welipse are discussed.

8.1 Main Components

The implementation of Welipse can be divided into six main components; namely *Data Model*, *Web Model*, *Joomla! Generator Model*, *Expressions Language*, *Code Generator*, and *Welipse UI*. These components are shown in a component diagram in Fig. 8.1 on the next page. As seen in this figure, the Data Model component is shaded. This is in order to emphasize that the Data Model is not implemented by us but instead is part of EMF-models which we use in Welipse.

Let us explore the diagram in Fig. 8.1 on the following page a little bit more. Bear in mind that this diagram is only meant to give a conceptual view of the implementation of Welipse, and it must not be considered as an actual picture of the implementation. For instance, the ports (the small boxes) of the compo-

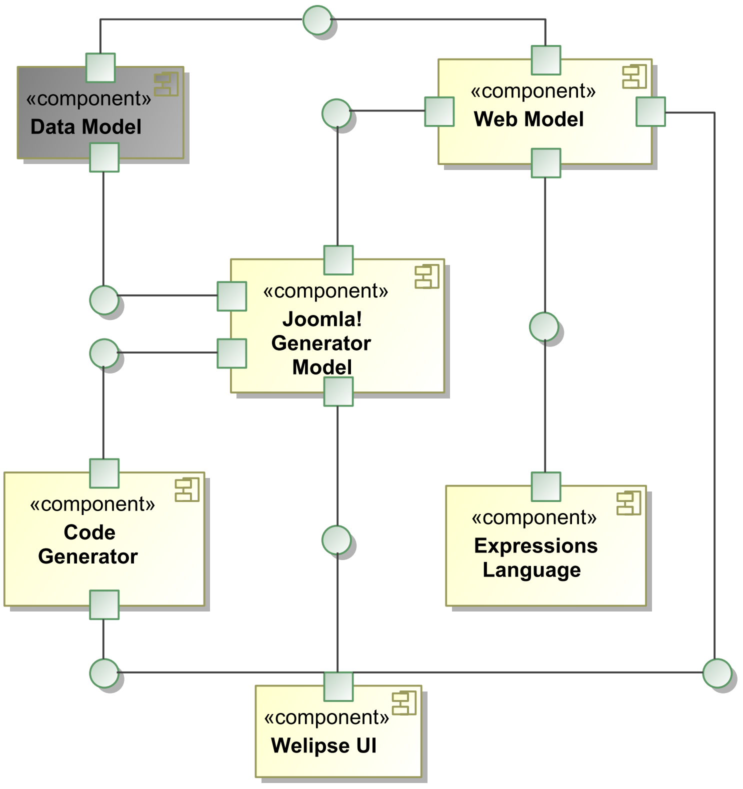


Figure 8.1: An overview of the main components of Welipse as a component diagram.

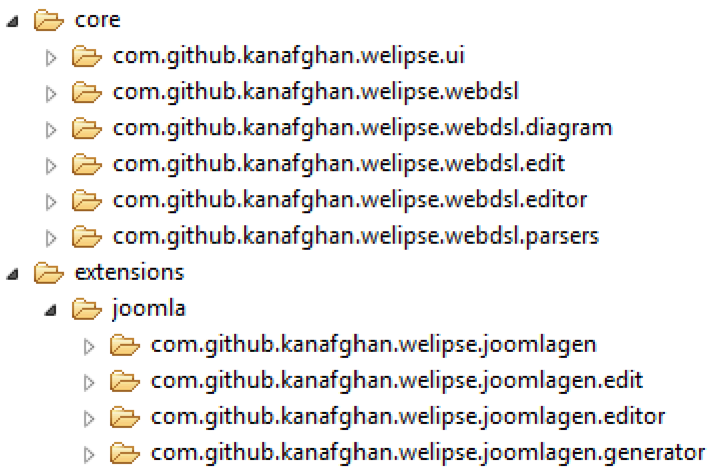
nents and the interfaces (the small circles) between these ports only represent a conceptual picture of communication between the components and not actual interfaces, e.g. Java interfaces. Another thing to bear in mind regarding the diagram in Fig. 8.1 is the fact that the Web Model and the Joomla! Generator Model components are interconnected with corresponding editors which also correspond to main components of Welipse. These are not presented in the diagram in Fig. 8.1 for the sake of simplicity and in order not to clutter the diagram. As indicated in the diagram in Fig. 8.1, the Data Model component is interconnected with the Web Model component only. However, in Welipse both the Joomla! Generator Model and the Code Generator components have access to the Data Model component, but indirectly through the Web Model component. This is also the case in the case of the Web Model and Code Generator components; the Code Generator component accesses the Web Model component indirectly through the Joomla! Generator Model component. The interface between the Web Model and the Expression Language components is representing a "bidirectional" communication, i.e. both components have both *provided* and *required* interfaces. The Welipse UI component consists of common user interface features of Welipse. For instance, this component contains an Eclipse wizard category for the various Eclipse wizards provided by other components, e.g. such as Web Model component.

8.2 Organisation of the Implementation

Since we are using EMF and other Eclipse based development tools, the implementation of Welipse has been divided among various Eclipse *plug-in* projects. For the time being, there are ten projects comprising the Welipse tools. These projects are further divided into two parts, a *core* part and the *extensions* part. The core part comprises the projects that contain the implementation of the web modeling tools, while the extensions part contains the implementation of various platform-specific tools targeting a specific platform. In the moment, this part only contains the implementation of tools that target the Joomla! platform. In Fig. 8.2 on the next page an outline of this organisation is given.

The naming of the projects, i.e. `com.github.kanafghan.welipse.x`, is based on the fact that the implementation of Welipse is put on *GitHub*[1] as open source code and can be cloned by anyone that is interested. In the following, we present the various parts of the implementation of Welipse in more details and will be referring to the above organisation as needed.

[1]The Welipse project is found on `https://github.com/kanafghan/welipse`



```
core
    com.github.kanafghan.welipse.ui
    com.github.kanafghan.welipse.webdsl
    com.github.kanafghan.welipse.webdsl.diagram
    com.github.kanafghan.welipse.webdsl.edit
    com.github.kanafghan.welipse.webdsl.editor
    com.github.kanafghan.welipse.webdsl.parsers
extensions
    joomla
        com.github.kanafghan.welipse.joomlagen
        com.github.kanafghan.welipse.joomlagen.edit
        com.github.kanafghan.welipse.joomlagen.editor
        com.github.kanafghan.welipse.joomlagen.generator
```

Figure 8.2: An overview of the projects in Core and Extensions part in the implementation of Welipse.

8.3 Web Modeling Tools

As we mentioned above, the core part of the implementation of Welipse contains the web modeling tools. These tools are model editors that are used to manipulate the web models and the *expressions language* integrated in these tools. The implementation of expressions language is presented later in this chapter. Here, we discuss the implementation of editors very briefly, since these are mostly generated automatically from their corresponding Ecore meta-models.

Welipse, at the current version, provides two kinds of editors, a tree editor for creation and manipulation of web models in a tree structure and a graphical editor that implements the graphical concrete syntax of web concepts presented in section 4.2 on page 33.

The tree editor is implemented using EMF tools, while the graphical editor is implemented using GMF tools. In the following, we will only briefly describe the approach used to implement these editors and will not go into details with the various technologies, as it is not in the scope of this project.

8.3.1 Implementation of Tree Editor

In order to implement the tree editor for web models, EMF is used. EMF provides Ecore model for defining the abstract syntax of the DSL (also called meta-modeling). Using the Ecore model together with the Gen model, one can generate code for fully functional tool set for working with the DSL. The tool set comprises, among others, a tree editor for creating and manipulating models that conform to the meta-model supported by the tool set. For instance the meta-model (the so called *WebDSL*) of web models is created using one Ecore model. This model was extended with various web concepts in many iterations in the course of this thesis. The tree editor generated from this model makes it possible to create and manipulate web models, load them from a file or save them to one. The Gen model corresponding to WebDSL is used to configure the generation of code for the resulting tool set.

The WebDSL (webdsl.ecore) together with the corresponding Gen model (webdsl.genmodel) are located in project `com.github.kanafghan.welipse.webdsl` under the `model` folder. The Gen model is used to generate both *model*, *edit* and *editor* codes. The code for model is located in the same project as the Ecore and Gen models. The edit and editor codes, however, are located in separate projects. These projects are `com.github.kanafghan.welipse.webdsl.edit` and `com.github.kanafghan.welipse.webdsl.editor`, respectively.

The generated code for model together with edit codes is changed manually in order to implement the `type()` and `initialize(Page)` methods of expressions and variables concepts. These changes are elaborated on in section 8.3.3 on the next page in the context of expressions language.

8.3.2 Implementation of Graphical Editor

In order to implement the graphical concrete syntax of web DSL, GMF is used. The GMF models, i.e. , are located in `com.github.kanafghan.welipse.webdsl` project, while the generated code for the diagram is located in `com.github.-kanafghan.welipse.webdsl.diagram` project. This code has not been changes manually.

8.3.3 Expressions Language

The syntax and semantics of expressions were presented in section 4.2.2 on page 42. In this section, we discuss how these are implemented.

The way expressions are designed and how they are used is similar to a textual programming languages. Therefore, the implementation approach we have chosen for implementing expressions is also similar to the one used for implementing programming languages. In this approach the syntax of the language is analyzed on three levels: the lexical, the syntactic, and the semantic analysis. In the first level, i.e. lexical analysis, a lexer (also called tokenizer or scanner) converts a sequence of characters into a sequence of tokens. These tokens are processed by the so called *parser* into a abstract syntax tree. This is known as parsing and is the second level, i.e. syntactic analysis. On the third level, i.e. semantic analysis, the names of objects and/or variables are resolved and types are checked in a given context.

From above, we can see, that in order to develop the expressions language we must have a lexer and a parser that can generate the abstract syntax tree for our expressions. This task can be accomplished either by using a so called *parser generator* technology or coding both the lexer and the parser from scratch. Although, our expression language is very simple, the later approach is not considered at all since it is both error prone and time consuming. Furthermore, it makes it difficult to extend the language compared to a parser generator technology. In the following, we discuss which technology we have chosen for this task and the reasons behind it.

8.3.3.1 Parser Generator Technologies

We have investigated three technologies in this field that all fulfill the basic criteria mentioned in section 5.3.2.1 on page 64. These technologies are *Xtext* [Foue], *JavaCC* [aia] and *ANTLR* [Par]. Our choice became ANTLR mainly because of its simplicity and in the same time being very powerful. Furthermore, it provides exactly what we need, no more and no less. Xtext on the other hand would also have been a great choice but it provides more than we need, i.e. it also generates compiler and a powerful editor for the language. Moreover, it is also based on ANTLR. Both ANTLR and Xtext are familiar technologies to us, while we have not encountered JavaCC before. It was one of the reasons that JavaCC was not chosen. Moreover, its decreasing popularity and less support from the community around it, was another reason that it was not chosen.

The details about ANTLR and how the expressions are parsed type checked and linked (semantic analysis) are presented in the following.

8.3.3.2 ANTLR in a Nutshell

Terence Parr the man behind ANTLR (ANother Tool for Language Recognition) defines ANTLR as a powerful parser generator for reading, processing, executing, or translating structured text or binary file [Par]. It can, from a grammar, generate a parser that can build and walk parse trees. ANTLR is a so-called $LL(*)$ parser generator [PF11]. In other words, it generates a top-down parser for a given grammar with infinite lookahead. In the following, we very briefly present the way ATNLR works. For more details, please refer to [Par].

Consider a simple grammar for CSV (Comma Separated Values) files in the EBNF syntactic metalanguage [ISO96]:

```
csv-file = {csv-row}-;
csv-row = csv-value {',' csv-value}
         (['<CR>'] '<LF>' | '<CR>' | <EOF>);
csv-value = csv-simple-value | csv-quoted-value;
csv-simple-value = {-(',' | '<CR>' | '<LF>' | '"')}-;
csv-quoted-value = '"' {'""' | -'"'} '"';
```

where <EOF>, <CR> and <LF> mean end of the file, carriage return, and line feed, respectively. The coloring indicates terminals (blue) and non-terminals (black). This grammar can parse the following kinds of files:

```
value1,value2,...,value3
value1,"multi line text
only in quotes",...,value3
...
```

ANTLR does not only generate a parser for a given grammar. Rather, it generates both a lexer and a parser. In the following, the above grammar is presented in the syntax of ANTRL.

```
grammar CSV;

file: row+;
```

Character	Meaning	Example	Matches
|	Alternative	`'a'	'b'`	Either `'a'` or `'b'`
?	Optional	`'a' 'b'?`	Either `'ab'` or `'a'`
*	0 or more	`'a'*`	Nothing, `'a'`, `'aa'`, ...
+	1 or more	`'a'+`	`'a'`, `'aa'`, ...
~	Match not	`~('a' | 'b')`	Any character except `'a'` and `'b'`
(···)	Subrule	`('a' 'b')+`	`'ab'`, `'abab'`, ...

Table 8.1: Meta-characters in ANTRL

```
    row: value (COMMA value)* (LINE_BREAK | EOF);

    value: SIMPLE_VALUE | QUOTED_VALUE;

    COMMA: ',';

    LINE_BREAK: '\r'? '\n' | '\r';

    SIMPLE_VALUE: ~(COMMA | LINE_BREAK | '"')+;

    QUOTED_VALUE: '"' ('""' | ~'"')* '"';
```

The first line of the above code defines a *grammar* with the name CSV. From this, ANTLR generates a `CSVLexer.java` and `CSVParser.java`. The remaining lines, having the `rule-name:  rule-definition;` form, are either a parser rule or a lexer rule. In fact, the three top rules, i.e. `file`, `row` and `value`, are all parser rules, while the rest are lexer rules. The lexer rules must be distinguished from the parser rules by naming them with initial letter capitalized. The terminals in the grammar are surounded with single quote characters. In addition, there are other meta-characters in ANTLR which are summarized in table 8.1. The generated lexer and parser can be used as demonstrated in the following Java code snippet:

```
...
    // the input source
    String source = "...";

    // create an instance of the lexer
    CSVLexer lexer = new CSVLexer(new ANTLRStringStream(source));

    // wrap a token-stream around the lexer
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // create the parser
    CSVParser parser = new CSVParser(tokens);

    // invoke the entry point of the grammar
    parser.file();
...
```

The power of ANTLR lies in the ability to integrate Java code as *actions* in the rules of the grammar. This way the ANTLR can return a data structure (the abstract syntax tree) that contains the parsed string which can be analyzed further, e.g. semantic analysis. For instance, in the above grammar, a CSV file could be represented as a two-dimensional collection of strings where each one-dimensional collection will represent a row in the file and an element of the collection will represent a value in the file. This is demonstrated in the following by extended the definition of the parser rules.

```
...
    file returns [List<List<String>> data]
        : row+
         {...}
        ;

    row returns [List<List<String>> row]
        : value (COMMA value)* (LINE_BREAK | EOF)
         {...}
        ;

    value returns [String value]
        : SIMPLE_VALUE | QUOTED_VALUE
         {...}
        ;
...
```

In ANTLR, each parser rule can return an object by placing `returns [Object object]` after the rule name. In the definition of the rule, Java code snippet can be inserted inside curly braces (`{...}` as shown in the above code) in order to populate the data structure.

In the following, we consider the actual implementation of the expressions language using ANTRL.

8.3.3.3 Expressions Parser

The parser (which is both the *lexer* and the *parser*) for the expressions language has been implemented in a separate Eclipse plugin. This plugin is found under the core part of the implementation of Welipse in the `com.github.kanafghan.-welipse.webdsl.parsers` project. The grammar of the expressions language, which was defined in section 4.2.2.3 on page 46, has been defined in ANTRL in the file `Expressions.g` in the same project. This file also contains two rules for parsing parameters and variables concepts which were specified in section 4.2.2 on page 42. The remaining of this file is straight forward, except the way precedence of operators have been defined. In the following, an excerpt of the `Expressions.g` file has been presented, which represents the way precedence of operators have been defined.

```
...
expression returns [Expression result]
    :    term11 {$result = $term11.result;}
    ;

term0 returns [Expression result]
    :    variableUse {...}
    |    '(' expression ')' {...}
    |    constantExpression {...}
    |    classifierOperation {...}
    |    structuralExpression {...}
    |    listExpression {...}
    |    webUtilsExpression {...}
    ;

term1 returns [Expression result]
   ...
   ;

term2 returns [Expression result]
   ...
   ;
...
term11 returns [Expression result]
   ...
   ;
...
```

The above listing shows that the `expression` rule consists of twelve levels. At the top most level, i.e. `term0`, an expression is either a variable expression, or a parenthesised expression, or a constant expression, and so forth. In total, there are seven kinds of expressions. One level below that, i.e. `term1`, can be a boolean negation operation, while even a level lower, i.e. `term2`, we can have an arithmetic negation operation. This continues downward in the order of precedence of operators presented in table 4.1 on page 49. Notice, that there are ten levels in table 4.1, however, in the above listing eleven levels are presented. The last level, i.e. `term11`, represents an element in either a simple list or associative list.

For the sake of convenience and high cohesion, the generated lexer and parser has been wrapped in a singleton class `ExpressionsLanguage.java` which is also located in the same project as the `Expressions.g` file. This class takes some text as input and provides methods for returning the result as either expression, variable or parameter. The result is an interface that other plugins can benefit from without the need for ANTLR binary residing in each plugin. In the following, we present the way expressions are analyzed, e.g. type checked, and how these are integrated with web model editor.

8.3.3.4 Semantic Analysis and Integration with Web Model Editor

As we mentioned above, once the expressions are parsed, they must be type checked and analyzed based on the context they are used in, i.e. semantic analysis. In the meta-model of web model presented in Fig. 4.13 on page 54, the `Expression` class has an operation `type()` which is defined for the purpose of type checking. Using this operation, the type of expression can be determined in a recursive manner. However, before we can type check an expression, we must set the various elements that has been referenced from the data model and resolve symbols, e.g. variables and parameters call. In other worlds *link* the elements from data model to elements from web model. For instance, the expression `player.name` consists of two parts, the variable part `player` which must point to a parameter or variable called `player`, and a property part called `name` which specifies a structural feature that must belong to the type of the `player`. In order to find this structural feature and thus determine the type of the second part in the above expression, we must find the type of the first part which, in turn, is determined by the type of the parameter or variable `player`. To this end, we have added yet another operation to the `Expression` class, namely `initialize(Page)`[2], which takes a `Page` as argument and, de-

[2]The name of this method is not descriptive enough. The implementation of this method does some of the semantic analysis, i.e. resolving names, and the linking between the data and web model elements.

pending on the kind of expression, sets the various elements referenced from the data model in recursive fashion. The context of this "initialization" is given by the page as argument. For instance, it will set the `name` attribute of the `Player` class as the value of the `feature` reference of the structural expression mentioned above. The `initialize(Page)` operation is also added to the `VariableDeclaration` class for the same purpose. In this case, however, this operation is implemented for the "initialization" of parameters and variables. The type of the the parameters and variables are determined when they are declared.

The `initialize(Page)` operation is shown in Fig. 8.3 on the facing page. This figure is elaborated on in the following. Here, we discuss the implementation of `initialize(Page)` method. This implementation does not comply to the usual compiler construction and the principles and practices related to it. Normally, a so-called *symbol table* is used in order to resolve names of variables, functions, classes, and so fourth, while doing syntactical analysis. In the implementation of `initialize(Page)` method, this symbol table is calculated each time the method is called using the given page as argument. This is a very inefficient way of addressing this issue, nevertheless it is a fast and easy way. Therefore, we have chosen this implementation in order to boost the over all implementation of the tool. This is also an obvious issue to be addressed with regard to future work.

