Enabling the Generation of Websites from Models

Mark Wood
School of Computer Science: MEng Software Engineering

Supervisor: Dr. Andy Carpenter

May 2009

Abstract

The development of both dynamic and data intensive websites requires the mastering of a number of different technologies and skills. This means that their development is restricted to a small number of specialist developers or teams that must be carefully managed. In reality, the development process involves sophisticated design skills and the substantial application of best practice patterns. This report describes a tool that uses a model of a data intensive website and code generation techniques, which encapsulate best practice patterns, to reduce the breadth of skills needed. By raising the level of abstraction at which development takes place, it allows designers to concentrate on the data managed by the website and its user interactions. Although the use of generation has numerous advantages, it can have the reputation of not giving users the ability to fully control what is produced. Thus, the tool described includes the ability for website developers to control generation choices and customise the final output.

Project Title:	Enabling the Generation of Websites from Models
Author:	Mark Wood
Supervisor:	Dr. Andy Carpenter
Date:	May 2009

Contents

Table of Figures

Chapter 1 Introduction

1.1 Websites and their Architectures

Driven by the rise of the Internet and Society's seemingly insatiable demand for information, the number of websites continues to grow at an ever increasing rate. In March 2009, Internet services company Netcraft reported that the total number of websites across all domains stood at 226 million [1]. However despite these figures and the increasing complexity of websites, the way in which websites are developed has remained largely unchanged for many years.

1.1.1 Website Architectures

As the use of websites became more widespread, the architecture and sophistication of their underlying systems changed accordingly. In this section, a brief analysis of three of the most widely used architectures is presented.

1.1.1.1 Static Websites

Static websites are those that are composed of pages that have static content. Static sites are most typically used for domains where the data within the site rarely changes. This architecture is rarely used in modern systems due to improvements in technology and the increasingly complex demands of users.

1.1.1.2 Dynamic Websites

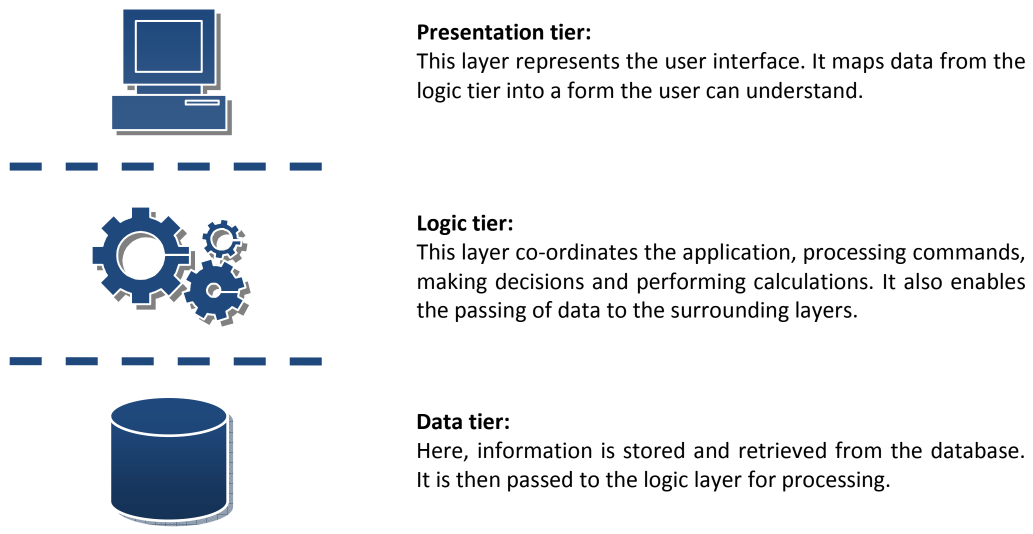
Dynamic websites are those that use a server side scripting language to retrieve content from a backend database prior to a page being displayed. Pages are structured using a mixture of static HTML integrated with a dynamic language, such as PHP. When the page is requested from the server, the server side language is executed and dynamic content is encoded into the page. This architecture is still widely used for many simple websites.

1.1.1.3 Data Intensive Websites (Three-Tier / N-Tier)

Data intensive websites use a more complicated architecture than the previous two. They are similar to dynamic websites in that they also make use of persistent data. However, the architecture of a data intensive website is more robust and as a result tends to be used for more sophisticated applications such as e-commerce.

The architecture for this class of website uses a number of separate levels, the minimum being three. This layered approach provides a separation of concerns allowing each layer to be modified without affecting any of the others. The different levels that make up three-tier architectures are shown in Figure 1. The amount of data used in this class of website is also typically far greater than that of a dynamic website. Indeed, huge websites such as eBay and Amazon use vastly more complicated architectures even than this; typically incorporating more tiers and wider distribution of processing.

A common characteristic of both data intensive and dynamic websites is that they support the creation and maintenance of data through the website. This is often known as CRUD functionality. CRUD stands for Create-Read-Update-Delete; it embodies all of the operations that can be performed on persistent data.



Presentation tier:
This layer represents the user interface. It maps data from the logic tier into a form the user can understand.

Logic tier:
This layer co-ordinates the application, processing commands, making decisions and performing calculations. It also enables the passing of data to the surrounding layers.

Data tier:
Here, information is stored and retrieved from the database. It is then passed to the logic layer for processing.

FIGURE 1: THREE-TIER ARCHITECTURE

1.1.2 Building Websites

Building a website from scratch takes time, typically anywhere between a few days to many months. The exact duration will depend on a number of factors, the main ones being the size of the website and the functionality it requires. In the past there were two main ways of developing a website; manually writing all of the source code by hand or using a software tool to assist in producing some of the source code automatically.

Tools aiding website development have existed for some time. Microsoft FrontPage [2] and Adobe Dreamweaver [3] are notable examples. These tools provide the developer with a WYSIWYG interface for creating pages whilst maintaining and generating the code behind them automatically. Although these tools can be useful for designing complex layouts, the code they produce has been criticised for being extremely verbose and in some cases going against best practices e.g. heavy use of inline CSS styles [4].

Experienced web developers tend to prefer hand crafting websites for many reasons: cleaner code, better search engine integration and greater control [4]. There are, however, obvious downsides to this approach. Clearly, there is a pre-requisite that the developer is fluent in at least the core web languages, HTML, CSS and JavaScript. Mastering just one of these languages in isolation is difficult and so to be able to use all three competently is less than straightforward. In addition, hand crafting code is often more time consuming compared to using one of the aforementioned software tools.

Regardless of which method is used, the developer is still ultimately responsible for producing at least a proportion of the website manually; usually its functionality.

1.2 Website Generation

There is now however another option. Technology exists that can generate complete websites directly from a specification / model. This allows developers to produce websites that are fully functional, in a matter of minutes without having to write any code by hand. Although this technology is relatively young, website generators do exist [5] that can produce quite sophisticated systems. The premise behind the technology is known as Model-Driven Software Development (MDSD) and will be covered in more detail in section 1.3.

1.2.1 Existing Website Generators

There are a number of existing website generation tools available [5], offering varying degrees of sophistication in the systems they generate. Two of the most well established are AndroMDA and WebML, both of which have existed for around 5 years. This section presents an analysis of both of these systems and looks at a new generator that is currently under development.

1.2.1.1 AndroMDA

AndroMDA (pronounced "Andromeda") is an extensible generator framework for transforming models defined in UML into deployable websites [6]. The tool comes with a number of "cartridges" (c.f. plug-ins) that allow the developer to choose the generated output technology. AndroMDA also provides a framework for developers to write their own cartridges enabling virtually any website technology to be generated. Although AndroMDA is open source, the platform as a whole is built from bespoke components. It is this point that proves to be AndroMDA's main drawback; not building upon standard tooling such as the Eclipse Modeling Framework means the usage of AndroMDA is likely to be less wide spread. This point is analysed in more detail in section 2.3.2.