Integration of Parser in Web Model Editor From section 4.2.2 on page 42, we know that many of the concepts of web model contain one or more expressions. For instance, the Text concept contains one expression, while the Selection List concept contains three expressions. These expressions are provided in a textual form using the defined syntax, e.g. `player.name`. This text, *expression text*, is parsed and the abstract syntax tree that results from this parsing is contained by the concepts of web models. The user of the tool must provide the expression text when defining web models in one way or another. To this end, we have extended the meta-model of web model by adding attributes to those classes that represent concepts that contain expressions. This is shown in Fig. 8.3 on the facing page. Here, the `expression` attribute of the `PresentationElement` class is used for providing expression text of Text, Image and List concepts. In the case of Text concept, for instance, the expression text for defining `content` is provided in `expression` attribute.

In the case of Input concept which contains two expressions, i.e. `label` and `value`, an attribute corresponding to each expression is added, i.e. `label-Expression` for `label` and `valueExpression` for `value` (see Fig. 8.3). The same idea is applied to the parameters and variables of the Page concept and

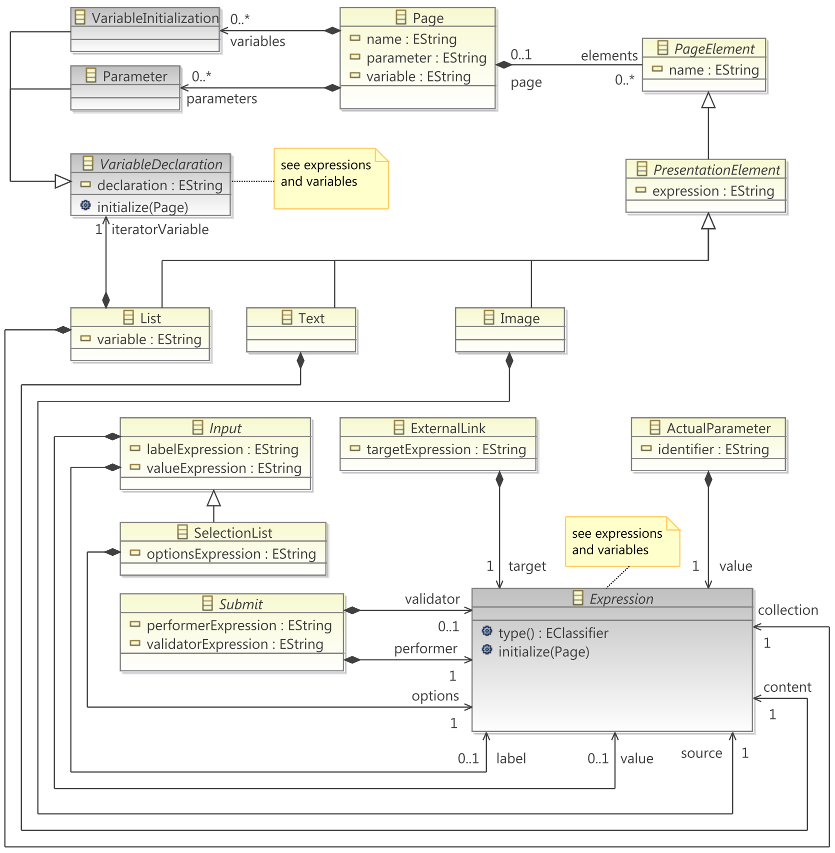


Figure 8.3: The meta-model of web model with the implementation related changes.

the iterator variable of the List concept, as shown in Fig. 8.3.

By extending the generated code for the web model editor, it is possible to integrate the expressions parser with web model editor. By editor, we mean the tree editor generated by EMF. We have integrated the expressions parser by specializing the `ItemPropertyDescriptor` class of EMF and overriding the `setProperty()` method of this class. The actual implementation is found in `ParseableItemPropertyDescriptor` class in the `com.github.kanafghan.-welipse.webdsl.edit` project. The `ParseableItemPropertyDescriptor` class is used instead of `ItemPropertyDescriptor` in the item providers generated for the various concepts that contain expressions. For instance, in the case of the List, Text and Image concepts, this class is used in the item provider of `PresentationElement` class since the List, Text and Image concepts inherit the `expression` attribute from `PresentationElement`. Likewise, in the case of Input concepts, it is in the item provider of `Input` class where `ParseableItem-PropertyDescriptor` is used.

Once the expression text is entered by the user in the editor (through the Properties View), the resulting abstract syntax from this expression text is achieved through a number of steps. First the expression text is parsed by using the parser discussed above. Next, the symbols within expression is resolved and linked to the corresponding elements (by the `initialize(Page)` operation discussed above). Thereafter, the type of the expression is checked (by the `type()` operation of the expressions). Finally, the expression is analyzed semantically. By semantic, we mean in which context the expression is being used. For instance, a variable expression will not make sense as the collection property of the List concept; it must be a classifier operation that returns a collection of elements of the same type. The above mentioned analysis are implemented in the `ExpressionsAndVariablesAnalyzer` class which is located in `com.git-hub.kanafghan.welipse.webdsl.edit` project.

The above approach concerning the extension of the meta-model in order to provide a way for providing expression text has both advantages and disadvantages. The disadvantage is the cluttering of meta-model with implementation specific information. The meta-model must always be kept clean with respect to implementation specific information. The advantage of the above approach is the simplicity of the implementation and fast way of doing semantical analysis. A better approach for addressing the same issue has been shown in Fig. 8.4 on the next page. As seen in this figure, the various concepts of web model contain the `ExpressionText` class instead of the `Expression` class. In this approach, the expression text is provided through the `text` attribute of the `ExpressionText` class which also contains the abstract syntax tree of the textual expression through the `expression` reference. Due to time-limitation, we were not able to implement this approach which requires more manual work

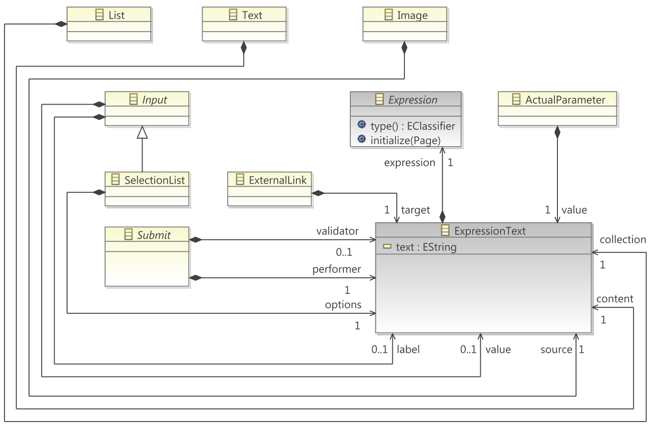


Figure 8.4: The alternative approach for providing expression text.

compared to the implemented approach. The implementation of this approach is definitely an obvious issue which can be considered as future work.

8.4 Joomla! Generator Model

As mentioned above (see section 8.2 on page 101), at the moment the extensions part only contains the implementation of tools that target the Joomla! platform. These tools consist of a tree editor for working with the Joomla! generator model, a wizard used to initialize the Joomla! generator model and a code generator that generates a runnable Joomla! 2.5 component.

We presented the meta-model of Joomla! generator model in chapter 5 on page 61 and discussed its design in chapter 7 on page 87. In the following, we will use this model in order to describe the implementation of the tools mentioned above. This model (joomlagen.ecore) is located in the `com.github.kanafghan.welipse.joomlagen` and defined using an Ecore model.

8.4.1 Configuration of Gen Model

In the following, we briefly cover the details specific to the implementation of the tree editor for Joomla! generator model. These details relate to how the Gen model (joomlagen.genmodel) of joomlagen.ecore has been configured. For most of the parts the default configuration of the Gen model is chosen, except the `DatabaseTable` class. This class is not possible to create in the tree editor since its edit mechanism has been disabled. The reason for this is that the database tables are automatically created based on the information in the GenClasses of the Joomla! generator model. This information is for instance the name of the database table, its columns and foreign keys. Since this information must be in sync with that in the database tables they must be editable only in one place. Furthermore, the purpose of the DatabaseTable class is to be used as data structure for code generation. We will elaborate on this in the following.

8.4.2 Initialization and Reconciliation

The generated code that forms the tree editor and the wizard for initializing an instance of the Joomla! generator model are located in the `com.github.kan-afghan.welipse.joomlagen.edit`, `com.github.kanafghan.welipse.joomla-gen.editor` projects. The generated model code is located in the same project as the joomlagen.ecore and joomlagen.genmodel. This code and the generated code in the editor project will be discussed in more details in the following, since we have extended this code manually in order to make the process of creation and reconciliation of an instance of Joomla! generator model automatic.

As mentioned in section 7.1 on page 87, the design of Joomla! generator model is based on the design of the EMF Gen model. Likewise, the implementation of the Joomla! generator model is based on the implementation of this model. In the following, we first discuss the implementation of the wizard for initialization and reconciliation of Joomla! generator models and later discuss the manual code that has been added to the generated code in order to make the initialization and reconciliation of Joomla! generator models automatic.

The default wizard (the so called New Wizard) that is generated by the EMF Gen model for creating instances of a model such as Joomla! generator model consists of two pages. The first page of the wizard is used to create the file that will contain the model, while the second and final page of the wizard is used for selecting the Model Object to be created as the base object. This process of selecting a model object is not needed in the creation of Joomla! generator model. Instead, we want the user to select and then load a web model that will be used to initialize the Joomla! generator model. This is exactly the same as the creation of an EMF Gen model instance.

The initialization and reconciliation of Joomla! generator model is initiated by the same wizard. There are not two wizards for these purposes. This wizard (`JoomlaGenModelWizard`) is located in the `com.github.kanafghan.welipse.-`

`joomlagen.editor` project and is the overriding of the default New wizard generated by EMF. In order to implement the behaviour described above, we have altered this class by changing the final page of the wizard. The implementation of this page is in the inner class in the JoomlaGenModelWizard. Furthermore, the `init()` method of the JoomlaGenModelWizard class has been altered in order to cope with the reconciliation case, i.e. depending on the `selection` given to this method it will either initiate the wizard for creating a new Joomla! generator model or reconcile an existing one.

8.4.2.1 Implementation of Initialization Process

The process of initialization of a Joomla! generator model begins when the web model is loaded. Through the web model the data model is retrieved and its elements are wrapped in the corresponding Joomla! generator model element, i.e. an EClass in data model is wrapped in a GenClass, an EAttribute in a GenAttribute and so fourth.

As also shown in the meta-model of Joomla! generator model (see Fig. 5.3 on page 71), the data model is represented by a GenPackage which wraps an E-Package from the ECore model. Hence, the data model must be contained in one EPackage.

Each of the GenPackage, GenClass, GenAttribute, GenReference, GenOperation, GenParameter, and GenDataType classes has an `initialize()` method which takes the corresponding Ecore element as argument. Using this method the initialization process is done in recursive manner, i.e. the initialize() method of GenPackage is called with the argument EPackage that represents the data model and the EClasses in this package are wrapped in the corresponding GenClasses and their initialize() method is called. This process is repeated for the attributes, references and operation of each class in their respective initialize() methods. This way the Ecore elements are wrapped recursively in Gen elements of Joomla! generator model.

8.4.2.2 Implementation of Reconciliation Process

The reconciliation process is initiated by a context menu action (`Reload...`). The implementation of the handler for this action is in class `ReloadGenModelHandler` in the same project as the wizard described above.

The implemented reconciliation process which is also based on the EMF Gen model implementation, is in a nutshell as follows: the loaded Joomla! genenerator model is adjusted first, i.e. missing elements are removed, and then it is marked as `oldGenModelVersion` and a new model is created based on the web model and data model referenced by the oldGenModelVersion. The configuration of the oldGenModelVersion is transferred to the newly created model. In other words, the reconciliation process takes into account the deletion and addition of new elements in the web and data models. Notice, that the loaded Joomla! generator model is assumed to need update and, thus, no checking is done to insure whether this is the case. It might be a good idea to do this checking in order to optimize the process. In our implementation, we have disregarded this due to the rapid prototyping of the tool.

Like the implementation of the initialization process, the reconciliation process is implemented recursively. In this case, however, the process begins in the JoomlaGenModel by calling its `reconcile()` method which takes the oldGenModelVersion as argument. The implementation of this method, in turn, calls the reconcile() method of the data model and the recursion continues until GenAttributtes, GenReferences and GenParameters.

8.4.3 Mapping Objects to Tables

In the following, we discuss the implementation of mapping objects to tables, i.e. generating the database tables from the data model. In section 7.2 on page 91, we presented some patterns for mapping objects to tables. The implementation of these patterns will be also discussed here.

As mentioned above, it is not possible to create an instance of the `DatabaseTable` class in the editor, since this class is used as the data structure for database tables resulting from mapping objects to tables. This also means that the `databaseTables` reference of the `JoomlaGenModel` class (see Fig. 5.3 on page 71) is a derived reference. The actual implementation of mapping objects to tables begins in the *getter* method of this reference. This implementation is found in `JoomlaGenModelImpl` class which is located in the `com.github.kanafghan.-welipse.joomlagen` project.

The implemented algorithm for mapping objects to tables can be summarized as follows. For each class c in the data model a table t is created. For each attribute of c, if it is not a derived attribute, a column is added to t, while for each non-derived reference of c, depending on the multiplicity, one of the patterns *foreign key association* and *association table*, is applied. If the reference has multiplicity $1 : 1$ (or $0 : 1$) it is added to t and the opposite reference, if any, is remembered. The case where the reference has multiplicity $1 : n$ (or $0 : n$), for $n > 1$, the foreign key association pattern is applied, while the association table pattern is applied if the reference has multiplicity $n : m$.

Notice that in the above algorithm, we divided the $1 : n$ multiplicity into two cases: $1 : 1$ and $1 : n$ for $n > 1$ and applied the foreign key association pattern only to the later case. The case $1 : 1$ is handled straightforward by adding a foreign key to the owner object table, only if the opposite reference (in the case of bidirectional association), if any, is also $1 : 1$. The bidirectionality of the associations is also taken into account in other cases, i.e. $1 : n$ and $1 : m$ multiplicities.

Before the above algorithm is applied, the data model is reconciled. This way, any addition and deletion of data model elements are taken into account before the database tables are created.

8.5 Code Generation

In the following, we discuss the implementation of code generation for the Joomla! 2.5 platform. First, we present the Joomla! MVC (Model-View-Controller) implementation. Next, the Joomla! component structure is presented. Finally, the actual implementation of the code generators and the technology used for the implementation is presented.

8.5.1 Joomla! MVC

As mentioned in section 3.3 on page 22, one of the reasons the Joomla! platform was chosen as the default platform was due to its MVC implementation. In the following, we will present this implementation in more details. This will help in understanding the way code generation for the Joomla! 2.5 component is implemented.

The MVC pattern, which is an architectural pattern, is used primarily in creating graphical user interfaces (GUIs). The major promise of the pattern is based on modularity; separation of the three aspects of the GUI, namely model, view and controller. The model aspect is the data while the visual representation of the data is the view aspect. The interface between the view and the model is the controller aspect. By keeping these three components (model, view and controller) as independent of each other as possible, advantages such as reusability of application's logic can be achieved. This advantage is exploited in the Joomla! MVC implementation where many views can be created using the same model. Exactly how this is achieved will become obvious later in this section.

The interactions between the three components in Joomla! MVC are illustrated in Fig. 8.5 on the facing page. In order to understand these interactions, we will refer to the generated Joomla! component in the running example (see section 6.5.1 on page 82). There, it is shown that in order to access the generated Squad page that shows the *The Azzuri* squad, the following URL is used:

```
<joomla-base>/?option=com_vivoazzurro&view=squad&squad=1
```

Where the `<joomla-base>` represents the location where Joomla! is installed, e.g. `http://localhost/thesis`. The query part of this URL is interesting here, i.e.

```
?option=com_vivoazzurro&view=squad&squad=1
```

This can be broken down into `option=com_vivoazzurro`, `view=squad`, and `squad=1`. From `option=com_vivoazzurro` the Joomla! knows that control should be handed over to the *vivoazzurro* component which is the application that contains the Squad page. Since this component is implemented according to the Joomla! MVC the *main controller* of the component will receive the control. This controller will pass the control over to the Squad view due to the `view=squad` part in the query. The view will retrieve the necessary data from the model in order to render the page. In this case, due to `squad=1`, the squad identified by number 1 is retrieved. Actually, this part represents the way parameters of a page are initialized. In this case, since the Squad page has one parameter (see `squad:Squad` in Fig. 4.2 on page 28), we only have `squad=1`.

In order to provide additional power and flexibility to web designers Joomla! splits the view into view and layout. In this way the data pulled by the view is made available to the layout which is responsible for formatting the data for presentation to the user. The advantage of this split is exploited in Joomla!'s template system which provides a simple mechanism for layouts to be overridden in the template[3].

The above scenario was a particular request made by the user in order to retrieve a particular page. Another and most common scenario is the request which

[3]The Joomla! template system is not in the scope of this thesis and thus not covered here. The interested reader is referred to Joomla!' documentation: `http://docs.joomla.org`.

contains form data. In the running example, an example of a form is given in the Player page (see Fig. 4.3 on page 30). In this form, when the *Subscribe* button is clicked a request is made with the following query:

```
...?option=com_vivoazzurro&task=doAction...
```

Here, the interesting part is `task=doAction`. From this, the main controller knows that the function *doAction()* must be called in order to process the form data. In the *Add a Player* form (see Fig. 4.6 on page 34) this part will be `task=player.save`. In this case, the main controller will pass the control to the *player* controller which, in turn, will call its *save()* function in order to process the form data. In both cases, the request will be redirected in the task function (the doAction() or save()) either to the same view where the request was originated from or to another view.

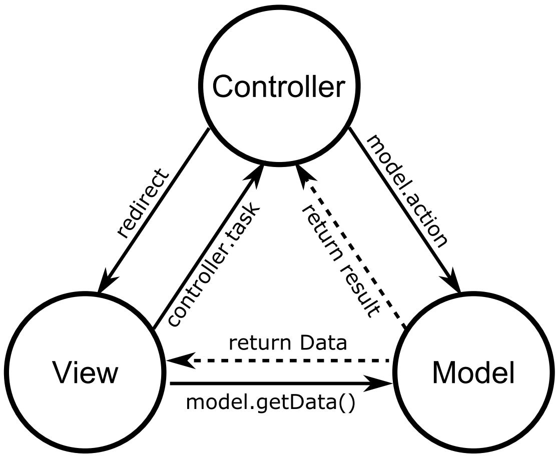


Figure 8.5: The interactions between Model, View and Controller in Joomla! MVC.

8.5.1.1 Joomla! MVC Framework Architecture

Joomla! provides a framework for implementing the MVC pattern. The architecture of this framework is shown in Fig. 8.6 on page 121. In order to understand this architecture, it is important to know some of Joomla!'s terms. In Joomla! the administrator area of the application is referred to as the back-end or admin whereas the public area which is meant for the end-user is called front-end or site. Joomla! comes with three kinds of *applications*, namely *administrator*, *site* and *installation*. The administrator and site applications are the implementation of back-end and front-end, respectively. The installation application is used during the installation process of a Joomla! CMS and is discarded immediately after the installation is finished. The notion of application in the Joomla! framework is not in the scope of this thesis and will not be discussed further.

As seen in Fig. 8.6 on the facing page, the component framework is a part of the Joomla! application. The three classes, `JModel`, `JController`, and `JView`, as the name suggests, represent the three components in MVC. In Joomla!, both the controller and the view reference the model, whereas the model is totally independent of both the controller and the view. This is a "passive" implementation of MVC, i.e. the controller and view must poll the model for updates rather than being notified by the model.

As also seen in Fig. 8.6 on the next page, a number of specialised classes are provided by the framework. These classes are mainly used in the back-end for implementing the *content managers*. We will cover this in next section. The `JComponentHelper` is helper class used by the Joomla! application. We will not go into details with this class either, as it is not in the scope of this thesis.

8.5.1.2 Joomla! 2.5 Component Structure

In order to understand the code generators implemented for generating a Joomla! 2.5 component, we must first understand the structure of such a component. In the following, we present this structure which is based on the Joomla! MVC framework.

At the outer most level a Joomla! component consists of three packages and a manifest file which is an XML file. This organisation is shown in Fig. 8.7 on the facing page. The three packages, as shown in this figure, are front-end (site), back-end (admin) and media. The two first, as the names suggest, represent the implementation of front-end and back-end of the component, respectively. The manifest file which is named after the name of the component or just `manifest.xml` contains the information about the content of the component. This file is used by the Joomla! component installer during the installation process. In the following, we will elaborate the back-end, front-end and media packages.

Front-End An overview of the content of the Front-End package is shown in Fig. 8.8 on page 122. As it is seen in the figure, each of the three components of

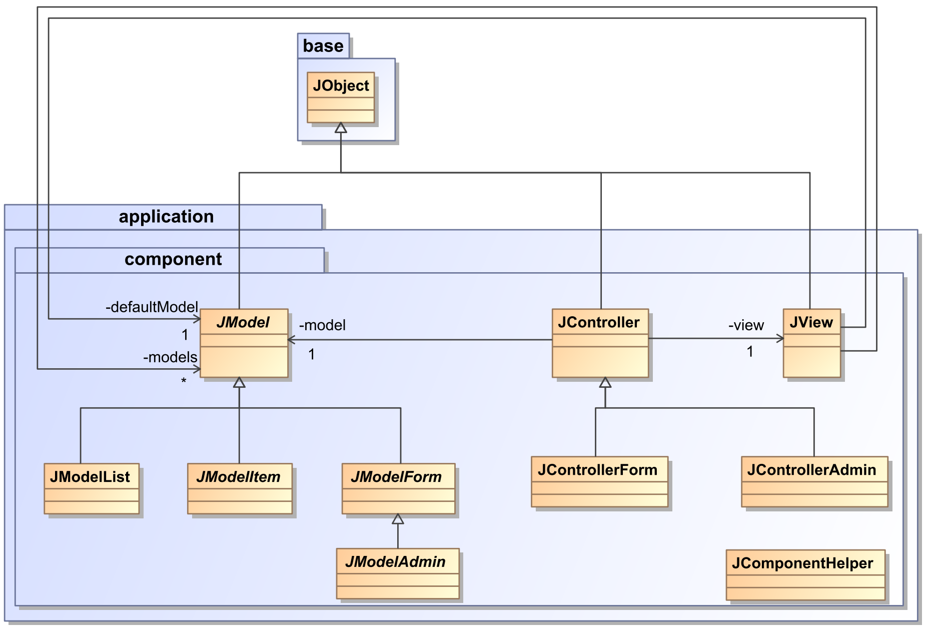


Figure 8.6: The architecture of Joomla! 2.5 MVC.

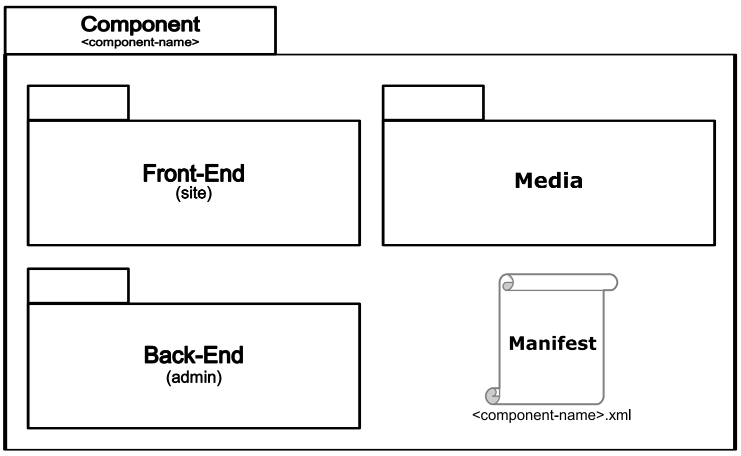


Figure 8.7: The organisation of Joomla! 2.5 component at the outer most level.

MVC has its own package, i.e. the models are put in the *models* package, controllers in *controllers* package and so fourth. Furthermore, the front-end package also contains two files, namely `controller.php` and `<component-name>.php`. The first file contains the main controller of front-end part. We mentioned this controller above in the context of Joomla! MVC. This controller will gain control from the Joomla! application whenever its component is accessed. The second file, `<component-name>.php`, is the *entry* point to the component from the Joomla! application. It is actually this file that the Joomla! application passes the control to whenever the `<component-name>` component is accessed. This file will in turn pass the control to the main controller. The following listing presents how this is normally done.

```
// import joomla controller library
jimport('joomla.application.component.controller');

// Get an instance of the main controller
$controller = JController::getInstance('<component-name>');

// Perform the Request task
$input = JFactory::getApplication()->input;
$controller->execute($input->getCmd('task'));

// Redirect if set by the controller
$controller->redirect();
```

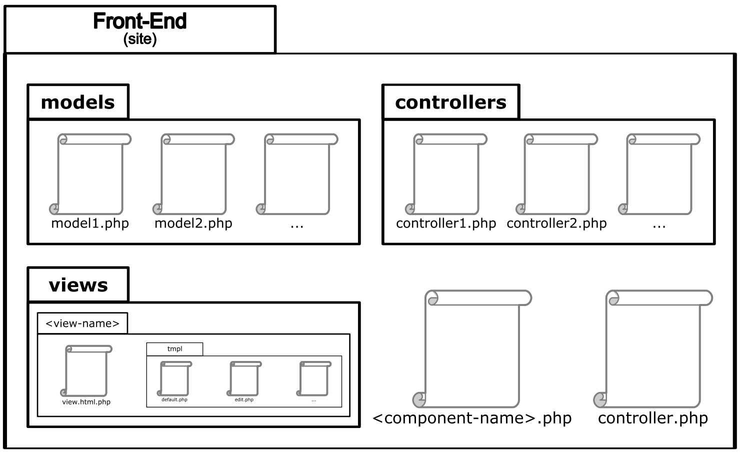


Figure 8.8: The organisation of Joomla! 2.5 component at the Front-End.

As mentioned above, in Joomla! MVC the view has been split into view and layout. This is shown in Fig. 8.8 on the facing page where the `views` package contains `view.html.php` and a package called `tmpl`. The `view.html.php` contains the implementation of a class which specialize the JView class from the Joomla! MVC framework. The `tmpl` package contains the different layouts for the view. The convention is that the default layout is named `default.php` and in order to access other layouts `layout=<layout-name>` is added to the URL requesting the view. Thus, in the default case the *layout* query can be omitted.

Back-End An overview of the content of the Back-End package is shown in Fig. 8.9. The MVC components in this package are equivalent to those in `Front-End` package except the `models` package. This package also contains a package called `forms` that contains the form for each model. These forms are defined using the XML notation provided by Joomla!. This is elaborated on in section 8.5 on page 117. The Joomla! Back-End part mainly consists of the so-called *content managers* which provide the means for adding, editing, deleting, listing etc. of data for administration purposes. For instance, in the case of Squad class in the data model in the running example (see Fig. 4.2 on page 28), a content manager will provide functionality such as listing available squads in the database, adding new ones or editing/deleting existing ones. Such content managers are created for all classes in the data model if the **Generate Content Manager** property is enabled in the Joomla! generator model. This property is elaborated on in the following.

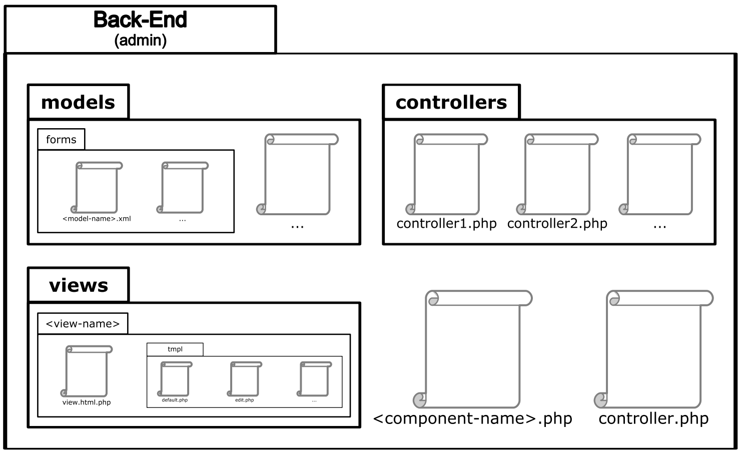


Figure 8.9: The organisation of Joomla! 2.5 component at the Back-End.

Media An overview of the content of the Media package is shown in Fig. 8.10. As seen in this figure, this package consists of three packages, `css`, `js`, and `images`. These packages contain the CSS (Cascading Style Sheets), JavaScript and images of the component, respectively.

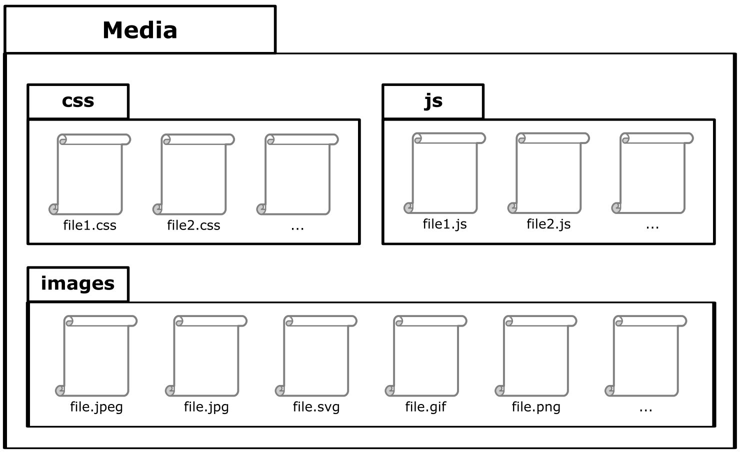


Figure 8.10: The organisation of Joomla! 2.5 component at the Media package.

Joomla! Class Naming Convention As we saw above, Joomla! enables developers to work with separate files for controllers, views and models. The file loading system in Joomla! takes care of the import of the right file in the right place. However, for this to work, certain naming conventions have been applied for naming the controllers, views and models. The name of each class, wether it is a controller, a model, or a view class must be prefixed with the name of the component and suffixed with the actual name of the controller, model, or view. For instance, the controller, model and view of the `Player` class from the data model in the running example would be `Vivoazzurro-ControllerPlayer`, `VivoazzurroModelPlayer`, `VivoazzurroViewPlayer`, respectively, where `Vivoazzurro` is the name of the component.

Customizing Joomla! Component Generation The Joomla! generator model provides various properties for customizing the generated Joomla! component. These properties are provided both on class and class feature (attribute and reference) level. These properties are specified in the following.

The **Generate Content Manager** property indicates whether to generate content manager for the administration purposes. The default value is *true* which means that content manager will be generated for the class. This property is a class level property.

The **Item MVC Name** property specifies the names of model, view, and controller that will be generated in order to construct the content manager for the class. This property specifies the case with single item, e.g. a player. This property is a class level property.

The **List MVC Name** property is equivalent to the Item MVC Name property, but specifies the name of the model, view and controller in the case of a list of items, e.g. list of players. This property is also a class level property.

By using the **Form Field Description** property it is possible to add description to the form field that corresponds to an attribute or a reference of a class. The value is used in the corresponding Joomla! content manager. This property is provided both for attributes and references of a class.

The **Form Field Label** property is the same as above one, but using this property it is possible to specify the label of the form field. This property is provided both for attributes and references of a class.

The **Form Field Type** property specifies the type of the form field that corresponds to an attribute of a class. The possible values are: `Text`, `Textarea`, `File`, `Password`, `Hidden`, `Radio` and `List`. This property is only provided for attributes of a class.

8.5.2 M2T Technologies

Before moving to the actual implementation of the code generator, we first discuss the choice of technology for Model-to-Text (or model-to-code) transformation. Next, we present the chosen technology in order to better understand the actual implementation later in the chapter.

We have investigated three technologies that all fulfill the basic criteria mentioned in section 5.3.2.1 on page 64. These technologies are *JET* (Java Emitter Templates) [Foub], *Acceleo* [Foua] and *Xpand* [Foud]. Another thing that these technologies have in common is that they are all template based. This reduces the complexity of the program that writes the code and increases readability of source code.

Using JET, the templates written are in a JSP-like syntax which is actually a subset of the JSP (Java Server Pages) syntax. This means that the templates are written in the Java programming language and in order to generate code using a meta-model (like webdsl.ecore and joomlagen.ecore) the java source code that is generated for the meta-model is used. This differs from Acceleo and Xpand which both use the meta-model directly when writing the templates, i.e. the abstract syntax is used directly in the templates instead of generated Java code of the abstract syntax. The JET syntax is easy to grasp for Java programmers but due to poor editors and IDE integration it is very error-prone. In the case of Acceleo and Xpand a new syntax must be learned, but due to support for a powerful IDE the learning curve is less steep.

Although, we started the development with JET (our first technology experiment was based on JET), we chose Acceleo as the M2T technology for the implementation of code generator. The reason behind this decision was mainly because Acceleo comes with a powerful IDE that makes the writing of source code very efficient and provides great features such as code-completion, refactoring, and better integration with the Eclipse IDE. Moreover, it supports merging of custom and generated code. Acceleo is also based on OMG standards, i.e. it is a pragmatic implementation of the OMG MOF Model to Text Language (MTL) [Foua]. These features make Acceleo advantageous compared to JET that due to lack of these feature is both error-prone and tedious to work with.

Xpand was as promising as Acceleo, but due to lack of contribution from the community maintaining it, it become less attractive. Acceleo has rapidly evolved and many features are still to come in the future. These features will make the technology even more powerful.

8.5.2.1 Acceleo in a Nutshell

In the following, we present the Acceleo technology. This presentation is very brief, therefor we refer to the Acceleo documentation for more details [Obe].

Our presentation will solely concern the Acceleo 3, which we have used in the implementation of the code generator. As mentioned above, Acceleo is the implementation of MOFM2T specification defined by OMG. The Acceleo language, which is named MTL by the OMG, is composed of two main kinds of structures, namely *templates* and *queries*. These structures are put inside a so-called *module*. Furthermore, the Acceleo language supports a subset of OCL (Object Constraint Language) [ISO12] for creating expressions in order to query the input models. In order to better understand these concepts, consider the following example.

```
[comment encoding = UTF-8 /]
[module Page2View('http://github.com/kanafghan/welipse/joomlagen/1.0',
                  'http://github.com/kanafghan/welipse/webdsl/1.0',
                  'http://www.eclipse.org/emf/2002/Ecore')]

[import ...::Expression2PHP /]
[import ...::common /]

[template public generateJView(page : Page, genModel : JoomlaGenModel)]

[comment @main/]
[file ('view.html.php', false, 'UTF-8')]
<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla view library
jimport('joomla.application.component.view');

/**
 * [page.name.toUpperFirst()/] View
 */
class [.../]View[.../] extends JView
{
    ...

    /**
     * [page.name.toUpperFirst()/] view display method
     * @return void
     */
    function display($tpl = null)
    {
    [if (page.parameters->size() > 0)]
        // Include other needed models
        JModel::addIncludePath(JPATH_COMPONENT.DS.'models');
    [/if]

    [for (param : Parameter | page.parameters)]
        $[param.var/] = ...;
        $this->assignRef('[param.var/]', $[param.var/]);
[/for]

        ...

parent::display($tpl);
}
```

```
    ...
}
[/file]
[/template]
```

The above example represents an excerpt of the transformation of the Page concept in the web model to the Joomla! view class.

As mentioned above, in Acceleo, templates and queries are put inside a module. In the above example, a module called `Page2View` is declared, which uses three different meta-models, namely meta-models of Joomla! generator model, web model, and Ecore model. The general syntax for declaring modules is:

```
[module <module-name>('meta-model-URI-1','meta-model-URI-2',...)]
```

Each module is contained within a single `.mtl` file. A module can extend another module, and thus override *public* and *protected* templates queries of that module. This is similar to object-oriented programming. A module can also *import* other modules in order to access and use the public templates and queries of these modules. In the example above, two modules have been imported, namely `Expression2PHP` and `common`. This is equivalent to import of packages in the Java programming language. The general syntax for importing modules is:

```
[import <qualified-name>::<module-name>/]
```

Notice that the OCL syntax for specifying qualified name is used.

The first line in the above example, i.e. `[comment encoding = UTF-8 /]`, is representing a comment in Acceleo. The general syntax for comments in Acceleo is:

```
[comment <actual-comment-text> /]
```

or in the case of a block of text as comment:

```
[comment] <actual-comment-text> [/comment]
```

In the above example, the `Page2View` module only contains a single template, namely `generateJView`. This template is a *public* template and has two parameters, namely `page` which is of type `Page` and `genModel` which is of type `JoomlaGenModel`. The general syntax for declaring templates is:

```
[template <visibility> <name>(<param1>,<param2>,...)]  ...[/template]
```

Templates can have *pre-conditions*, *post-treatments* and *variable initialization* in their declaration. The Acceleo language elements such as *file tags*, *for loops*, etc. are enclosed in the `[template]` and `[/template]`. Furthermore, the template can be annotated with `[comment @main/]` in order to indicate to Acceleo that a Java class must be generated to encapsulate the code required to run a generation. This is also the case in the above example.

There is no example of the query construct in the above example. However, as an example of this construct, consider:

```
[query public hasPageForm(page : Page) : Boolean
    = page.elements->exists(e | e.oclIsKindOf(Form)) /]
```

In this example, it is checked whether a given page contains the Form concept. Notice the OCL expression that performs this check. The general syntax of query is:

```
[query <visibility> <name>(<param1>,<param2>,...)  :  <return-type>
= <expression> /]
```

In the above example, an example of various Acceleo language elements is given. For instance, the file tag, i.e. `[file ('view.html.php', false, 'UTF-8')]`, represents the generation of a file called `view.html.php` with `UTF-8` encoding. The content of this file is inclosed in the `[file]` and `[/file]`.

The *template expression*, i.e. `[.../]`, generates text by querying the input model elements using OCL expressions. For instance, in the above example, `[.../]View[.../]` generates the view name according to the Joomla! class naming convention mentioned above. In this case, which is not presented in the above example, `View` is prefixed with template expression:

`[genModel.extensionName.toUpperFirst()]`

which generates the name of the Joomla! extension defined in the Joomla! generator model as capitalized text. The template expression for the suffix part is:

`[page.name.toUpperFirst()/]`

which generates the name of the page as capitalized text. For instance, if the name of the Joomla! extension is *vivoazzurro* and a page called *player*, the generated text will be `VivoazzurroViewPlayer`.

There is more to Acceleo which can not be covered here. For more information, we refer to the Acceleo documentation [Obe].

8.5.3 Code Generators

In the following, we present an overview of the various code generators. By code generators, we mean the Acceleo modules where each module generates a specific part of the resulting application. Next, we present the actual code generation flow that from a given Joomla! generator model generates a complete Joomla! component.

8.5.3.1 Acceleo Generators

There are 26 Acceleo modules that form the core part of the code generators for generating Joomla! component. Additionally, there are 3 common modules where one of these, namely `common.mtl`, contains all the common templates that are used by many templates in the core part. The other two modules, are the so-called *Java services wrappers*. These are special queries that invoke Java code from inside an Acceleo template. For instance, the `InitialData2SQLService.mtl` module contains one query which invokes the `generateInsertSQL()` method from the `InitialData2SQLGenerator` class which will generate the SQL corresponding to the provided initial data. This is elaborated on in section 8.6 on the facing page.

The core part of the code generator, i.e.the 26 modules mentioned above, mainly consists of templates that generate the various parts of the Joomla! component, e.g. models, views, controllers, etc. Some of the module files containing these templates are either named with the prefix `genFE` or `genBE` in order to distinguish between front-end related and back-end related modules. The files containing the 26 modules are named with a prefix `gen` except 3 of the modules. These 3 modules generate code for the web model concepts, i.e. Page, List, Expressions, etc. The Page concept is mapped to view in the Joomla! MVC. We have already seen the module that contains the template for generating the Page concept as view in the Joomla! MVC in section 8.5.2.1 on page 126. In the view, only the parameters of the page are initialized, while the actual generation of page and its elements as HTML is done in the `Page2HTML.mtl` module. In this module, the variables of the page are also initialized. The `Expression2PHP.mtl` module together with the `Page2HTML.mtl` and `Page2View.mtl` modules form the 3 modules that generate code for the web model concepts. In `Expression2PHP.mtl` module, as the name suggests, expressions are transformed to PHP codes.