1.2.1.2 Web Modeling Language (WebML)

WebML [7] is slightly different to AndroMDA. Instead of using UML, WebML defines its own graphical syntax for specifying the structure and functionality of websites. WebML uses a CASE tool known as WebRatio to support both the definition and the generation of websites. Although WebML seems slightly more sophisticated in the structure of sites it can produce, it only supports generation of one specific technology [8]. Unlike AndroMDA it is not extensible and therefore binds the developer to a particular implementation technology. Furthermore, as with AndroMDA, WebML is also built from bespoke components and so suffers the same problems that this brings.

1.2.1.3 Texo (Under Development)

As has been mentioned, the two previous generators are both built on bespoke frameworks. Texo is a tool being built on the Eclipse Modeling Framework (EMF). EMF is quickly becoming the de facto standard for model-driven tooling and so this system is potentially more extensible in the future. Texo has not been officially released yet so it is difficult to truly analyse its capabilities. All of the information currently available on Texo is found on its Project Proposal page [9]. Texo uses EMF's own modelling notation, Ecore, as the basis for the website model. The information provided indicates Texo will initially only be able to generate data intensive websites in a single implementation technology. It also suggests that the website model in Texo will use Ecore directly rather than defining more appropriate, website specific concepts.

1.3 Model-Driven Software Development (MDSD)

The process of generating implementations from models is not just restricted to websites. In fact, it is possible to generate source code for any purpose from any domain specific model. This process is known as Model-Driven Software Development (MDSD) [10] and is a growing field within Software Engineering.

1.3.1 Modelling in Software Development

The use of models in software development is not a new concept. Before looking at how modelling is currently used, it is useful to first have a clear understanding of what a model is.

A model is an abstract representation of a software related artefact. Although a model is an abstraction it is not inaccurate. It contains only the necessary detail such that it is sufficient for its purpose. Figure 2 illustrates a very simple model of a University system drawn in UML. The model shows the concepts within the domain and their attributes and relationships.

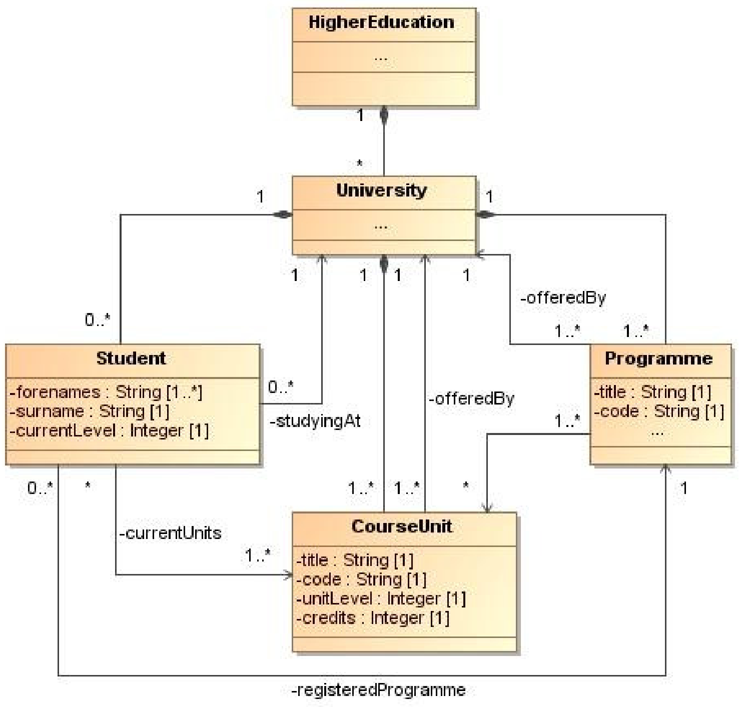


FIGURE 2: MODEL OF A UNIVERSITY SYSTEM

In the software engineering world, modelling has a rich tradition, dating back to the earliest days of programming. Modelling is typically performed during the analysis and design stages of development and is primarily used for gaining a better understanding of the domain. However, with MDSD, models are centric to all development to the extent whereby, from the developer's point of view, the model *is* the code.

Generating implementations with MDSD brings with it a number of advantages over hand crafting code. By raising the level of abstraction, generation can be used to automatically encode best practice patterns into the systems it produces. This allows developers to concentrate on the more unique features of the system. Furthermore, not only does generation vastly reduce the amount of time it takes to build systems, it also produces robust, error free, efficient code. Generation therefore significantly decreases the cost of software development and maintenance. As the systems we produce continue to grow in complexity, the need to rise to higher levels of abstraction is likely to increase. Indeed, this trend has already been exhibited in programming languages; the use of binary was replaced by assembler code, assembler code was replaced by low level languages such as C, which have now largely been replaced by higher level languages such as Java.

1.3.2 Model Transformation Basics

The process of generating an implementation from a model can be described as a series of transformations.

The model driven life cycle begins by defining a model at a high level of abstraction that is independent of any implementation detail. This is known as a Platform Independent Model (PIM). The PIM describes the concepts within the domain.

In the next step, the PIM is transformed into one or more Platform Specific Models (PSMs). A PSM is a model that contains information linked to a specific platform or intended implementation technology. For example, a PSM of a relational database is likely to include concepts such as "table", "index" and "foreign key". Typically, a separate PSM is produced for each implementation technology. As most modern systems are built from a range of technologies, it is common to have many PSMs associated with one PIM. The process of transforming from a PIM into a PSM is known as a Model-to-Model (M2M) transformation.

The final step in the process is to transform the PSMs into textual artefacts. In this context, a textual artefact may be program code, configuration files, XML, or any other textual syntax. The process of transforming a PSM into textual artefacts is known as a Model-to-Text (M2T) transformation.

To illustrate this flow a simple case study, Rosa's Breakfasts, is used. It is taken from *MDA Explained The Model Driven Architecture: Practice and Promise* [11].

Rosa sells home delivered, customisable breakfast packages from a website. There are different menus e.g. Full-English and Continental, and different styles e.g. Simple, Grand and Deluxe. It is decided that the system will be a standard web based three-tier application. The application will consist of a database, a middle tier using Enterprise Java Beans (EJB), and a user interface built with Java Server Pages (JSP). The flow from PIM to textual artifacts is shown in Figure 3.