All of the modules mentioned above are located in the `com.github.kanafghan.-welipse.joomlagen.generator` plugin. In the following, we present the way these modules (or code generators) are invoked.

8.5.3.2 Code Generation Flow

The generation of code initiates by the creation of the Joomla! component structure as presented in section 8.5.1.2 on page 120. This structure is contained within an Eclipse project which is named `com_<component-name>`, e.g. `com_vivoazzurro`. The implementation of this step is found in the `JComponentGenerator` class that is located in the `com.github.kanafghan.welipse.joomlagen.generator` plugin. At the root of the created project, three packages (as folders), namely `admin`, `media`, and `site`, together with the manifest file are created. This structure corresponds to the one shown in Fig. 8.7 on page 121. The three packages are also added their corresponding structure, e.g. to the `admin` and `site` packages, among others, the `controllers`, `models`, and `views` packages are added as shown in figures 8.9 on page 123 and 8.8 on page 122, respectively. Furthermore, in `JComponentGenerator` class any images and custom CSS files are copied to the corresponding package in the `media` package.

When the Joomla! component structure is generated, the rest of the work is delegated to the front-end and back-end generators. The implementation of these generators are found in the `JFEGenerator` and `JBEGenerator` classes in the same plugin as mentioned above. These generators, in turn, delegate the work to controllers, models, and views generator. At the lowest level, i.e. the generation of a controller, model or view, the corresponding Acceleo module is invoked in order to generate the actual code.

8.6 Initial Data Import

As mentioned in section 8.5.3.1 on the facing page, the generation of SQL for the provided initial data is found in the `generateInsertSQL()` method of class `InitialData2SQLGenerator` which is located in the `com.github.kanafghan.welipse.joomlagen.generator` plugin. In the following, we discuss the implementation of this generator.

The implemented algorithm that reads and generates SQL for the provided initial data can be summarized as: For each CSV file in the provided archive containing the initial data, match the file to a database table from the Joomla! generator model. A CSV file is matched to a database table if both have the same name (case insensitive) and the number of columns of the CSV file is the same as the number of columns and foreign keys of the database table. For each such match an `INSERT INTO` query is generated by parsing each line (or row) of the CSV file and generating the value part of the query. This value part corresponds to a record in the database table. The algorithm will output the generated query.

The above algorithm is very simple and not at all flexible. The user must ensure that each CSV file match exactly the corresponding table with regards to number of columns and foreign keys. Furthermore, the user must ensure that columns in the CSV file are separated with a semicolon (`;`) and each value is properly escaped with regards to special symbols, e.g. single quote (`'`) and semicolon (`;`).

A far better approach for parsing the CSV file would be to generate a parser using ANTLR. For instance, the example presented in section 8.3.3.2 on page 105 could be used. For the actual import of the initial data, it would be very usable to provide a user interface where columns of an imported CSV file is mapped to a specific database table column or foreign key.

The initial data import is an obvious feature where future works can contribute a lot. Our solution is a fast and very simple way of providing initial data of an application.

8.7 Welipse UI

So far, we have mentioned all the plugins, both in the core part and in the extensions part. However, one plugin, namely `com.github.kanafghan.welipse.ui`, has not been mentioned yet. The purpose of this plugin is to contain common user interface (UI) configurations and extensions. In the moment, this plugin only contains the definition of categories for commands (from the Eclipse command framework) and wizards.

Chapter 9

Testing Welipse

In this chapter, as the title suggests, we present the way we have tested the various parts of Welipse.

Testing is an essential activity in software engineering. Bertolino (2007) postulates, that although testing is a widespread validation approach in industry, it is still largely ad-hoc, expensive, and unpredictably effective [Ber07]. Testing software is a wide spectrum of different activities. To name a few, these activities range from unit testing (testing of small piece of code by the developer) to acceptance testing (customer validation of a large information system). Our approach of testing Welipse has mainly consisted of acceptance tests conducted ad-hoc in order to ensure that the requirements specified in section 5.3.3 on page 66 are met. Furthermore, for some parts of the tool that have been implemented manually, a kind of unit test has been conducted. This will be elaborated on in the following. Since a large part of Welipse has been generated using technologies such as EMF, GMF, ANTLR, and Acceleo, unit testing has been assumed irrelevant and instead the focus has been on the acceptance tests.

ID	Expression/Variable
VX01	Variable Initialization
VX02	Parameter
VX03	Integer constant
VX04	Real constant
VX05	String constant
VX06	Boolean constant
VX07	Arithmetic operation
VX08	Comparison operation
VX09	Boolean operation
VX10	String operation
VX11	List expression
VX12	Variable expression
VX13	Classifier operation
VX14	Structural expression
VX15	Web utility expression

Table 9.1: The various kinds of expressions and variables identified by an ID.

9.1 Unit Tests

We have conducted unit tests for testing the parser of the expressions language. The purpose of these unit tests were only to test the parser and the returned abstract syntax tree. These tests are organized in a number of test cases. These test cases are shown in table 9.2 on the next page. Here, each test case is identified by an ID (e.g. `TC01`). Furthermore, each test case has a short description explaining the purpose of the test case. Each test case has also an input which is either an expression or a declaration depending on whether it is an expression or a variable/parameter that is being tested. The *Expected* and *Actual* columns indicate the kind of expression/variable that was expected by the parser to parse and what was the actual result, respectively. The values of these columns are identifiers of the various expressions/variables which are presented in table 9.1. The *Result* column indicates whether the test case has been passed (PASSED) or (FAILED).

All of the test cases shown in table 9.2 on the next page are conducted automatically using a test program. The implementation of this program is located in the `com.github.kanafghan.welipse.webdsl.parsers` project in `Tester` class. This class is located in the `tests` package in the same project. The output of this program is as follows:

```
    'Test Case': <test-case-input> <'PASSED' | 'FAILED'> (<XV>)
```

ID	Description	Input	Expected	Actual	Result
TC01	Variable declaration with spaces	`name :  EString = player.name`	VX01	VX01	PASSED
TC02	Variable declaration without any space	`name:EString=player.name`	VX01	VX01	PASSED
TC03	Variable declaration with wrong syntax	`name:EString=`	Error	Error	PASSED
TC04	Variable declaration without initialization	`name:EString`	Error	Error	PASSED
TC05	Parameter declaration with spaces	`player:Player`	VX02	VX02	PASSED
TC06	Parameter declaration without any space	`player :  Player`	VX02	VX02	PASSED
TC07	An integer constant expression	`42`	VX03	VX03	PASSED
TC08	A decimal number expression in x.y format	`42.987`	VX04	VX04	PASSED
TC09	A decimal number expression in 0.y format	`0.42`	VX04	VX04	PASSED
TC11	A decimal number expression in x.0 format	`42.0`	VX04	VX04	PASSED
TC12	A decimal number expression in .y format	`.42`	VX04	VX04	PASSED
TC13	A decimal number expression in y. format	`42.`	VX04	VX04	PASSED
TC14	An empty string expression	“”	VX05	VX05	PASSED
TC15	A string constant expression	“Welipse”	VX05	VX05	PASSED
TC16	The boolean constant expression true	`true`	VX06	VX06	PASSED
TC17	The boolean constant expression false	`false`	VX06	VX06	PASSED
TC18	Use of a variable as an isolated expression	`player`	VX12	VX12	PASSED
TC19	Querying structural feature of an object	`player.name`	VX14	VX14	PASSED
TC20	Calling an operation of an object; with no arguments	`player.calculateAge()`	VX13	VX13	PASSED
TC21	Calling an operation of an object; with one argument	`player.getGoals(match)`	VX13	VX13	PASSED
TC22	Calling an operation of an object; with multiple arguments	`player.calculateAge(bY, bM, bD)`	VX13	VX13	PASSED
TC23	Calling an utility function within the framework	`WebUtils.getAllPlayers()`	VX15	VX15	PASSED
TC24	Expressing an empty list	`[]`	VX11	VX11	PASSED
TC25	Expressing a list with one element	[“Welipse”]	VX11	VX11	PASSED
TC26	Expressing a list with two elements	[“Male”, “Female”]	VX11	VX11	PASSED
TC27	Expressing an associative list with multiple elements	[1 => “One”, 2 => “Two”, 3 => “Three”, 4 => “Four”, 5 => “Five”]	VX11	VX11	PASSED
TC28	The string length operation	“Welipse”.length	VX10	VX10	PASSED
TC29	The string concatenation operation	“Hello”.concat(“ World!”)	VX10	VX10	PASSED
TC30	The string concatenation operation with other expressions than only string constants	`Name:  .concat(player.name)`	VX10	VX10	PASSED
TC31	Arithmetic expression with + and - operators	`1+2-5`	VX07	VX07	PASSED
TC32	Arithmetic expression with * and / operators	`2*3/4`	VX07	VX07	PASSED
TC33	Arithmetic expression with combination of all the operators	`2*3/4+5-(3+(-2))`	VX07	VX07	PASSED
TC34	Arithmetic expression with both numbers and variables as operands	`2*3/height+5-(weight+(-2))`	VX07	VX07	PASSED
TC35	Comparison expression with >, <, <=, >= operators	`3>2<6>=(5<=19)`	VX08	VX08	PASSED
TC36	Comparison expression with == and != operators	`(3==5)!=19`	VX08	VX08	PASSED
TC36	Comparison expression with combination of operators and operands of numbers and variables	`3>weight<6==(height!=190)`	VX08	VX08	PASSED
TC37	Boolean negation operation	`!(3>x)`	VX09	VX09	PASSED
TC38	Boolean conjunction operation	`(3>x) && (y==4)`	VX09	VX09	PASSED
TC39	Boolean disjunction operation	`(5>y) || (3==y)`	VX09	VX09	PASSED

TC40	Boolean operation with combination of operators and use of variables as operands	`!((5>y) || (3==y))`
		`&& ((player.weight>x)`
		`&& (y==player.height))`
			VX09	VX09	PASSED

Table 9.2: Test cases for testing expressions and variables. The IDs under **Expected** and **Actual** refer to table 9.1 on the preceding page.

where the `<XV>` represents the string representation of the expression or variable object.

For instance the output of the program in the case of test cases TC06 and TC07 would be:

```
Test Case: player : Player PASSED (...Parameter...)
Test Case: 42 PASSED (...IntegerConstant...)
```

9.2 Acceptance Tests

As mentioned above, we have conducted an acceptance test in order to determine if the functional requirements of the tool mentioned in section 5.3.3 on page 66 are met. This test is conducted by developing an example provided by the Peytz & Co.; vivoazzurro.it. In fact, this example is the one used in the running example (see chapter 2 on page 5).

In this example, all the features of Welipse have been used to some extend. These cover all the functional requirements specified in chapter 5 on page 61. Since the result is a runnable web application with the desired functionality, the test has been passed. However, this test is conducted by the developer of Welipse and not by the Peytz & Co or by a third-party. Moreover, this test only tests one possible configuration of the various features. Although, the result of this test indicate that the tool can be accepted, there might be issues and some part of the tool might malfunction.

Chapter 10

Reflections

In this chapter, we discuss the evaluation of the end result and present some requirements for platform specific extensions of the tool. Furthermore, we relate this thesis to other MDWE approaches, and discuss the similarities and differences in our approach compared to these approaches.

10.1 Related Work

As mentioned in the introduction, existing MDWE approaches already provide methodologies and tools for the design and development of most kinds of web applications, [FP00, CGP00, DPRC07, KKK07, CFP99], just to name a few. However, these proposals also come with some limitations. In [MRV08], an extensive overview of these proposals is presented and their limitations are outlined. Furthermore, it is discussed how these proposals have been addressing these limitations both in the context of MDA and using other approaches.

In order to relate the approach developed in this thesis to various other MDWE approaches, we distinguish between conceptual and non-conceptual approaches. The main difference between the conceptual and non-conceptual approaches is that the non-conceptual approaches provide a tool for supporting their approaches, while conceptual approaches do not do this. We will only related to the non-conceptual approaches. Among conceptual approaches, we have WEI [MRV08], WebML [MFV06, SWK06], OOHDM [SR98], RMM [FHV01], Webile [RMP04], WSDM [TL98], OOWS [PFPAa06], OO-Method [PGIP01], OO-H [GC03], MIDAS [CMS06] etc. We will not relate to these approaches as mentioned above. On the other hand, the approach in this thesis can be related to non-conceptual approaches such as *Torri* [CFP99], WebRatio [Web], W2000 [BCMM06], Hera [VFHB03], UWE [KKK07]. Especially, WebRatio based on the Interaction Flow Modeling Language (IFML) [Gro13] resembles our approach with regard to modeling flow (navigation) between pages and generating automatically code for these pages. At the time of writing this thesis IFML was adopted by OMG as a new OMG standard for integrating the front-end design into the system and enterprise models [Gro13].

10.2 Evaluation

In the previous chapter, we presented the way Welipse has been tested. We stated, that the acceptance tests, conducted so far, validate the tool. We also stated, that these tests include a small portion of the possible configuration of all the features of the tool. Taking this into account, we can not conclude confidently that the tool is ready to be used in production.

Developing only a single example (vivoazzurro.it– see chapter 2 on page 5), not even in full scale, does not indicate that the tool will be feasible in production. We are aware of this fact and in order to prove this hypothesis, a number of real-life examples, e.g. two examples, must be developed using the tool. Furthermore, these examples must be developed in a production house such as Peytz & Co in order for the evaluation to be valid. Due to the time-limitation of this thesis such an evaluation was not possible.

Although, we acknowledge the lack of proper evaluation of the tool and the notation developed in this thesis, we are confident that the tool is feasible in production and will boost the development once applied. Our confidence is based on the comments from Peytz & Co and our own figures, i.e.the time it take for us to develop an application using Welipse compared to developing it in traditional way.

10.3 Platform Specific Extension Requirements

In introduction to this thesis, we mentioned that one of the strengths of our tool is the possibility of extending it to multiple platform specific extensions (PSMs) and technologies such that from one PIM, multiple applications can be generated each targeting a specific platform. As we have seen so far, the default platform specific extension provided by us is targeting the Joomla! platform. In the following, we present some of the requirements that must be met by other platform specific extensions and assess the amount of work required to develop such extensions. To this end, we will use the Drupal platform as a case study.

Two requirements must be met by any platform specific extension. These are:

Generator Model (PSM): Each platform specific extension must have a generator model such as Joomla! generator model (see section 8.4 on page 113), e.g. Drupal generator model (`DrupalGenModel` in the case of Drupal platform). Such a model must contain the configuration of the generated code by wrapping the elements from web and data models. The Joomla! generator model is an obvious inspiration and much of its implementation can be reused in this case.

Code Generator: Each platform specific extension must also provide a code generator in a way similar to the code generator targeting the Joomla! platform (see section 8.5 on page 117). Also in this case, much inspiration can be taken from the code generator targeting the Joomla! platform.

It is a difficult task to assess the amount of work required for the development of a platform specific extension. However, we have tried to accommodate to the best of our knowledge. First thing to take into account is the architecture of the specific platform being targeted. If the platform provides a framework such as the one provided by the Joomla! platform (see section 8.5.1 on page 117), the implementation of the code generator becomes much easier. However, if this is not the case, there are possible solutions to make the implementation both less complex and easy. One such solution is to develop a framework as an interface between the platform and the code generator. In the case of choosing such solution the overall amount of work required for developing the extension will obviously be much greater. Considering the case with Joomla!, if we had to develop the extension from scratch, it would take 10-15 days. Bear in mind that in this case we both know Joomla! and the generator technology very well, not to mention that we have done it before.

Considering the Drupal platform, there are two possible approaches for the development of a platform specific extension that targets the Drupal platform.

Due to the architecture of the Drupal platform which differs a lot from the Joomla! platform, it is only the code generator that can be developed in two different ways. The Drupal platform supports a generic built-in data model which allows the definition of various kinds of content. To this end, a user interface is provided for the creation and management of content. For instance the different classes in the data model in the running example (see section 2.2 on page 9) can be mapped to the concept of *content type* in Drupal where each instance will then represent the concept of *node*. This is done manually through the provided interface. The way these nodes are displayed are controlled through the concept of *theme*.

One possible way to implement the code generator for the Drupal platform is the mapping from the data model of Welipse to the generic data model provided by Drupal. This can barely be called code generation, since the result of this generation would be a configuration that replaces the manual work required to define the data model within Drupal. It is also unclear to us whether this approach can be applied to any kind of web application. However, this approach would be very straightforward and require much less time to implement.

Another possible way to implement the code generator for the Drupal platform is by generation of Drupal *module*. This approach requires a great understanding of the Drupal platform and its application programming interface (API). Since the only way to extend the functionality of Drupal is through implementation of various *hooks*, it might not be possible to implement the code generator in a generic and extensible manner, which might require the implementation of a framework as a bridge between the code generator and the Drupal platform. Due to our limited knowledge of the Drupal platform, it is not possible to assess the precise amount of work this approach might need. However, if we had to develop this extension, it would take around 30 to 40 days.

Chapter 11

Conclusion

The development approaches involved in developing web applications remain ad-hoc and on a low level of abstraction, while web applications are becoming more sophisticated [KPRR06]. To this end, web Engineering, a new emerging discipline [ALA05], has been proposed through a number of approaches providing excellent methodologies and tools for the design and development of most kinds of web applications [MRV08]. However, these proposals also present some limitations such as lack of support for modeling architectural styles. One way to address these limitations is to adopt the Model-Driven Architecture (MDA) principles which is one of the best known Model-Driven Software Development (MDSD) initiatives.

In our approach, we have also adopted the MDA principles in order to provide a domain specific notation for modeling web applications on a higher level of abstraction. The resulting tool (*Welipse*) makes it possible to prototype rapidly and thus promptly capture the requirements of the application under development in an enhanced way. This, however, is only an assessment and need to be evaluated further. Due to time-limitation regarding this thesis, this evaluation was not possible.

11.1 Future Work

In this thesis, we have developed a DSL for modeling web applications on higher level of abstraction. It is possible to define the data of the application in the *data model* and, using this model together with the *web model*, define the pages of the application and the flow between them. Furthermore, the content of these pages are also defined in the web model. By using the *Joomla! generator model* which is initialized from the data and web models, it is possible to generate code targeting the Joomla! platform. The result is a runnable Joomla! 2.5 component which can be installed and run on a Joomla! 2.5 CMS.

The DSL developed for modeling web applications, in its present form, is limited and can not express many kinds of web applications. For instance, it is not possible to model user permissions and access control to pages. Future work might address this limitation by extending the notation.

The implementation of the tool in its present form do not fully comply with the MDA principles. Future work can improve on this by providing a neat implementation. For instance the *database generator model* must be separated from *Joomla! generator model* as mentioned in section 7.3 on page 95.

In order to demonstrate the extensibility of the tool and at the same time provide generation of web applications of different category, future work could extend the tool with a platform specific extension for developing web 2.0 applications.

Appendix A

User Guide

In the following, a detailed guide for how to install and use Welipse is presented. We begin by specifying the system requirements in order to install the tool. Next, the installation process is presented. Finally, a guide for each of the steps in the development process with Welipse, as presented in section 5.3.1 on page 63, is presented. Furthermore, these guides are related to each other such that following them in order must result in the process as depicted in Fig. 5.3.1. In addition, the application (*vivoazzurro*) in the running example is used as development task. The artifacts, e.g. data model, web model, and Joomla! generator model, of the application in the running example are available here:

`http://www.welipse.com/downloads/vivoazzurro.zip`

The generated code for this application, e.g. the deployable Joomla! 2.5 component, is also made available:

`http://www.welipse.com/downloads/com_vivoazzurro.zip`

A.1 System Requirements

Since Welipse is composed of a number of Eclipse plug-ins, an installed Eclipse version is required. To this end, the *Eclipse Modeling Tools*[1] package is recommended, since it fulfills the following requirements.

In order to use data models, the EMF (Eclipse Modeling Framework) is required. If the Eclipse Modeling Tools package is used, this requirement will be automatically fulfilled. For this user guide, the Eclipse *Kepler* (v4.3, 64-bit) as Eclipse Modeling Tools package with EMF v2.0.1 was used.

A.2 Installation

Welipse can be installed in a convenient manner through the update site provided below:

```
http://www.welipse.com/update
```

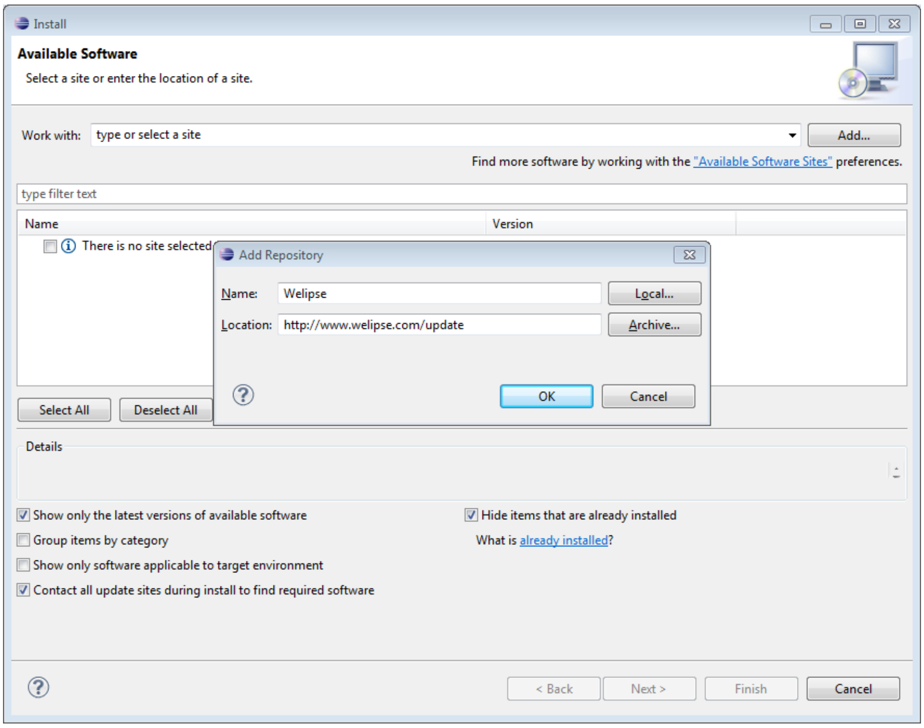
In order to accomplish this, choose `Help > Install New Software...` in Eclipse workbench. In the *Install* dialog, add the Welipse update site by clicking on the `Add` button. This is shown in Fig. A.1 on the facing page.

After completing the form as shown in Fig. A.1, click on `OK` button. Eclipse will fetch the plug-ins from this update site after a few seconds. The result should be a *Welipse* category containing two features: *Core* and *Joomla!*. Select both of these features, as shown in Fig. A.2 on page 146, and follow the installation wizard by clicking `Next`. If you encounter a security warning during the installation process, please ignore it.

A.3 Development Project

Any web application, developed with Welipse, will consist of a number of models and diagrams of these models. This results in many files that must be organized in one way or another. To this end, an Eclipse project can be used as *development project*. The creation of such a project is shown in Fig. A.3 on page 147.

[1]`http://www.eclipse.org/downloads`



Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ There is no site selected.	

Add Repository

Name: Welipse

Location: http://www.welipse.com/update

- [✓] Show only the latest versions of available software
- [ ] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software
- [✓] Hide items that are already installed

What is already installed?

Figure A.1: Adding the Welipse update site in Eclipse.

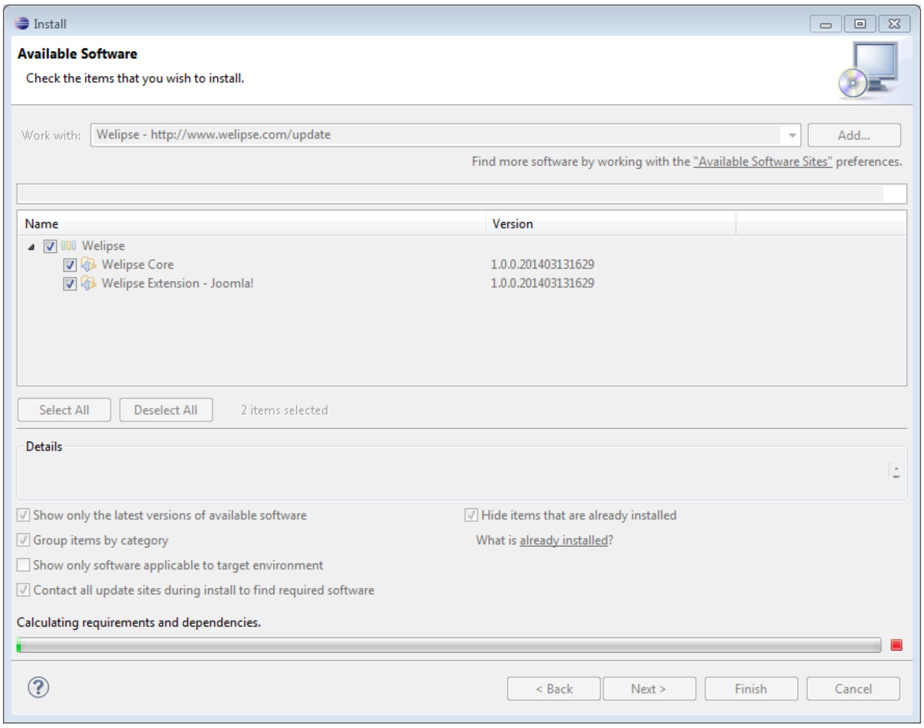
In order to open the *New Project* dialog shown in Fig. A.3a on page 147, choose `File > New > Project...` in Eclipse workbench.

Since we will be developing the *vivoazzurro* application in this user guide, we have created a project called `vivoazzurro` (see Fig. A.3b on page 147).

A.4 Data Model

The data model is created using an Ecore model. In order to edit the Ecore model, the Ecore Diagram can be used. In the following, we present how these are created. For more detailed information about these models, please refer to [SBPM09].

In order to create the data model, select the development project and choose


Install

Available Software

Check the items that you wish to install.

Work with: Welipse - http://www.welipse.com/update

Find more software by working with the "Available Software Sites" preferences.

Name	Version
☑ Welipse	
☑ Welipse Core	1.0.0.201403131629
☑ Welipse Extension - Joomla!	1.0.0.201403131629

2 items selected

Details

☑ Show only the latest versions of available software
☑ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

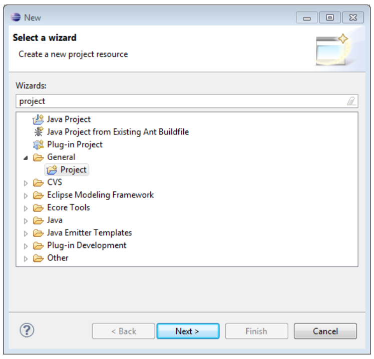
Calculating requirements and dependencies.

Figure A.2: Selection of the Welipse features that must be installed.

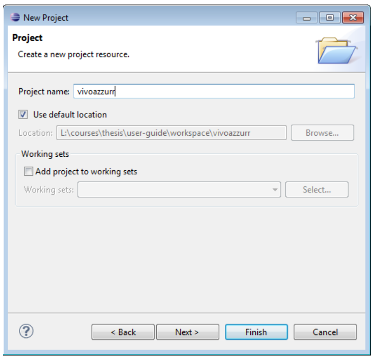
`File > New > Other...`. Alternatively, you can right-click on the development project and select `New > Other...`. In the dialog that opens, select the Ecore Diagram wizard as shown in Fig. A.4a on the next page. In this wizard, create the Ecore model by providing a name, e.g. `datamodel` as shown in Fig. A.4b on the facing page. By clicking the `Finish` button, the wizard will produce two files, e.g. `datamodel.ecore` and `datamodel.ecorediag`. This is shown in Fig. A.5 on page 148 under the *Package Explorer*. The wizard will also open the diagram file in the editor.

Using the Ecore diagram the data model of the vivoazzurro application (see 2.6 on page 11 in the running example) can be defined. A part of this model (the Player and Squad classes) has been shown in Fig. A.5 on page 148.

It is a good idea to validate the data model, once it is defined. In order to do this, open the Ecore model (the `.ecore` file) and right-click on the root element in the tree editor that has been opened and click on `Validate`. This step has been highlighted in Fig. A.6 on page 149 with the number 1. If the model is

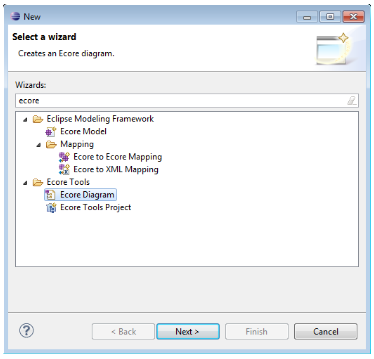


(a) Wizard Selection

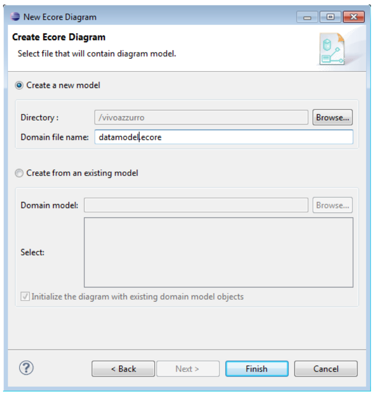


(b) New Project Wizard

Figure A.3: Creation of a development project in Eclipse.

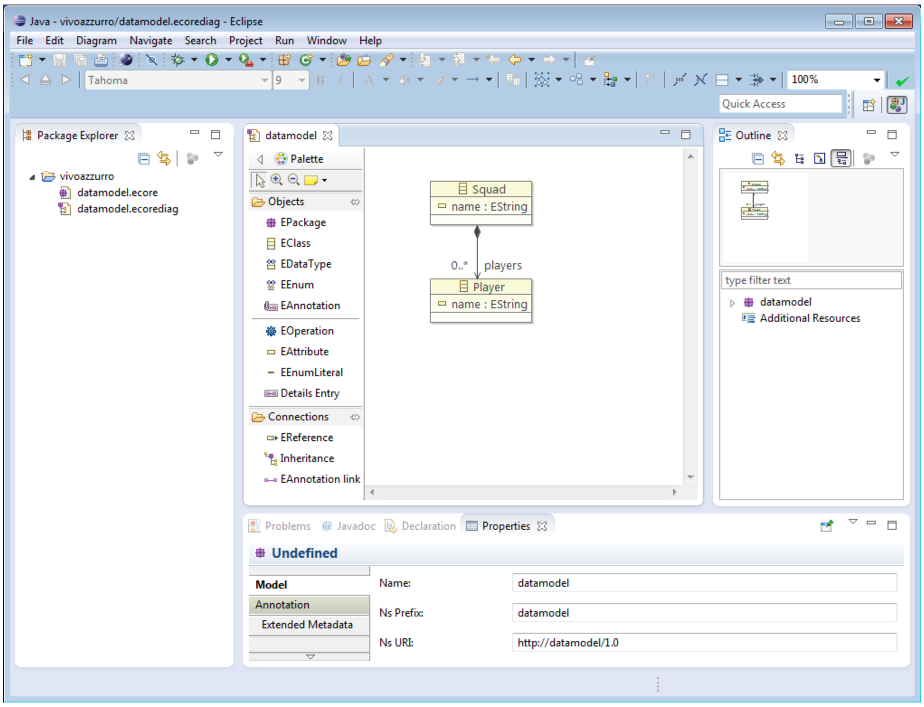


(a) Ecore Diagram Wizard Selection


New Ecore Diagram
Create Ecore Diagram
Select file that will contain diagram model.
Create a new model
Directory : /vivoazzurro
Browse...
Domain file name: datamodel.ecore
Create from an existing model
Domain model:
Browse...
Select:
Initialize the diagram with existing domain model objects
< Back
Next >
Finish
Cancel

(b) Ecore Model Creation

Figure A.4: Creation of data model as Ecore model in Eclipse.



Name: datamodel
Ns Prefix: datamodel
Ns URI: http://datamodel/1.0

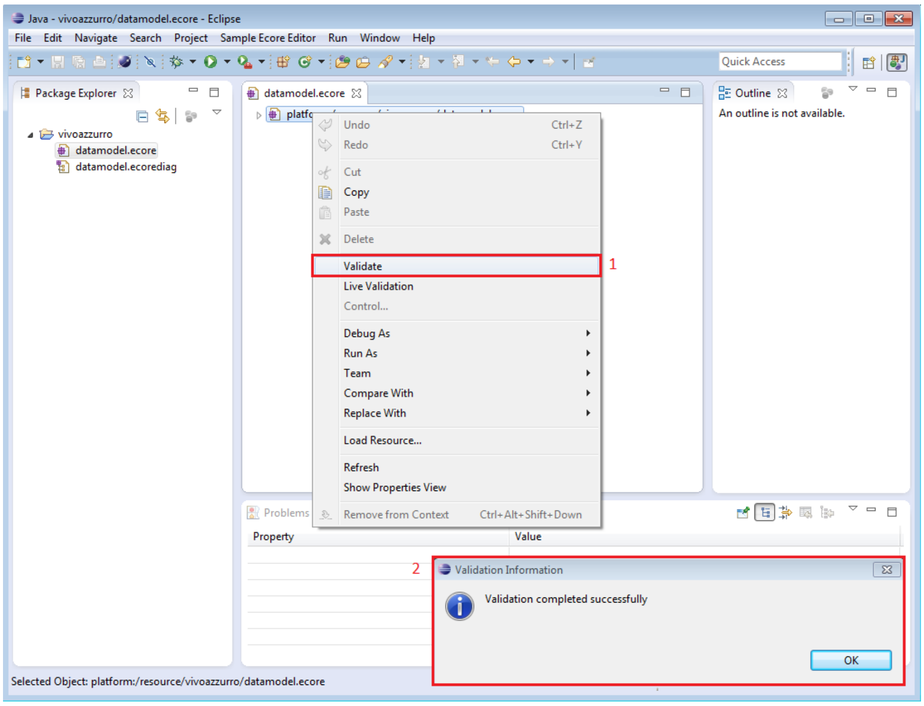
Figure A.5: Definition of data model using the Ecore diagram.

valid, you should get the message that has been highlighted with the number 2 in Fig. A.6.

A.5 Web Model

The second step in the development process is modeling the actual web application, the pages comprising the application, the navigation between these pages, and their contents. In the following, we begin by describing how web models are created in Welipse.

The creation of web models is somewhat similar to the creation of the data model presented above. To create a web model, right-click on the development project and select `New > Other...`. In the dialog, that opens, select the Web Model Diagram wizard as shown in Fig. A.7a on page 150. In this wizard, create the web model diagram (`.webdsldiag` file) by providing a name, e.g. `vivoazzurro`

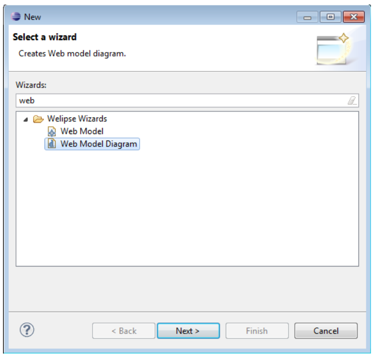


Validation completed successfully

Figure A.6: Validation of data model.

as shown in Fig. A.7b on the following page. Unlike the creation of data model, the creation of web model has yet another step. Click on the `Next` button in the wizard and provide the same name as for the diagram file, i.e. `vivoazzurro`, in order to create the *web domain model*. This is shown in Fig. A.7c on page 150. By clicking the `Finish` button, the wizard will produce two files, e.g. `vivo-azzurro.webdsl` and `vivoazzurro.webdsldiag`. This is highlighted in Fig. A.8 on page 151 with the number 1. The wizard will also open the diagram file in the graphical editor (see 2 in Fig. A.8).

In the following, we elaborate on the definition of the various concepts of web model in order to model a complete application. Furthermore, the integration of data model into web model and how it is used in web model is also elaborated on. Before doing so, we will give an overview of the graphical editor for working with web models. As seen in Fig. A.8 on page 151 (see 4 in this figure), this editor consists of a *palette* which contains the tools for creating the various elements of web models, e.g. Page, Parameter, Variable etc. Moreover, the editor consists of a white area (canvas) which will contain the various graphical


New

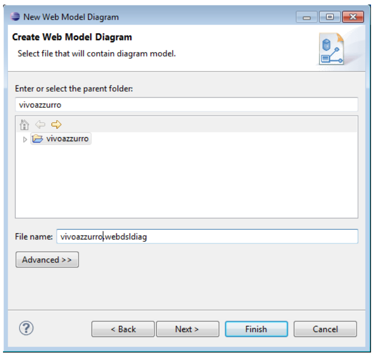
Select a wizard

Creates Web model diagram.

Wizards: web

- Welipse Wizards
  - Web Model
  - Web Model Diagram

(a) Web Model Diagram Wizard Selection


New Web Model Diagram

Create Web Model Diagram

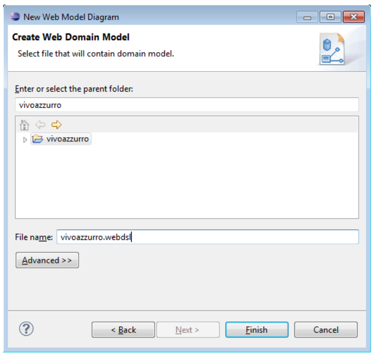
Select file that will contain diagram model.

Enter or select the parent folder: vivoazzurro

- vivoazzurro

File name: vivoazzurro.webdsldiag

(b) Web Model Diagram Creation


New Web Model Diagram

Create Web Domain Model

Select file that will contain domain model.

Enter or select the parent folder: vivoazzurro

- vivoazzurro

File name: vivoazzurro.webdsl

(c) Web Domain Model Creation

Figure A.7: Creation of web model in Eclipse.