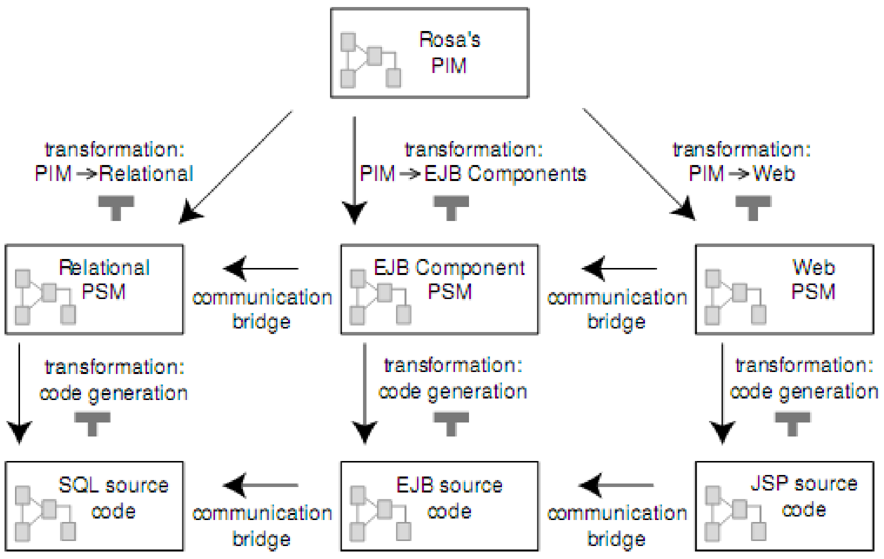

Rosa's PIM
transformation: PIM → Relational
transformation: PIM → EJB Components
transformation: PIM → Web
Relational PSM
communication bridge
EJB Component PSM
communication bridge
Web PSM
transformation: code generation
transformation: code generation
transformation: code generation
SQL source code
communication bridge
EJB source code
communication bridge
JSP source code

FIGURE 3: A TYPICAL MODEL-DRIVEN FLOW [11]

Notice how the model-driven process works on three different levels of abstraction:

- At the highest level of abstraction we define the PIM. This model is independent of any specific implementation detail.
- At the next level are the PSMs. These models abstract away low level details such as coding patterns but are still platform specific.
- At the lowest level we have the generated textual artifacts. These are also models, which by definition, are entirely platform specific.

This process is explored in further detail in Chapter 2.

1.4 Project Rationale

From the information presented so far the rationale for the project should be clear. Websites are an important part of modern life and are applicable for a wide range of uses. Existing methods of developing websites remain time consuming and require at least some core knowledge of web based languages. Technology is available that allows the systematic transformation from a model of a domain into an implementation. Existing website generators are built on bespoke frameworks that are not easily extensible. Tools are being built that use standardised frameworks, however there is no reliable indication as to when they will be completed. There is therefore a clear demand for website generators built on standardised tooling. This is what the project aims to fulfil.

1.5 Project Requirements

The tooling must meet a number of requirements. This section explores what these requirements are and why they are important.

1.5.1 Data Intensive Websites

The tool should be targeted specifically at generating data intensive websites. It should enable the user to define full CRUD functionality on any of the persistent elements in the model. The website the tool generates should be deployable immediately, requiring no code to be added or changed by hand. The styling of the generated website should be clear, attractive and easy to follow.

1.5.2 Graphical Model Editor

A graphical editor is to be produced to allow the definition of website models in a clear and intuitive manner. Graphical editors are typically easier to use than a textual syntax. Generation of websites directly from the model produced by the editor should require minimal user interaction. The editor should incorporate some form of validation to ensure the generated site is error free.

1.5.3 Website Customisation

Ideally, the tool should enable websites to be customized in some way before generation takes place. Some examples of potential customisation are different styles and layout options, different ways of defining similar functionality, different ways to structure pages or support for multiple languages & internationalization. A key requirement is that the website developer should not be constrained when making design choices; flexibility is vital.

1.5.4 Transparency between Generated vs. Hand Written

The fact the websites are generated should not have a detrimental impact on the final implementation technology or architecture of the generated website. The generated implementation should be as good as a hand crafted version of the same site.

Chapter 2 Background Reading

Thus far a rather simplistic view of modelling and model-driven systems has been presented. In reality, there are a number of subtle concepts that are important to understand before the tool itself can be discussed. In this section, a more formalised view of MDSD is presented and an overview of some of the technologies used in the project is given.

2.1 Model-Driven Architecture

In 2001, the Object Management Group (OMG) defined a software design approach known as Model-Driven Architecture (MDA) [12]. MDA is a standard approach to developing software systems using models [10]. MDA is slightly restrictive as it tends to focus solely on the use of UML-based modelling languages. Although MDSD follows many of the key principles of MDA, it is important to realise they are not the same thing. This section focuses on formalising many of the key principles common to MDA and MDSD so the reader can gain a better understanding of how modelling works.

2.1.1 The Domain

The starting point of all MDSD is a domain. This term describes a bounded field of interest or knowledge. All MDSD is domain specific. Attempting to model multiple domains in one model is likely to introduce overwhelming complexity and in practice is infeasible.

2.1.2 Metamodeling

In order to define a model and subsequently generate an implementation from it, we must first have a formalised structure and understanding of the domain. A formalised model is required that describes the possible structure of models in an abstract way. It should define the concepts within a model and their relationships as well as any constraints or rules that apply. This model is known as a metamodel. We say that a metamodel defines the abstract syntax and static semantics of a model.

Figure 4 below shows a subset of the UML metamodel [13] that can be used to describe the model in Figure 2. Classes represent the main entities in the UML diagram i.e. Student, Course Unit, Programme, University and Higher Education. Classes contain zero or more Attributes. Attributes have a type described by a Classifier that can be either a PrimitiveDataType or a Class. Associations are relationships between two Classes; a source and a target.

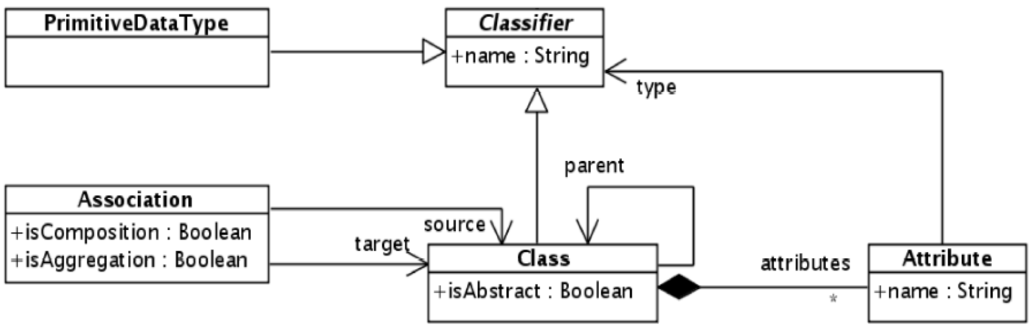


FIGURE 4: SUBSET OF THE UML MODEL

2.1.3 The Meta-Meta Model

A metamodel describes the concepts within a model. A metamodel must itself have a metamodel for describing the concepts available for metamodeling. This is the role of the meta-meta model. Meta-meta models are important not only for defining metamodels, but also as a basis for integration between different metamodels.

Each level of model defines an abstract syntax for the model in the level below; metamodels define the abstract syntax for a model, meta-meta models define an abstract syntax for metamodels. In theory this ever increasing layering of abstract syntaxes could continue ad infinitum, however the abstract syntax for a meta-meta model is defined within itself.

2.1.4 OMG Modelling Standards and Their Influence

The role of the OMG Standard, MDA was mentioned earlier. MDA actually has a number of closely related standards that shape the way all modelling is performed. These are outlined below.

2.1.4.1 Unified Modeling Language (UML)

UML [14] is a standardised, general purpose modelling notation defined by the OMG. The current state of practice in MDSD sees UML employed as the primary modelling notation. However, it is important to realise that UML is not the only modelling notation available; it is simply the most widely used.

2.1.4.2 Meta-Object Facility (MOF)

MOF [15] is the standardised meta-meta modelling notation defined by the OMG. Being a meta-meta model notation, MOF is self describing and self conforming. MDA uses MOF as its meta-meta modelling notation. This means that in order for a modelling system to be fully MDA compliant, metamodels must conform to MOF [16]. The metamodel of UML is defined using MOF.

2.1.4.3 XML Metadata Interchange (XMI)

XMI [17] is an OMG standard for the exchange of metadata via XML. XMI can be used for any data whose metamodel can be expressed in MOF. It is typically used for the serialization of models as it represents a vendor independent standard.

2.1.4.4 Object Constraint Language (OCL)

OCL [18] is a declarative, textual language for describing rules and constraints on MOF compliant models. It is widely used throughout model-driven tooling. OCL makes model definition more precise by associating assertions with model elements. It is also used in M2M transformations to specify conditional mapping of elements.

2.1.5 Modelling Levels

Figure 5 presents the relationship between the different levels of models in a model-driven architecture using the UML (left) and a typical model-driven flow (right).

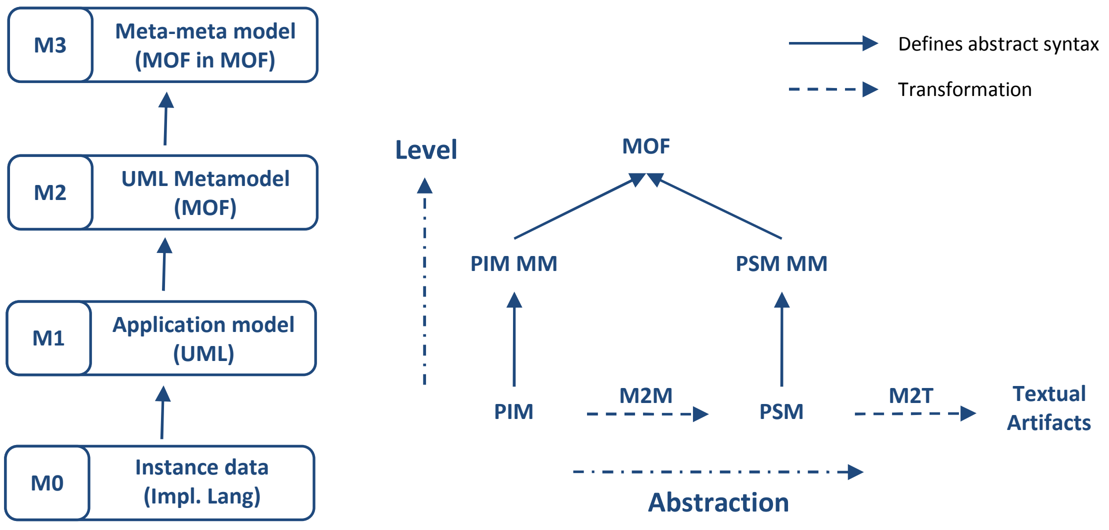


FIGURE 5: MODELLING LEVELS AND THE MODEL DRIVEN FLOW

M1 represents the model of the domain e.g. a traditional UML model. Level M2 is the model of the modelling notation i.e. the metamodel. M3 is the model of the metamodel; the meta-meta model. The meta-meta model is defined in itself. M0 is the instance data for the application used to define real world objects. The increasing numbers from M0 to M3 represent different levels of abstract syntax. The arrows between each level also represent conformance; we say that each model conforms to (is an instance of) its metamodel.

The right hand side of the diagram shows a typical model-driven flow. It is important to clarify that a metamodel is *not* an abstraction of a model; rather it is a PIM that is an abstraction of a PSM. The notion of abstraction is therefore a separate one that lies orthogonally to the modelling levels as is emphasized by the axis labels on the figure.

2.2 Model Transformations

The typical model-driven life cycle was presented in the Introduction, where a PIM is transformed into one or more PSMs, which are in turn transformed into code. All model transformations fall into one of three categories: Model-to-Model (M2M), Model-to-Text (M2T) or the previously unmentioned Text-to-Model (T2M). In this section a more detailed description is given on each of these types of transformation and the sort of tooling available to perform them.

2.2.1 Model-to-Model (M2M)

An M2M flow transforms one model into a new model. Such transformations are defined as a mapping from the source metamodel to the target metamodel. M2M transformations are most commonly used to map from a PIM into a PSM.

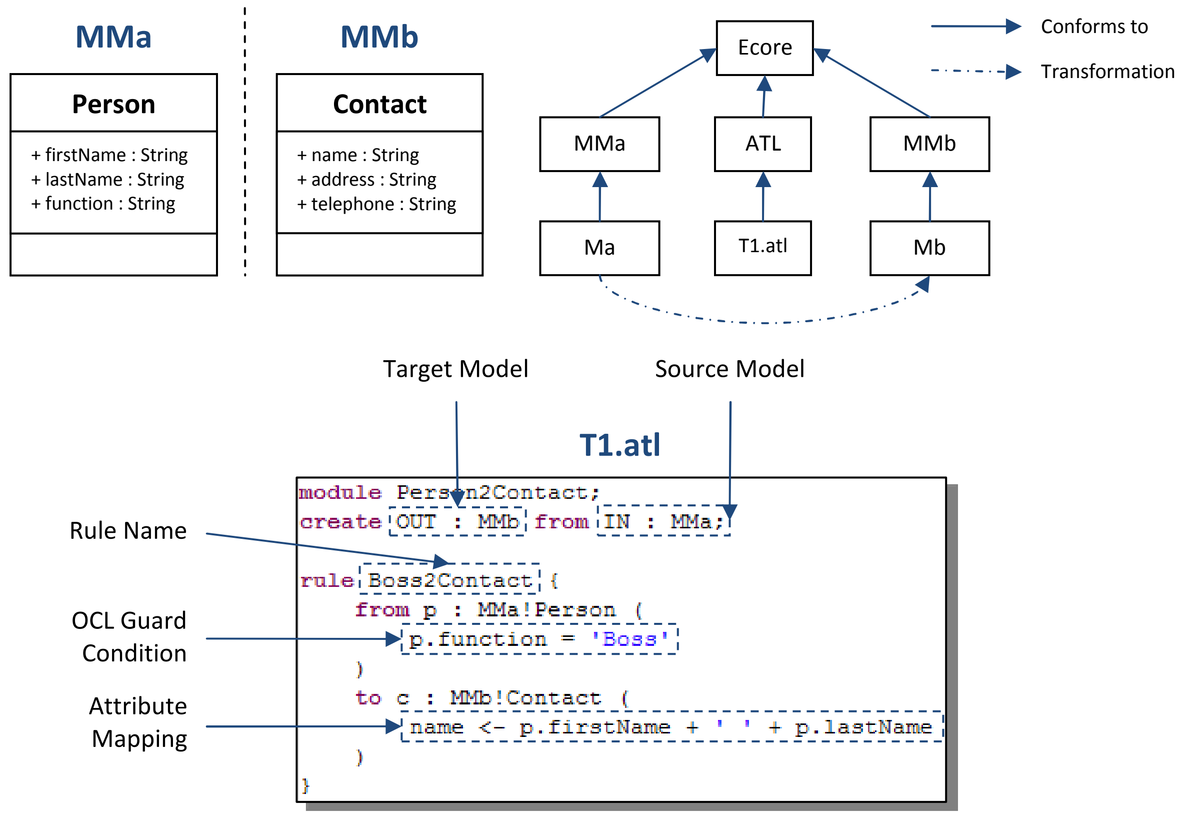
2.2.1.1 ATLAS Transformation Language (ATL)

The ATLAS Transformation Language (ATL) [19] is a model transformation language developed by the ATLAS Group. ATL enables a set of source models to be transformed into a set of target models.