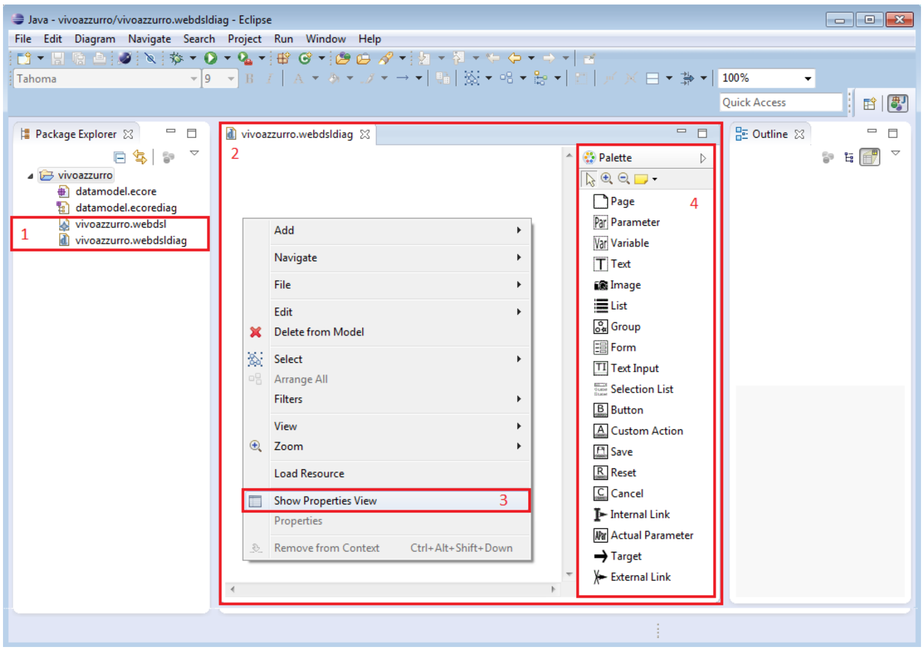
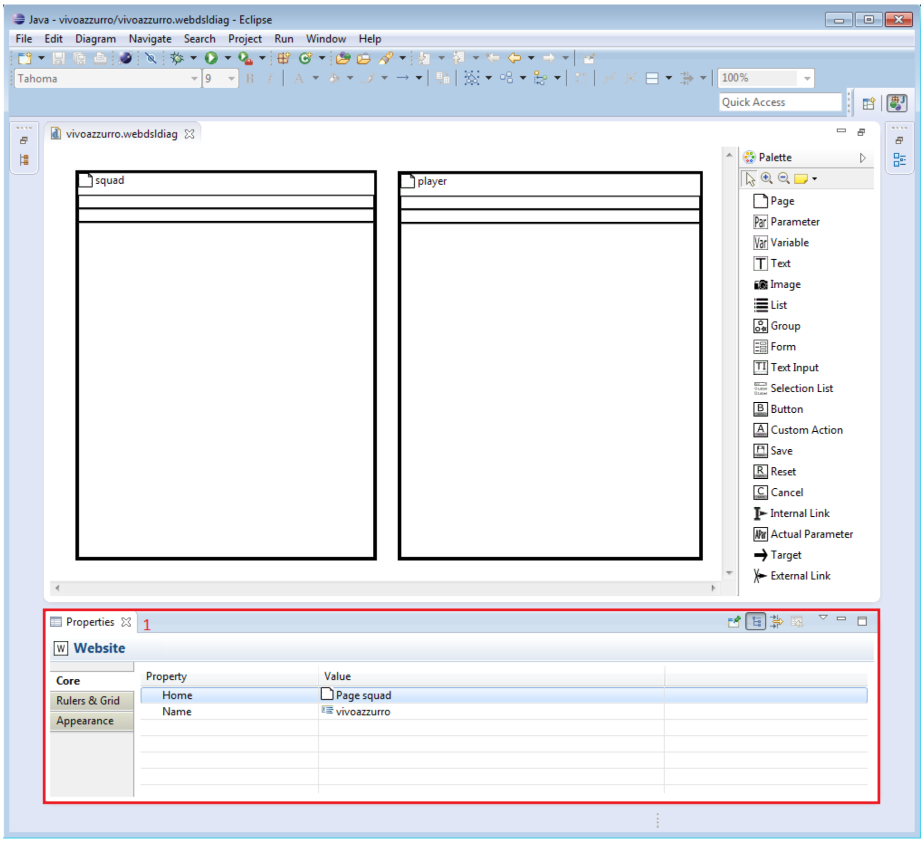


Figure A.8: The graphical editor for editing web models.

elements representing the concepts of web model.

A.5.1 Website Definition

The canvas mentioned above, represents the Website concept (see section 4.2.1.1 on page 36 for more information) which models the whole application. To begin with, we must set the name of the application by providing a value to the `name` property of the Website concept. This is accomplished in the *properties* view. If it is not already open, open it by right-clicking on the canvas and clicking on `Show Properties View`. This is highlighted in Fig. A.8 with the number 3. When the properties view is open, click on the canvas again in order to view its properties in the properties view. This is highlighted with the number 1 in Fig. A.9 on the next page. As seen in this figure, the name of the application is set to *vivoazzurro* and the `squad` page is selected as the *home* page. Furthermore, two empty pages, namely `squad` and `player`, has been defined for the application. These are elaborated on in the following section.



Property	Value
Home	Page squad
Name	vivoazzurro

Figure A.9: Definition of the Website concept.

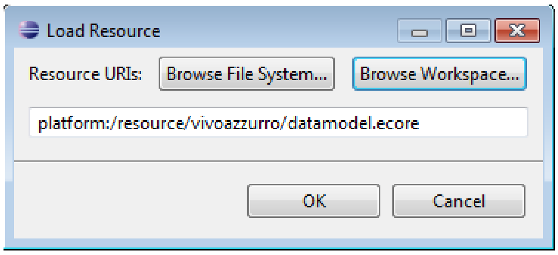
A.5.2 Page Definition

A page is added to the application by selecting the `Page` tool in the palette and clicking in the canvas. This will result in a rectangle that is divided into four compartments stacked vertically. The top most compartment contains the name of the page which can be provided as soon as the page is added to the application. The compartment below the name compartment is the parameters compartment. The variables of the page are added to the compartment below the parameters compartment, while the page elements are added to the fourth and biggest compartment; page elements compartment. Page elements are *Text*, *Image*, *List*, *Group*, *Form*, *Text Input*, *Selection List*, *Button*, *Custom Action*, *Save*, *Reset*, *Cancel*, *Internal Link*, and *External Link*. In other words, only tools with the same name from the palette are usable in the page elements compartment.

In the following, we elaborate on the definition of parameters, variables and elements of the page.

A.5.3 Definition of Parameters and Variables

The parameters and variables of a page can be defined using two different approaches. In the following, we present both approaches.



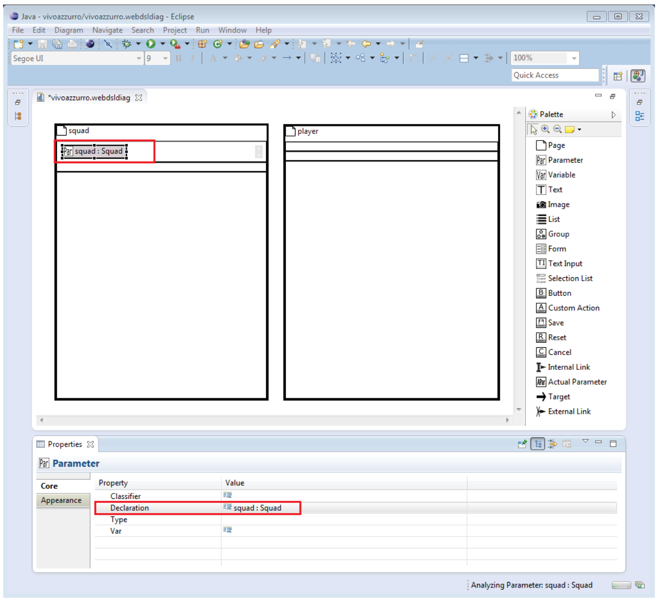
platform:/resource/vivoazzurro/datamodel.ecore

Figure A.10: Loading data model into web model.

In order to defined the parameters and variables of a page, the data model must be referenced by the web model. This is accomplished by right-clicking on the canvas and clicking `Load Resource...`. This will result in a dialog where the data model, which is the `.ecore` file, can be selected. This is done for the data model defined in previous section; see Fig. A.10.

NOTICE
In the following in the case of variable declaration and expression definition, whenever a problem is encountered during the parsing or analyzing process, an error dialog will be displayed. After correcting the expression or the declaration, the parsing and analyzing process will only take place if the newly added value in the property view is different than the previously added one. For instance, if `player.photo` is entered as expression and an error occurred due to the `photo` attribute of the `Player` class in the data model is not defined yet, the only way to parse and analyze this expression again is by removing it in the property view and hitting the Enter key, and then entering it again.

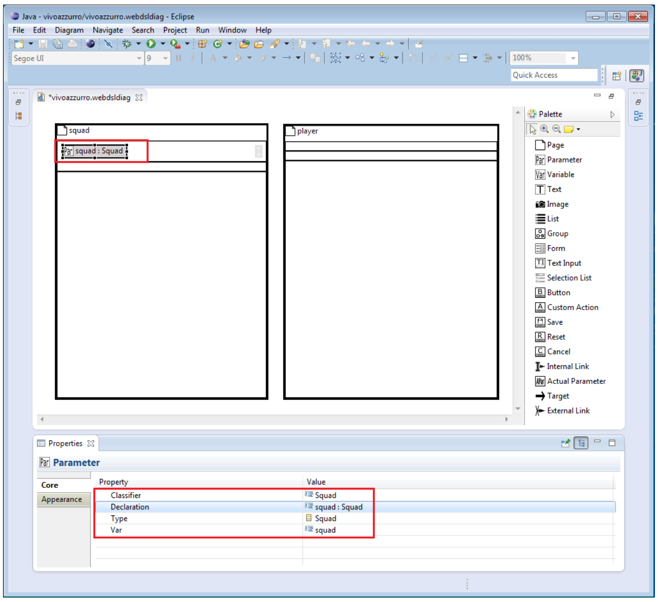
The normal approach for defining the parameters and variables of a page is using the corresponding tools in the palette. Select the Parameter or Variable tool



Property	Value
Classifier	
Declaration	squad : Squad
Type	
Var	

Analyzing Parameter: squad : Squad

(a) Declaration of **squad** parameter.

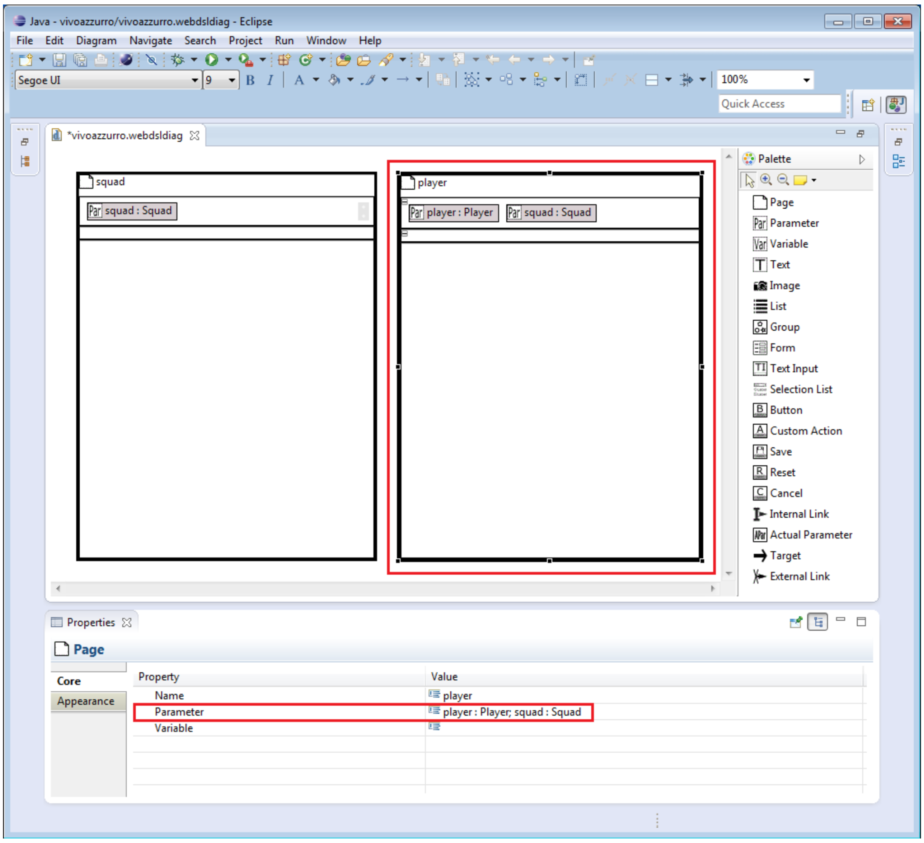


Property	Value
Classifier	Squad
Declaration	squad : Squad
Type	Squad
Var	squad

(b) The **squad** parameter successfully declared.

Figure A.11: An example of the normal approach for defining parameters and variables.

in the palette and click the corresponding compartment of the page where the parameter or variable should be added. This will add the graphical notation of the parameter or variable with a `<declaration>` label which is not editable on the diagram, but instead must be added through the properties view. Select the properties of the parameter or variable and provide its declaration by setting the `Declaration` property as shown in Fig. A.11a on the preceding page. If the declaration is correct, it will be parsed and the remaining properties will be automatically set to the proper values. This is shown in Fig. A.11b on the facing page.



Property	Value
Name	player
Parameter	player : Player; squad : Squad
Variable	

Figure A.12: The alternative approach for defining the parameters and variables of a page. In this case the parameters of the page are defined.

The alternative approach for defining the parameters and variables of a page is using the `Parameter` and `Variable` properties of the page, respectively. In

Fig. A.12 on the previous page, this is illustrated for the `player` page. In general, select a page and in the properties view using either the Parameter or Variable property add the declarations, e.g. `player:Player; squad:Squad`, by separating these with a semicolon (;). This will both create the graphical notation of the parameters and variables in the corresponding compartment and parse the declaration and set all the properties of the individual parameters and variables. This is a convenient way for adding multiple parameters or variables at once.

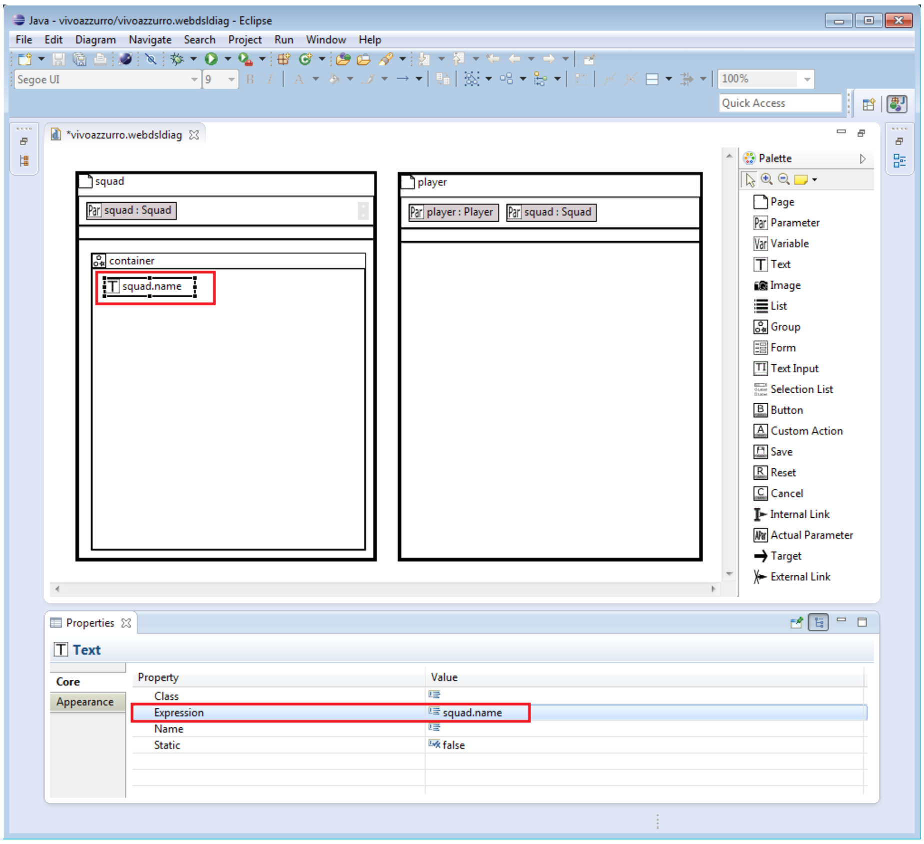
A.5.4 Definition of Page Elements

The page elements were mentioned above. These can only be added to the page element compartment of the page and other page elements such as the `Group`, `List` and `Form`. Most of the page elements can have one or more expressions. These expressions are added as text according to the syntax defined in section 4.2.2.2 on page 43. In the following, we elaborate on the page elements where, among others, their expressions are explained in more details.

A.5.4.1 Text

The definition of the Text element is straightforward; select the Text tool in the palette and click in the page elements compartment or any other element accepting the Text element. In Fig. A.13 on the next page, the Text element has been added to the Group element. In this figure, it is also shown how the expression of the Text element is defined. The expression of the Text element defines its content; a particular peace of text. In Fig. A.13, the expression expresses the name of a squad. The expression must be added through the properties view the same way the declaration of parameters and variables were added.

In the properties view in Fig. A.13 on the facing page, three more properties of the Text element are also shown. These are `Class`, `Name` and `Static`. The later one is a derived property based on the expression and must not be changed. The `Class` property is used to set the CSS class for styling the Text element. The `Name` property can be used to name this element in order to identify it easily. This property is used as the value of the *id* attribute of the generated HTML element that corresponds to the Text element.

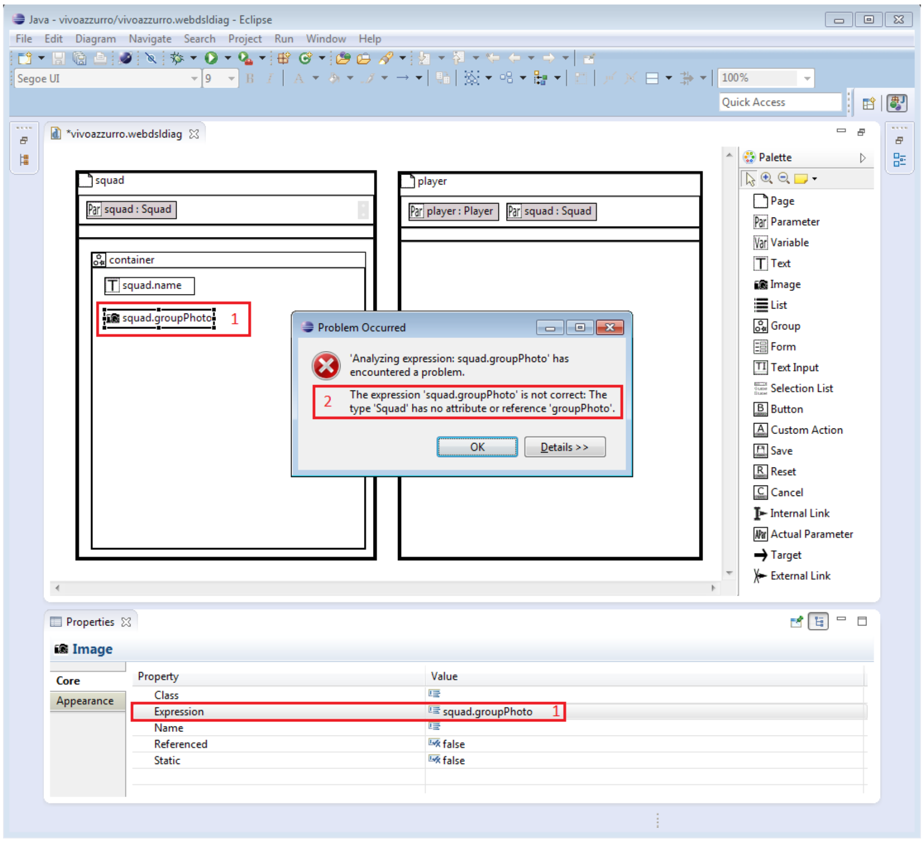


Property	Value
Class	
Expression	squad.name
Name	
Static	false

Figure A.13: Definition of Text element.

A.5.4.2 Image

The definition of the Image element is similar to that of the Text element. Like the Text element, the Image element also has an expression which expresses the actual image that will be displayed. In Fig. A.14 on the next page the definition of the Image expression is highlighted with the number 1. This expression, `squad.groupPhoto`, expresses the group photo of a squad. Since we have not defined the `groupPhoto` attribute of the `Squad` class in the data model, this expression is not correct and will result in an error. This is what we see in the error dialog in Fig. A.14. The part of this error dialog that has been highlighted with the number 2 is very important. It is here, that the main cause of the error will be presented. In order to parse and analyze the expression again after correcting the problem, the expression `squad.groupPhoto` must be removed in the properties view by clearing the input field and hitting the Enter key, and then entering the expression again. This was also mentioned in the notice above.



Problem Occurred

'Analyzing expression: squad.groupPhoto' has encountered a problem.

The expression 'squad.groupPhoto' is not correct: The type 'Squad' has no attribute or reference 'groupPhoto'.