The typical structure of an ATL file is a series of rules and a set of side-effect free functions known as Helpers. A rule consists of:

- A unique name
- Source language and target language
- Constants used
- Set of named source language model elements, S
- Set of named target language model elements, T
- Source language condition (in OCL) in terms of S
- Target language condition (in OCL) in terms of T
- Set of mapping rules mapping elements of S to elements of T

Figure 6 shows a very simple example of some ATL code. The ATL program, T1.atl, will take model Ma as input and produce model Mb as output. The model Ma conforms to metamodel MMa. Similarly, model Mb conforms to metamodel MMb. The program itself is also a model; it conforms to the ATL metamodel**.** The ATL code itself describes a single mapping rule that maps Person elements from MMa to Contact elements in MMb. The rule uses a guard condition to ensure that only Person elements whose function is “Boss” are mapped. The name of the mapped Contact is the concatenation of the firstName and lastName of the Person.



```
module Person2Contact;
create OUT : MMb from IN : MMa;

rule Boss2Contact {
    from p : MMa!Person (
        p.function = 'Boss'
    )
    to c : MMb!Contact (
        name <- p.firstName + ' ' + p.lastName
    )
}
```

FIGURE 6: SIMPLE ATL EXAMPLE

2.2.2 Model-to-Text (M2T)

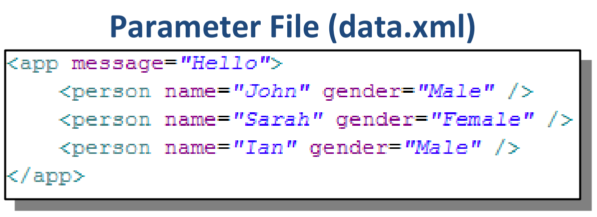
An M2T flow transforms a source model into one or more textual artifacts. This transformation is typically seen as the "code-generation" stage, however in practice any artefact containing primitive text can be generated e.g. XML documents, configuration & property files. M2T transformations rely on the use templates combined with queried data from the source model.

2.2.2.1 Java Emitter Templates (JET)

JET [20] is a template based, code generation language. It uses a number of components to generate textual artefacts from input models:

- Template files – provide a standard structure for the artefact being generated
- Control files – control the overall generation process e.g. output filenames and destinations
- Parameter files – the data the templates query i.e. the model
- JET Tags – a tag library enabling embedded control of output within a template
- XPath [21] – used to refer to nodes and attributes within the source model

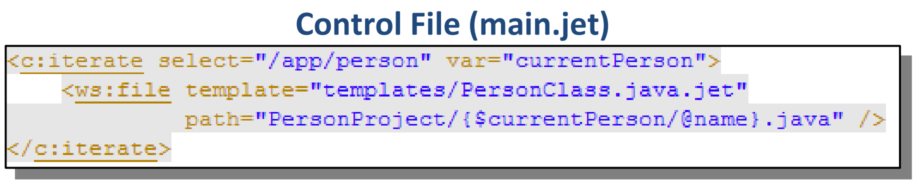
To understand how all of these files work together a small example is presented. Figure 7 shows the data used in this transformation. This example is a plain XML file containing details on three people and their gender. In practice this data would be a serialized model in XMI.


Parameter File (data.xml)

```
<app message="Hello">
    <person name="John" gender="Male" />
    <person name="Sarah" gender="Female" />
    <person name="Ian" gender="Male" />
</app>
```

FIGURE 7: EXAMPLE JET TRANSFORM INPUT DATA

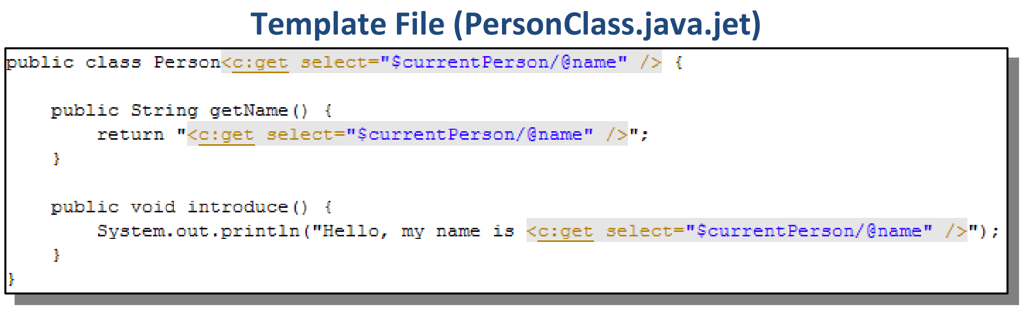
Figure 8 shows a snippet from a control file. Control files are used to manage the execution of transformations that take place. This involves specifying which templates to use and where the generated files should be output. The excerpt below iterates over all the instances of Person in the input data and creates a Java file in the PersonProject directory for each one.


Control File (main.jet)

```
<c:iterate select="/app/person" var="currentPerson">
    <ws:file template="templates/PersonClass.java.jet"
             path="PersonProject/{$currentPerson/@name}.java" />
</c:iterate>
```

FIGURE 8: JET CONTROL FILE SNIPPET

The template file PersonClass.java.jet is presented in Figure 9. This template produces a Java class containing two methods. The bodies of these methods contain tags that query the input data and retrieve the name of the current Person instance being processed. XPath is used here to navigate through the XML and retrieve the values at the nodes.


Template File (PersonClass.java.jet)

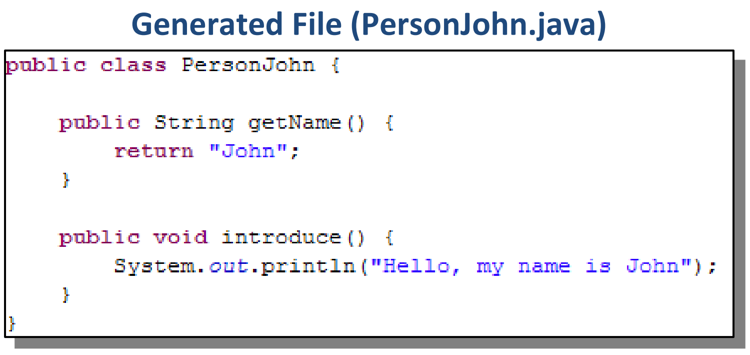
```
public class Person<c:get select="$currentPerson/@name" /> {

    public String getName() {
        return "<c:get select="$currentPerson/@name" />";
    }

    public void introduce() {
        System.out.println("Hello, my name is <c:get select="$currentPerson/@name" />");
    }
}
```

FIGURE 9: JET TEMPLATE FILE EXAMPLE

The resulting execution of the transformation causes files as shown in Figure 10 to be generated. Notice that all of the strings that were previously tag expressions have been resolved into data from the model. PersonJohn.java is one of three files that have been generated along with PersonSarah.java and PersonIan.java.


Generated File (PersonJohn.java)

```
public class PersonJohn {

    public String getName() {
        return "John";
    }

    public void introduce() {
        System.out.println("Hello, my name is John");
    }
}
```

FIGURE 10: RESULTANT GENERATED JAVA CLASS

Although this example is very simple, it shows how the principles could be applied to larger scale systems that use many templates and much more detailed data files.

2.2.3 Text-to-Model (T2M)

This third category of transformation is T2M. T2M transformations take model instance data in the form of a textual artefact and use generated parsing tools to create an instance of a model. This model instance can then be manipulated with either M2M or M2T transformations. T2M transformations are out of the scope of the tool and are mentioned here for completeness only.

2.3 Eclipse Modeling Framework (EMF)

The two most established existing website generators, AndroMDA and WebML, are both built from bespoke tooling. However, as MDSD becomes more popular, the Eclipse Modeling Framework (EMF) is quickly becoming the de facto standard for developing model-driven applications. In this section the reasons for this are presented and some of the components of EMF that are relevant to the project are discussed.

2.3.1 What is Eclipse?

Eclipse is a multi language software development IDE. The functionality of Eclipse is extended through plug-ins. Other than a small kernel, everything in Eclipse is a plug-in. Plug-ins are typically developed as part of Eclipse Projects. Eclipse Projects enhance the functionality of Eclipse in a wide range of areas such as Web Tools, C/C++ Development, Testing & Performance tools and PHP Development.