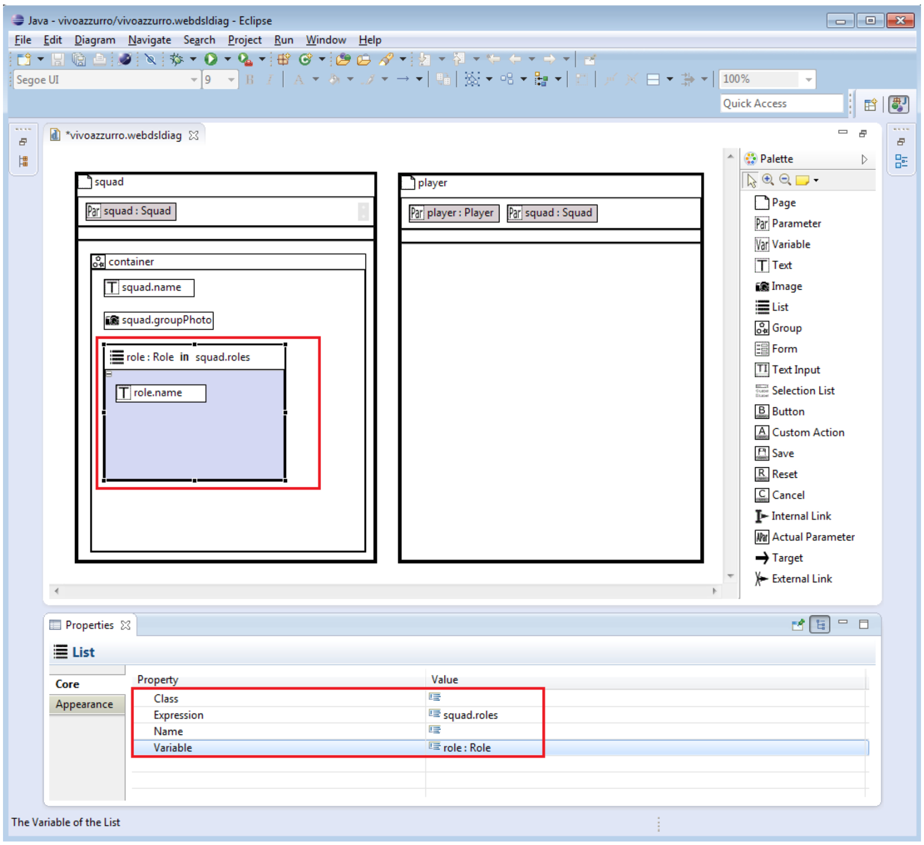
Property	Value
Class	
Expression	squad.groupPhoto
Name	
Referenced	false
Static	false

Figure A.14: Definition of Image element.

In the properties view in Fig. A.14, beside the three properties `Class`, `Name` and `Static`, which are the same as in the case of the Text element, a fourth property, namely `Referenced`, is also shown. If the actual image is provided by an URL the value of this property must be `true`.

A.5.4.3 List

This element is defined by using the List tool in the palette. In the properties view, the `Expression` and the `Variable` properties must be defined. This is shown in Fig. A.15, where the `Variable` property is defined as `role:Role` and the `Expression` property as `squad.roles`. The two other properties are similar to what was discussed above.



Property	Value
Class	
Expression	squad.roles
Name	
Variable	role : Role

Figure A.15: Definition of List element.

As mentioned before, the List element can also contain any page element. This is also shown in Fig. A.15, where a Text element is added to the list.

A.5.4.4 Group

This element is defined by using the Group tool in the palette. In the properties view only the `Name` and `Class` properties are applicable; just ignore the `Expression` property. Like the List element, the group element can contain any page element. In Fig. A.13 on page 157, an example of this element which contains the Text element is presented.

A.5.4.5 Form

This element is somewhat similar to the Group element. It is defined similarly and has the same properties as the Group element. In addition, this element has also the `Method` property with the default value `POST`; the other possible value is `GET`. Furthermore, the Form element can contain any other page element. However, only the *form elements* and elements such as Text, Image and Group makes sense as subelements of this element. Form elements are `Text Input`, `Selection List`, `Button`, `Custom Action`, `Save`, `Reset` and `Cancel` elements. These are elaborated on in the following.

An example of the Form element with some of the form elements is given in Fig. A.16 on page 167. In this figure, the Form element is highlighted with the number 1.

A.5.4.6 Text Input

This element is one of the form elements, but can also be added to a page or any other elements such as the Group element. The definition of this element is similar to any other page element, e.g. Text element. The Text Input element has two expressions, namely `Label Expression` and `Value Expression`, defining its *label* and *value*, respectively. These are defined similar to the expressions of other page elements presented above. Furthermore, the Text Input element has the `Class` and `Name` properties as any other page element. In addition, there are three properties specific to Text Input element, namely `Is Password`, `Is Text Area`, and `Required`. These can be used to configure the resulting Text Input as desired. The Text Input element and the corresponding tool in the palette are shown in Fig. A.16 on page 167 (see 2).

A.5.4.7 Selection List

This element is also one of the form elements, but can also be added to a page or any other elements such as the Group element. The definition of this element is similar to any other page element, e.g. Text element. The Selection List element has three expressions, namely `Label Expression`, `Value Expression`, and `Options Expression`. The latter one is specific to this element, while the other two are similar to that of the Text Input element. These expressions are defined similar to the expressions of other page elements presented above. Furthermore, the Selection List element has the `Class`, `Name`, and `Required` properties, where the later one is similar to that of the Text Input presented above, while the other two are similar to those of any other page element. In addition, there are two properties specific to this element, namely `Is Multiple` and `Rendering`. These can be used to configure the resulting Selection List as desired. The Selection List element and the corresponding tool in the palette are shown in Fig. A.16 on page 167 (see 3).

A.5.4.8 Button

This element is also one of the form elements, but can also be added to a page or any other elements such as the Group element. The definition of this element is similar to any other page element, e.g. Text element. The Button element does not have any expression. In addition to the `Class` and `Name` properties, this element has the `Type` and `Value` properties. These can be used to configure the resulting button as desired.

The Button element and the corresponding tool in the palette are shown in Fig. A.16 on page 167 (see 4).

A.5.4.9 Custom Action

This element is one of the form elements, but can also be added to a page or any other elements such as the Group element. However, since this element represents the submit action of the HTML form, adding it to any element except Form element will not make it function. The definition of this element is similar to any other page element, e.g. Text Input element. The Custom Action element has two expressions, namely `Performer Expression` and `Validator Expression`. These expressions are defined similar to the expressions of other page elements presented above. In addition to the `Class` and `Name` properties, this element has the `Type` and `Value` properties similar to the Button element specified above. The Type property, however, can not be changed, and will always have the Submit value.

The Custom Action element and the corresponding tool in the palette are shown in Fig. A.16 on page 167 (see 5). In this figure, the Performer Expression is defined as player.save, while the Validator Expression is defined as player.-validate.

A.5.4.10 Save, Reset and Cancel

These elements are also some of the form elements, but can also be added to a page or any other elements such as the Group element. However, since these elements represent the submit action of the HTML form, similar to the Custom Action above, adding them to any element except Form element will not make them function. These elements are defined in a similar manner as any other page element, e.g. Text Input element. The properties of these elements are equivalent of that of the Custom Action element.

The Save element and the corresponding tool in the palette are shown in Fig. A.16 on page 167 (see 6). In this figure, the Performer Expression is defined as player, while no expression is provided for the Validator Expression property.

NOTICE

Since only one submit button makes sense in a web form, only one of the Save, Reset, Cancel and Custom Action elements must be used. For a standard Reset button, use the Button element with the Type property set to Reset.

A.5.4.11 Internal Link

The Internal Link element is a *navigation element* among the page elements. This element is defined similar to any other page element; using the Internal Link tool. The properties of this element consists of Name and Target. Furthermore, the Internal Link element can contain the Text and Image elements and have a number of Actual Parameter elements. An example of Internal Link element is given in Fig. A.17 on page 168 (see 1). As seen in this figure, the Text element has been added to the to-player-page Internal Link element. Moreover, the Internal Link element contains two Actual Parameter elements, namely player and squad. These are defined using the Actual Parameter tool in the palette (see $*$ in Fig. A.17).

The Target property of the Internal Link element is defined by the blue arrow. For this, the Target tool in the palette is used. In Fig. A.17 on page 168, the blue arrow is pointing to the Text element player.name in the player page.

A.5.4.12 External Link

This element is also a navigation element among the page elements. The definition of this element is similar to any other page element; using the External Link tool. The External Link element has one expression which defines its target. This must be provided through the properties view by setting the Target Expression property. Like Internal Link element, the External Link element can contain Text and Image elements. An example of the Internal Link element is shown in Fig. A.17 on page 168 (see 2), where player.link is used as the target expression and a Text element ("Player Website") is used as its source.

A.5.5 Validation of Web Model

The web model consisting of four pages in the running example can be created using the above specification. Once the model is created, it is a good idea to validate it before moving on to the next step in the development. This can be accomplished in similar way that was described in the case of the data model above. In the case of web model, however, the .webdsl file must be used.

Open the .webdsl file, and right-click on the root element and select Validate. You must correct any error that is encountered during validation before moving on to the next step.

A.6 Joomla! Generator Model

The fourth step in the development process with Welipse is the refinement of the data and web models using a platform specific model. At the moment, the only available platform specific model is the Joomla! generator model. In the following, we present how this model is created.

In the development project right-click on the .webdsl file and select New > Other.... In the dialog, that opens, select the Joomla! Generator Model wizard as shown in Fig. A.18a on page 169. In the first page of this wizard, since we had selected the web model file, the name of the Joomla! generator model file will be derived as shown in Fig. A.18b on page 169. In the next and final page of the wizard the web model must be loaded. This is shown in Fig. A.18c on page 169. Again, since we had selected the web model file, it has already been selected. If it is not the case, locate it through the Browse File System... or Browse Workspace... actions. Click on the Load button and then click the Finish button. The wizard will produce a file, e.g. vivoazzurro.joomlagen, and open it in the tree editor.

As seen in Fig. A.19 on page 170, many of the Joomla! generator model properties have been initialized by deriving them from the web model. The Custom CSS Files and the Initial Data properties, however, will not have any value. Setting the Initial Data property is discussed in more details in section A.7.1.

The various properties of the Joomla! generator model and the wrapping data model (see 1 in Fig. A.19) can be specified in order to customize the generated code. The initial values of these properties, however, are reasonable for immediate code generation.

A.7 Code Generation

Once the Joomla! generator model (or any other platform specific model supported by Welipse) has been configured, the code for the application can be generated. In the case of Joomla! generator model, right-click on the root element, i.e. Model element, and select Generate Code. This is shown in Fig. A.20 on page 171. The result will be a new project, e.g. com_vivoazzurro, containing the complete code of a Joomla! 2.5 component. This is shown in Fig. A.21 on page 172.

A.7.1 Initial Data

The initial data of the application must be provided in a specific way in order to be interpreted and imported properly by the code generator. Although, the initial data must be provided during configuration of the Joomla! generator model and before the code generation, we recommend that the code generator should be run before defining the initial data. This is due to the fact that the generated SQL for creating the database tables is needed for defining the initial data properly, and in order to get this SQL the code generator must be run. The generated SQL is found in the `install.mysql.uft8.sql` file which is located under `admin/sql` folder in the generated project. This is shown in Fig. A.21 on page 172 (see 1).

For instance, consider the Player class in the data model in the running example (see Fig. 2.6 on page 11). This class has nine attributes and the corresponding database table will also contain these nine attributes as columns. However, it is not enough to create a CSV file with these nine attributes, since the generated database table has also some foreign keys that must be included in the initial data. In the following, the generated table for the Player class is presented.

```
CREATE TABLE IF NOT EXISTS #__vivoazzurro_Player (
    id int(11) NOT NULL AUTO_INCREMENT,
    name varchar(255),
    photo varchar(255),
    thumbnail varchar(255),
    biography varchar(255),
    link varchar(255),
    birthDate varchar(255),
    height varchar(255),
    weight varchar(255),
    gender int(11),
    Role_primaryPlayers int(11),
    Squad_players int(11),
    PRIMARY KEY (‘id‘)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

As we can see above, there is also the `Role_primaryPlayers` and `Squad-_players` columns that must be included in the initial data. These are foreign keys and correspond to the player's primary role and squad, respectively. The initial data for the above table must be defined in the following way:

```
name;photo;...;Role_primaryPlayers;Squad_players
Marco Parolo;parolo__xpx.jpg;...;1;1
Manuel Pasqual;Pasqualprofilo_.jpg;...;2;1
.
.
.
```

where, due to the lack of space, only the two first and the two last columns are shown. The order of the columns do not matter, whereas the names must match. The above excerpt of the players data is also presented as a table in the following.

`name`	`photo`	⋯	`Role_primaryPlayers`	`Squad_players`
Marco Parolo	parolo__xpx.jpg	⋯	1	1
Manuel Pasqual	Pasqualprofilo_.jpg	⋯	2	1
⋮	⋮	…	⋮	⋮

It is important that the columns in the CSV file are separated by a semicolon (;). Any images that are part of the data model, e.g. player photo and thumbnail, must be added to a folder named `img` and archived together with the CSV files representing the initial data of each database table. This archive must be created as a zip file and the absolute path to this file must provided as the value of the `Initial Data` property of the Joomla! generator model.

A.8 Deployment

The above generated code can be packaged into a zip archive and installed through the *Extension Manager* of a running Joomla! CMS (Content Management System). In order to package the generated code, right-click on the generated project and select `Export...`. In the dialog that opens, select the `Archive File` wizard as shown in Fig. A.22a on page 173. In this wizard select the location of the archive file on the local filesystem as shown in Fig. A.22b on page 173. By clicking the `Finish` button the archive file will be created. If you encounter a problem during this process, please refresh the generated code by either selecting the generated project and pressing the `F5` key or right-clicking on the generated project and selecting `Refresh`.

In the running example (see section 6.5.1 on page 82), it is shown how the installed application is run within the Joomla! CMS.

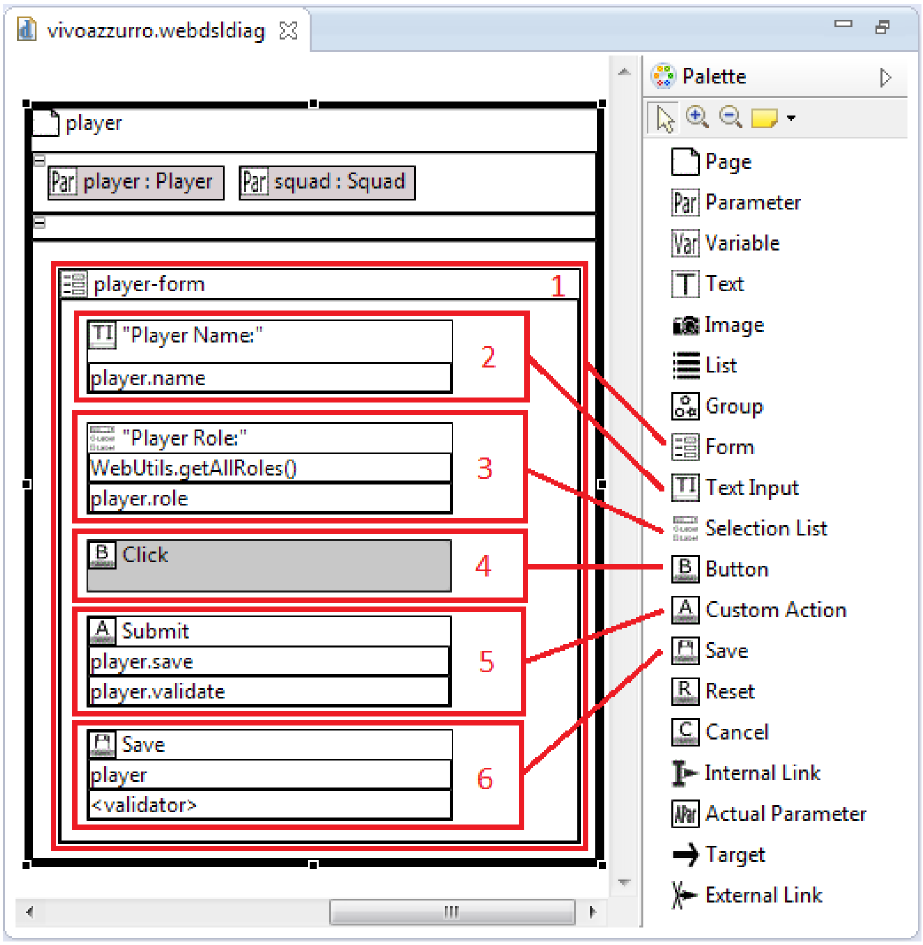


Figure A.16: Definition of Form element and its elements.

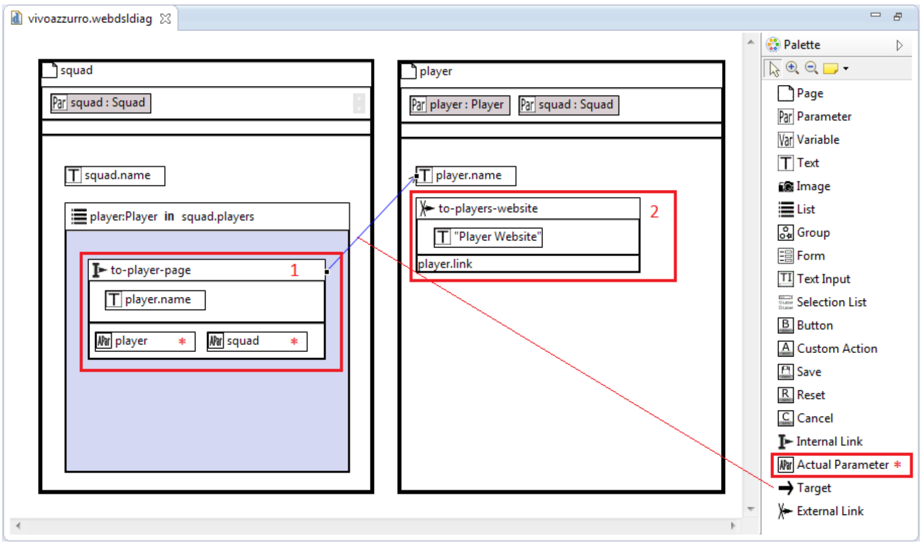
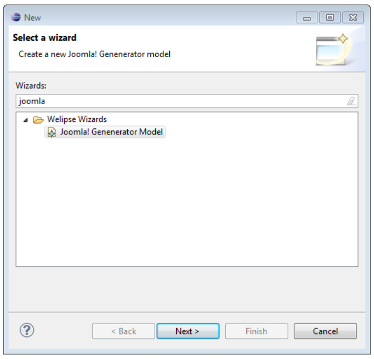


Figure A.17: Definition of the navigation elements of the Page element.

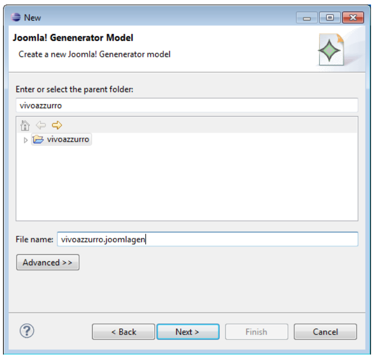


Select a wizard
Create a new Joomla! Genenerator model

Wizards: joomla

Welipse Wizards
Joomla! Genenerator Model

(a) Joomla! Generator Model Wizard Selection



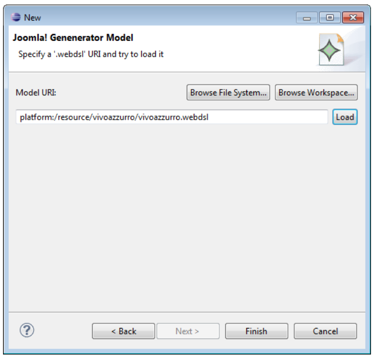
Joomla! Genenerator Model
Create a new Joomla! Genenerator model

Enter or select the parent folder: vivoazzurro

vivoazzurro

File name: vivoazzurro.joomlagen

(b) Joomla! Generator Model Creation

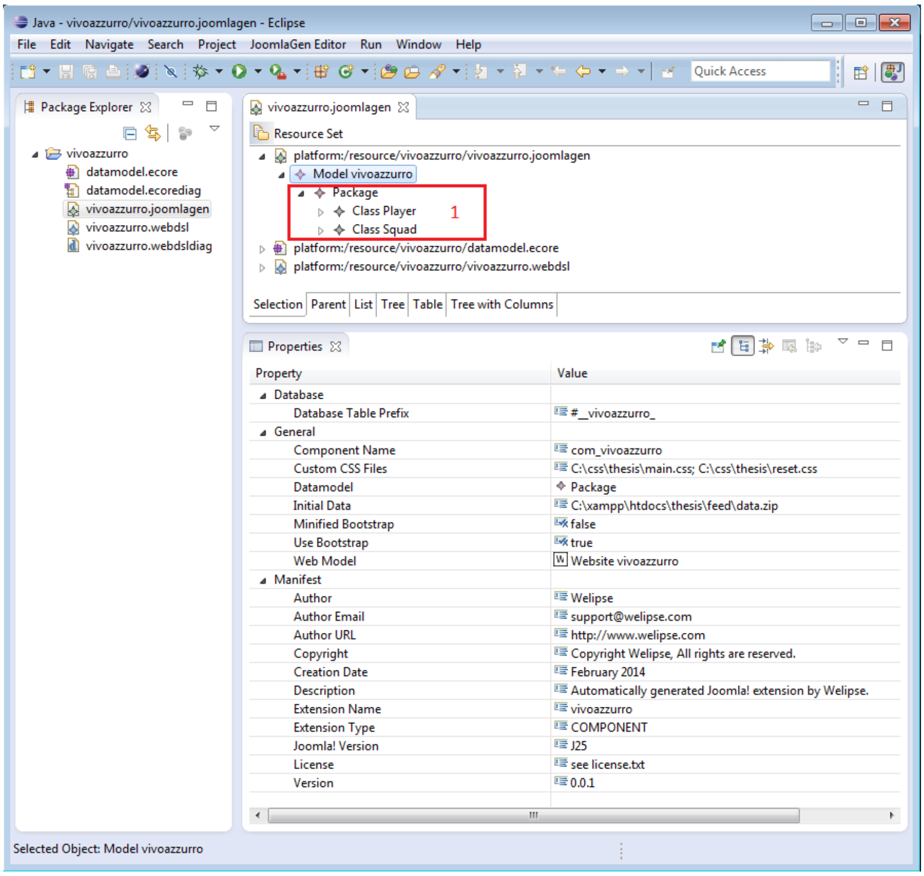


Joomla! Genenerator Model
Specify a '.webdsl' URI and try to load it

Model URI: platform:/resource/vivoazzurro/vivoazzurro.webdsl

(c) Initializing Joomla! generator model through web model.

Figure A.18: Creation of Joomla! generator model.


Java - vivoazzurro/vivoazzurro.joomlagen - Eclipse
File Edit Navigate Search Project JoomlaGen Editor Run Window Help
Quick Access
Package Explorer
vivoazzurro.joomlagen
Selection Parent List Tree Table Tree with Columns
Properties
1

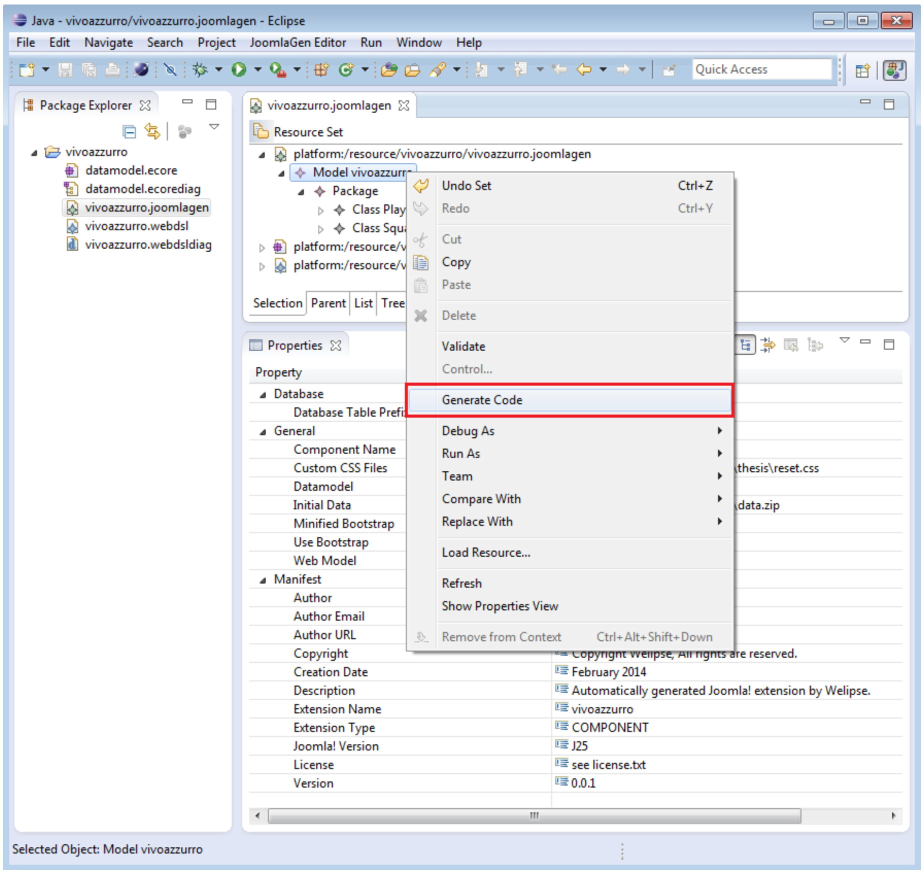
```
vivoazzurro
    datamodel.ecore
    datamodel.ecorediag
    vivoazzurro.joomlagen
    vivoazzurro.webdsl
    vivoazzurro.webdsldiag
```