2.3.2 EMF Project

One of the projects within Eclipse is the Eclipse Modeling Project. The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies by providing a unified set of modelling frameworks, tooling, and standards implementations [22].

The Eclipse Modeling Project contains a subproject known as the Eclipse Modeling Framework (EMF). The EMF project is a modelling framework and code generation facility for building tools and other Java applications based on a structured data model [23].

There are two main reasons why EMF is gaining popularity as a platform: its compliance to standards and its enthusiastic developer community. By adhering to standards, such as MDA, interoperability between other standardised tooling is guaranteed. This enables developers to have confidence that their applications are not constrained to use on specific bespoke or proprietary systems. The highly active developer community working on the Eclipse Modeling Project ensures that progress and improvements to tooling are being made on a frequent basis. It also provides a stronger support network when help is required on the use of tooling.

2.3.3 Model Definition within EMF

2.3.3.1 Ecore

Ecore is the metamodel of EMF. Ecore models are composed of concepts such as class, data type, attribute, reference, operation; i.e. the Ecore abstract syntax has a subset of the expressiveness of a UML class diagram. Ecore is virtually identical to a subset of MOF known as Essential MOF (EMOF) other than for some minor naming differences. Ecore was originally intended for defining the abstract syntax of a metamodel. However, the availability of tooling for generating implementations of an Ecore model means that they are often used as application models. When used for this purpose, Ecore models can only capture the data definition aspects of an application. Figure 11 below shows the structure of the Ecore components.

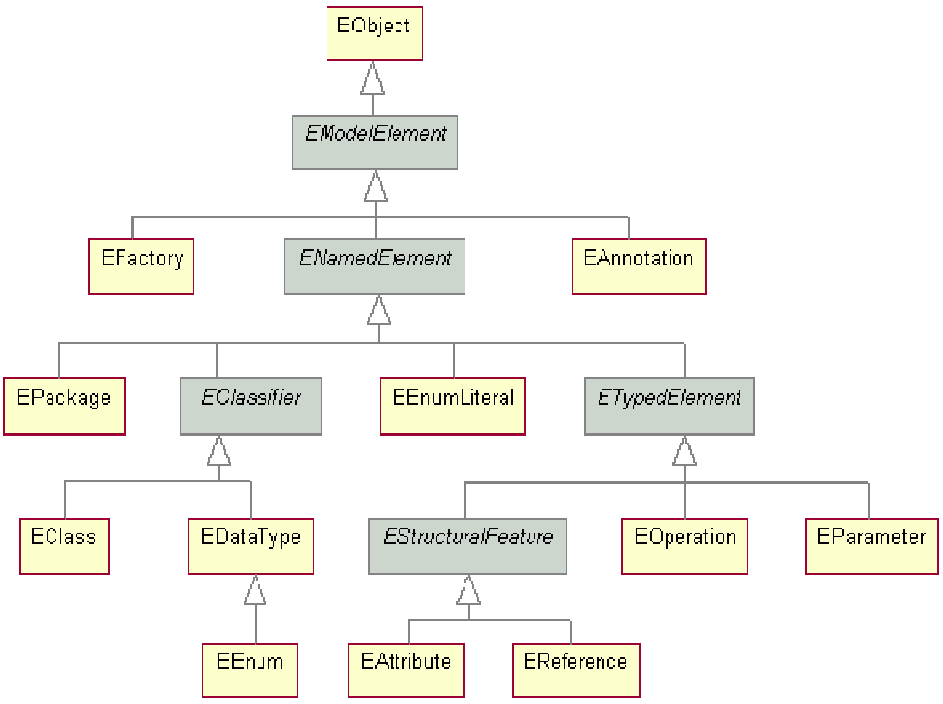
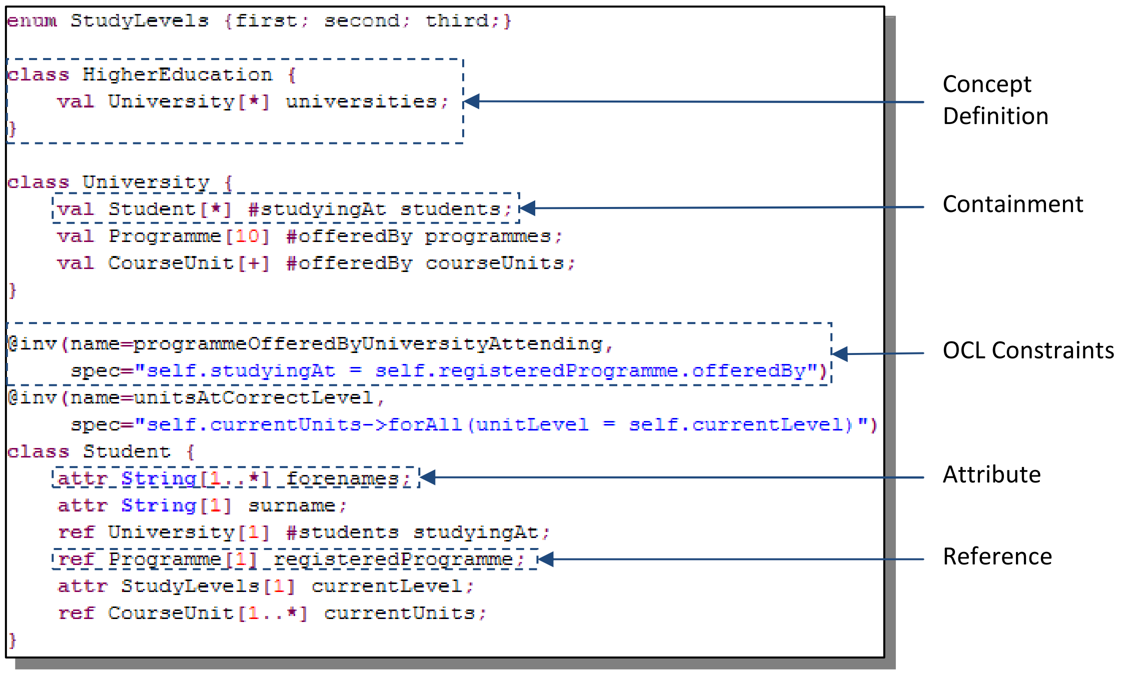


FIGURE 11: ECORE COMPONENTS [24]

2.3.3.2 Emfatic

Emfatic [25] provides a textual concrete syntax for the definition of Ecore models. It is described as providing "a Java-like syntax familiar to many programmers" [26]. An example code snippet of Emfatic is shown in Figure 12. This highlights a few of the main features of Emfatic and demonstrates some of the language's syntax. The code snippet defines a selection of concepts from the model of a University system. It contains the concept definitions for Higher Education, University and Student. The attributes and relationships are defined along with their cardinalities within the class they belong to.


```
enum StudyLevels {first; second; third;}

class HigherEducation {
    val University[*] universities;
}

class University {
    val Student[*] #studyingAt students;
    val Programme[10] #offeredBy programmes;
    val CourseUnit[+] #offeredBy courseUnits;
}

@inv(name=programmeOfferedByUniversityAttending,
     spec="self.studyingAt = self.registeredProgramme.offeredBy")
@inv(name=unitsAtCorrectLevel,
     spec="self.currentUnits->forAll(unitLevel = self.currentLevel)")
class Student {
    attr String[1..*] forenames;
    attr String[1] surname;
    ref University[1] #students studyingAt;
    ref Programme[1] registeredProgramme;
    attr StudyLevels[1] currentLevel;
    ref CourseUnit[1..*] currentUnits;
}
```

FIGURE 12: EMFATIC EXAMPLE CODE SNIPPET

2.3.3.3 *Ecore Tools*

Ecore Tools provide a graphical concrete syntax for the definition of Ecore models. Due to the similarity between Ecore and UML, the graphical editor within the Ecore Tools should be familiar to many developers. Graphical editors are typically easier to use initially than their textual counterpart, however this benefit tends to fade as the developer becomes more familiar with the textual syntax. Graphical editors also tend to hide a great deal of detail from users. Again, for more experienced developers this is often not desirable and so using a textual syntax such as Emfatic is preferred.