```
Resource Set
  platform:/resource/vivoazzurro/vivoazzurro.joomlagen
    Model vivoazzurro
      Package
        Class Player
        Class Squad
  platform:/resource/vivoazzurro/datamodel.ecore
  platform:/resource/vivoazzurro/vivoazzurro.webdsl
```

Property	Value
Database	
Database Table Prefix	#_vivoazzurro_
General	
Component Name	com_vivoazzurro
Custom CSS Files	C:\css\thesis\main.css; C:\css\thesis\reset.css
Datamodel	Package
Initial Data	C:\xampp\htdocs\thesis\feed\data.zip
Minified Bootstrap	false
Use Bootstrap	true
Web Model	Website vivoazzurro
Manifest	
Author	Welipse
Author Email	support@welipse.com
Author URL	http://www.welipse.com
Copyright	Copyright Welipse, All rights are reserved.
Creation Date	February 2014
Description	Automatically generated Joomla! extension by Welipse.
Extension Name	vivoazzurro
Extension Type	COMPONENT
Joomla! Version	J25
License	see license.txt
Version	0.0.1

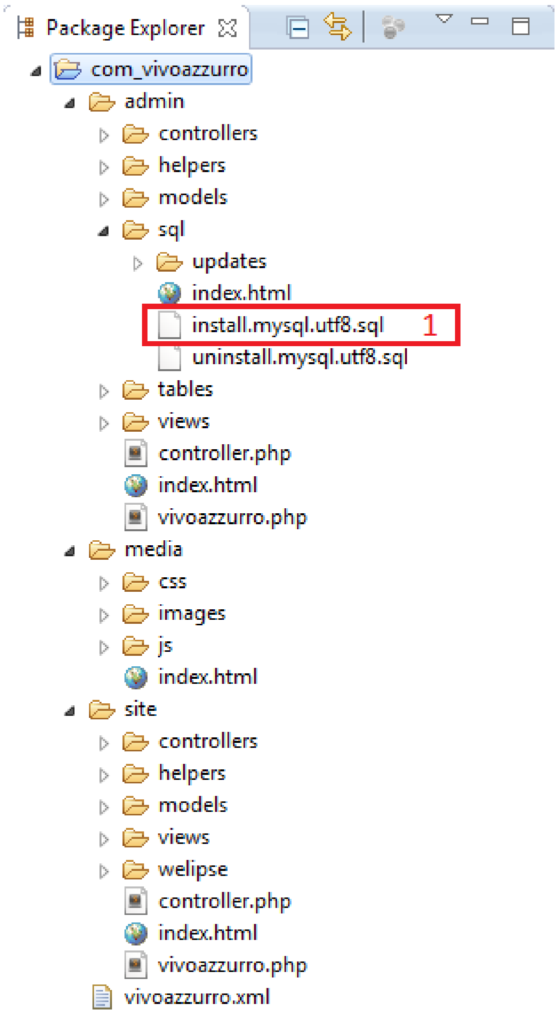
Selected Object: Model vivoazzurro

Figure A.19: Configuration of Joomla! generator model.



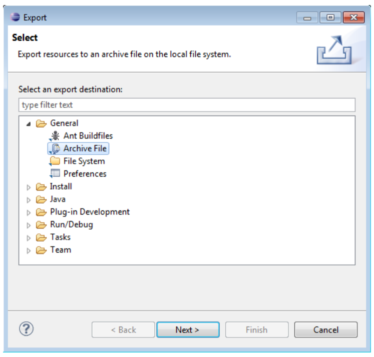
Property	
Database	
Database Table Prefi	
General	
Component Name	
Custom CSS Files	\thesis\reset.css
Datamodel	
Initial Data	\data.zip
Minified Bootstrap	
Use Bootstrap	
Web Model	
Manifest	
Author	
Author Email	
Author URL	
Copyright	Copyright Welipse, All rights are reserved.
Creation Date	February 2014
Description	Automatically generated Joomla! extension by Welipse.
Extension Name	vivoazzurro
Extension Type	COMPONENT
Joomla! Version	J25
License	see license.txt
Version	0.0.1

Figure A.20: Code generation from the Joomla! generator model.



```
Package Explorer
com_vivoazzurro
  admin
    controllers
    helpers
    models
    sql
      updates
      index.html
      install.mysql.utf8.sql   1
      uninstall.mysql.utf8.sql
    tables
    views
    controller.php
    index.html
    vivoazzurro.php
  media
    css
    images
    js
    index.html
  site
    controllers
    helpers
    models
    views
    welipse
    controller.php
    index.html
    vivoazzurro.php
  vivoazzurro.xml
```

Figure A.21: The generated code for the *vivoazzurro* application.


Export

Select

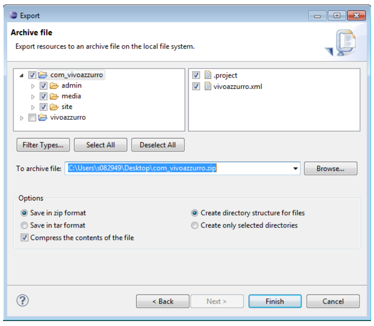
Export resources to an archive file on the local file system.

Select an export destination:

type filter text

- General
  - Ant Buildfiles
  - Archive File
  - File System
  - Preferences
- Install
- Java
- Plug-in Development
- Run/Debug
- Tasks
- Team

(a) Selection of Archive File wizard


Export

Archive file

Export resources to an archive file on the local file system.

- [✓] com_vivoazzurro
  - [✓] admin
  - [✓] media
  - [✓] site
- [ ] vivoazzurro

- [✓] .project
- [✓] vivoazzurro.xml

To archive file: C:\Users\s082949\Desktop\com_vivoazzurro.zip

Options

[● Save in zip format] ○ Save in tar format

[✓] Compress the contents of the file

[● Create directory structure for files] ○ Create only selected directories

(b) Archiving the generated code

Figure A.22: Packaging the generated code as zip archive.

Bibliography

[aia] Oracle and/or its affiliates. Javacc home. `https://javacc.java.net/`. Accessed: 15-11-2013.

[ALA05] Rashid Ahmad, Zhang Li, and Farooque Azam. Web engineering: A new emerging discipline. *Proceedings - IEEE 2005 International Conference on Emerging Technologies, ICET 2005*, 2005:445–450, 2005.

[BCMM06] Luciano Baresi, Sebastiano Colazzo, Luca Mainetti, and Sandro Morasca. W2000: A modelling notation for complex web applications. In Emilia Mendes and Nile Mosley, editors, *Web Engineering*, pages 335–364. Springer Berlin Heidelberg, 2006.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Buy] Dries Buytaert. Drupal - open source cms | drupal.org. `https://drupal.org`. Accessed: 06-02-2014.

[CFP99] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-driven, one-to-one web site generation for data-intensive applications. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 615–626, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[CGP00] Cristina Cachero, Jaime Gómez, and Oscar Pastor. Object-oriented conceptual modeling of web application interfaces: the OO-HMethod abstract presentation model. In Kurt Bauknecht, SanjayKumar Madria, and Günther Pernul, editors, *Electronic Commerce and Web Technologies*, volume 1875 of *Lecture Notes in Computer Science*, pages 206–215. Springer Berlin Heidelberg, 2000.

[CMS06] Valeria De Castro, Esperanza Marcos, and Marcos Lopez Sanz. A model driven method for service composition modelling: a case study. *Int. J. Web Eng. Technol.*, 2(4):335–353, July 2006.

[DPRC07] Damiano Distante, Paola Pedone, Gustavo Rossi, and Gerardo Canfora. Model-driven development of web applications with uwa, mvc and javaserver faces. In Luciano Baresi, Piero Fraternali, and Geert-Jan Houben, editors, *Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 457–472. Springer Berlin Heidelberg, 2007.

[FHV01] Flavius Frasincar, Geert-Jan Houben, and Richard Vdovjak. An RMM-based methodology for hypermedia presentation design. In Caplinskas and Eder [FHV01], pages 323–337.

[Foua] The Eclipse Foundation. Acceleo. `http://www.eclipse.org/acceleo/`. Accessed: 15-11-2013.

[Foub] The Eclipse Foundation. Eclipse modeling - m2t - home. `http://www.eclipse.org/modeling/m2t/?project=jet#jet`. Accessed: 15-11-2013.

[Fouc] The Eclipse Foundation. Gems home page. `http://www.eclipse.org/gmt/gems/`. Accessed: 13-01-2014.

[Foud] The Eclipse Foundation. Xpand - eclipsepedia. `http://wiki.eclipse.org/Xpand`. Accessed: 15-11-2013.

[Foue] The Eclipse Foundation. Xtext - language development made easy! `http://www.eclipse.org/Xtext/`. Accessed: 15-11-2013.

[FP00] Piero Fraternali and Paolo Paolini. Model-driven development of web applications: the AutoWeb system. *ACM Trans. Inf. Syst.*, 18(4):323–382, October 2000.

[GC03] Jaime Gómez and Cristina Cachero. OO-H method: Extending UML to model web interfaces. In Patrick van Bommel, editor, *Information Modeling for Internet Applications*, pages 144–173. IGI Global, Hershey, PA, USA, 2003.

[Gro03] Object Management Group. Common warehouse metamodel (CWM) specification. Specification Version 1.1, Volume 1, Object Management Group, March 2003.

[Gro13] Object Management Group. Interaction flow modeling language (IFML). Specification Beta 1, Object Management Group, March 2013.

[ISO96] ISO/IEC 14977:1996 information technology - syntactic metalanguage - extended bnf, 1996.

[ISO12] ISO/IEC 19507:2012(E) information technology - object management group - object constraint lanuage (OCL), 2012.

[Kel97] Wolfgang Keller. Mapping objects to tables. In *Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97). Siemens Technical Report 120/SW1/FB.* Citeseer, Siemens, 1997.

[Kin09] Ekkart Kindler. Model-based software engineering and process-aware information systems. In Kurt Jensen and WilM.P. Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *Lecture Notes in Computer Science*, pages 27–45. Springer Berlin Heidelberg, 2009.

[KKK07] Andreas Kraus, Alexander Knapp, and Nora Koch. Model-driven generation of web applications in uwe. *MDWE*, 261, 2007.

[KPRR06] Gerti Kappel, Birgit Pröll, Siegfried Reich, and Werner Retschitzegger, editors. *Web Engineering - The Discipline of Systematic Development of Web Applications.* John Wiley & Sons Ltd., England, 2006.

[Met] MetaCase. Metaedit+ domain-specific modeling tools. `http://www.metacase.com/products.html`. Accessed: 13-01-2014.

[MFV06] Nathalie Moreno, Piero Fraternalli, and Antonio Vallecillo. A UML 2.0 profile for WebML modeling. In *Workshop proceedings of the sixth international conference on Web engineering*, ICWE '06, New York, NY, USA, 2006. ACM.

[MM03] J. Miller and J. Mukerji. MDA guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MRV08] Nathalie Moreno, José Raúl Romero, and Antonio Vallecillo. An overview of model-driven web engineering and the mda. In Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 353–382. Springer London, 2008.

[Obe] Obeo. Acceleo 3.1.0 user guide - obeo network. `http://www.obeonetwork.com/group/acceleo/page/acceleo-3-1-0-user-guide`. Accessed: 27-02-2014.

[oE] ISIS / Vanderbilt University / School of Engineering. Gme: Generic modeling environment | institute for software integrated systems. `http://www.isis.vanderbilt.edu/projects/gme/`. Accessed: 13-01-2014.

[OMG] Inc. Object Management Group. Object management group. `http://www.omg.org`. Accessed: 03-03-2014.

[OMG06] OMG Object Management Group. Meta object facility (MOF) core specification. `http://www.omg.org/spec/MOF/2.0/PDF`, January 2006.

[OSM] Open Source Matters OSM. Joomla! the CMS trusted by millions for their websites. `http://www.joomla.org`. Accessed: 21-09-2013.

[Par] ANTLR / Terence Parr. Antlr. `http://www.antlr.org/`. Accessed: 15-11-2013.

[PF11] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In Hall and Padua [PF11], pages 425–436.

[PFPAa06] Oscar Pastor, Joan Fons, Vicente Pelechano, and Silvia Abrahão. Conceptual modelling of web applications: The OOWS approach. In Emilia Mendes and Nile Mosley, editors, *Web Engineering*, pages 277–302. Springer Berlin Heidelberg, 2006.

[PGIP01] Oscar Pastor, Jaime Gómez, Emilio Insfrán, and Vicente Pelechano. The OO-method approach for information systems modeling: From object-oriented conceptual modeling to automated programming. *Inf. Syst.*, 26(7):507–534, October 2001.

[RMP04] Davide Di Ruscio, Henry Muccini, and Alfonso Pierantonio. A data modelling approach to web application synthesis. *Int. J. Web Eng. Technol.*, 1(3):320–337, sep 2004.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, 2nd edition, 2009.

[SR98] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based applications design. *Theor. Pract. Object Syst.*, 4(4):207–225, October 1998.

[SWK06] Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. Bridging existing web modeling languages to model-driven engineering: A metamodel for WebML. In *Workshop Proceedings of the Sixth International Conference on Web Engineering*, 155, 2006. ACM International Conference Proceeding Series. eingeladen; Vortrag: 2nd Workshop on Model-Driven Web Engineering (MDWE 2006), in conjunction with ICWE 2006, Standford Linear Accelerator Center, Palo Alto, USA; 2006-07-11.

[TL98] O.M.F. De Troyer and C.J. Leune. WSDM: a user centered design method for web sites. *Computer Networks and ISDN Systems*, 30(1-7):85–94, 1998. Proceedings of the Seventh International World Wide Web Conference.

[VFHB03] R. Vdovjak, F. Frasincar, G. Houben, and P. Barna. Engineering semantic web information systems in hera. *JOURNAL OF WEB ENGINEERING*, 2(1/2):3–26, 2003.

[Web] WebRatio. High productivity web and mobile app dev platform | webratio. `http://www.webratio.com`. Accessed: 11-03-2014.