Figure 13 represents the same model as that in Figure 12 drawn in Ecore Tools. As you can see, this graphical syntax needs no real explanation due to its intuitive visual representation and similarity to UML.

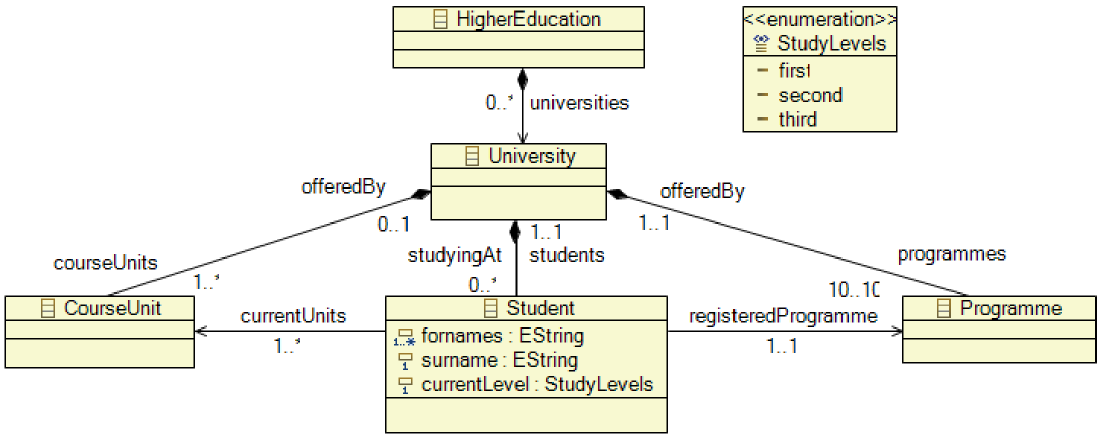


FIGURE 13: ECORE TOOLS EXAMPLE

2.4 Graphical Modeling Framework (GMF)

One of the subprojects of the Eclipse Modeling Project is the Graphical Modeling Framework (GMF). Part of the functionality GMF provides is a generative component and runtime infrastructure for developing graphical editors based on EMF [27]. The graphical editor from Ecore Tools is built on GMF; a screenshot of this is shown in Figure 14.

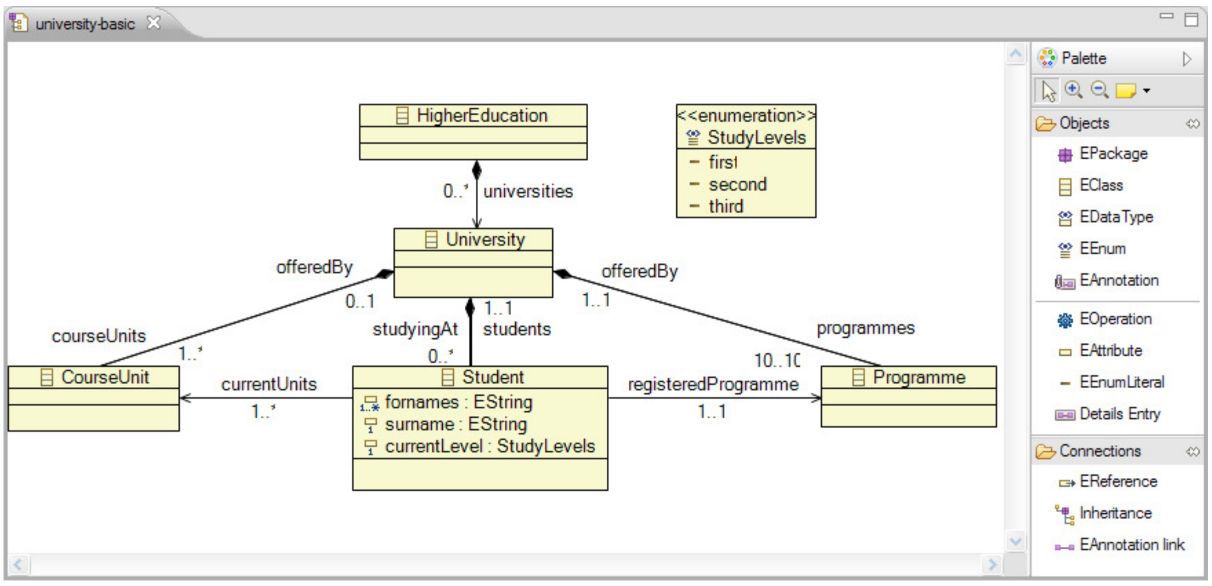


FIGURE 14: GMF EDITOR EXAMPLE

GMF enables the generation of fully functional graphical editors from model definitions. It requires the input of four models:

1. genmodel – this model contains the elements of the model whose instances will be created by the editor
2. gmftool – this model determines the tools that will be available in the editor: it controls what appears in the Palette on the right hand side of Figure 14
3. gmfgraph – this model controls what elements can appear on the diagram and the elements that can be used to construct it
4. gmfmap – this model is the most important as it provides a mapping between the contents of the genmodel and the information defined in the gmftool and gmfgraph models.

Once these models are set up, a model editor can be generated by following a wizard. Generation typically takes around 1 minute, after which time a fully functional graphical model editor is available as an Eclipse plug-in.

2.5 Summary

This chapter has discussed a range of modelling concepts and technologies. A summary of the key points is presented to aid the understanding of the reader.

- Concepts within models are described by metamodels
- Metamodels represent an abstract syntax that the model conforms to
- PIMs are abstractions of PSMs
- EMF is one of the most widely used modelling frameworks
- Ecore is the metamodeling notation within EMF
- There are multiple concrete syntaxes for Ecore, both textual and graphical

Chapter 3 Data Intensive Websites

Before discussing the tool itself, it is vital to have a clear understanding of the websites it generates. Moreover, it is important to realise that the development of the websites has been performed in the same rigorous manner as if they were hand coded. This emphasizes the fact that the quality of the generated software artifacts is mutually exclusive to the use of MDSD.

3.1 Design

The website is built on a classic three-tier architecture (see Figure 1). By using this architecture a separation of concerns can be achieved whereby details of each layer can be modified in isolation without affecting the other layers. Figure 15 shows a slightly more detailed structural breakdown of the website architecture that the tooling produces. The architecture possesses three distinct layers: persistence, business logic and presentation, and is built upon a web application framework. The details of the architecture are discussed below.

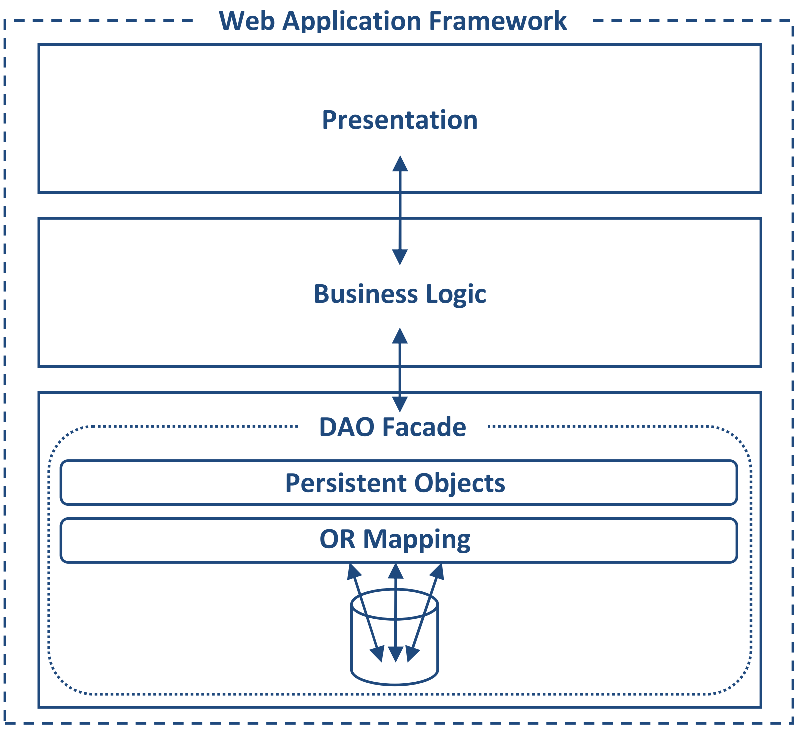


FIGURE 15: STRUCTURAL DESIGN OF A DATA INTENSIVE WEBSITE

3.1.1 Persistence

The persistence layer is the lowest layer in the three-tier architecture. This layer is responsible for storing and retrieving data from a backend database.

The choice of data model for the persistence layer is an important one. The data model used in modern data intensive websites is typically Object-Relational (OR). The OR model was introduced in order to reduce the impedance mismatch caused by the original Relational data model [28]. The OR approach provides a much more natural mapping to object oriented (OO) languages used in the business layer. Consequently, for data intensive websites, where the creation and analysis of data is paramount, the use of the OR data model is the most logical approach. Due to this decision, the persistence layer is comprised of both persistent objects and an OR mapping layer that constructs these objects from the data stored in the database.

In order to enhance the separation of concerns between the persistence layer and business logic layer, data operations that access the persistent objects are wrapped by a DAO facade. DAO is a design pattern that stands for Data Access Object. The facade contains DAO objects that provide an abstract interface to the persistent objects in the system. DAO objects contain the code to perform database operations; they encapsulate CRUD functionality in four methods:

1. *saveOrUpdate()* – persist new instances of entities or update existing ones
2. *findAll()* – retrieve all instances of a particular entity
3. *find()* – retrieve a particular instance of an entity with a given ID
4. *delete()* – remove an instance of an entity from the database

DAO objects hide the implementation details of the underlying persistence mechanism, completely decoupling the business logic and persistence. Figure 16 shows the architecture of a possible DAO implementation. The diagram shows an object in the business logic layer making the same method calls to different DAO objects each of which has a different underlying implementation. The code within each DAO object will be specific to the implementation technology it encapsulates, however its interface remains unchanged. This approach makes changing between persistence implementations trivial.

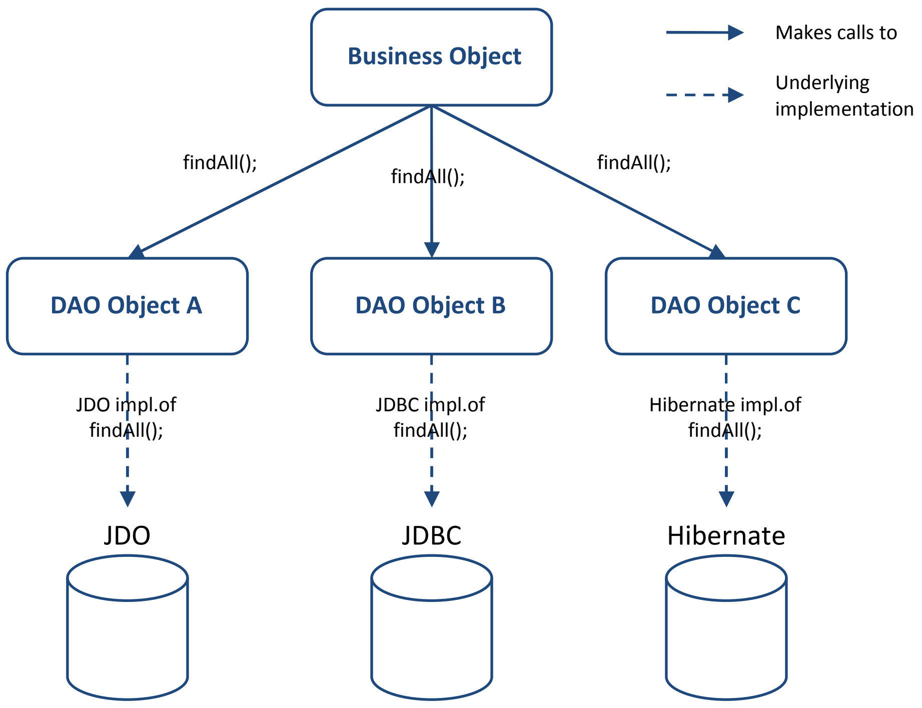


FIGURE 16: DAO ARCHITECTURE

3.1.2 Business Logic

The business logic tier is where the bulk of the processing within the website takes place. It is responsible for retrieving data from the persistence layer, processing it and providing the subsequent information to the presentation layer.

Due to the nature of the functionality within the website, this layer is largely transparent; it is designed to provide generic CRUD functionality rather than any specific analytics or calculations. The majority of the processing within the business logic layer involves communication with one or more DAO objects in order to retrieve persistent data. The design of the business logic layer is therefore relatively straightforward.

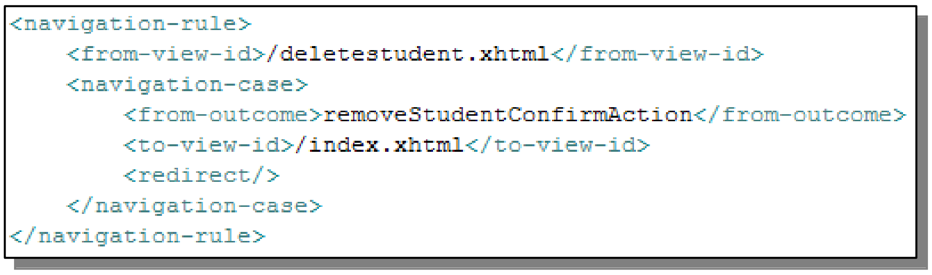
3.1.3 Presentation

The presentation tier is the uppermost layer in the three layer stack. This layer represents the user interface to the application. It is responsible for presenting information supplied by the logic layer in a meaningful way and providing controls for interaction.

The design of the presentation layer requires a number of key decisions to be made. Firstly, the divide in website functionality between the client and server must be considered. There are advantages and disadvantages of adopting both thick and thin client architectures depending on the application of the website. Data intensive websites by their very nature require dynamic content to be injected into web pages. Therefore, the majority of the functionality must be performed on the server side; data access and page composition can only be done on the server. For dynamic functionality within the website e.g. displaying a pop up, client side languages such as JavaScript are very useful. Similarly, being able to perform validation without requiring a page refresh increases the responsiveness of websites. However, there can be no guarantee that a client has scripting facilities enabled in their browser. Consequently, the website under development tries to minimise the amount of client side based functionality in favour of server side.

The way in which confirmations of actions are presented to the user also requires a choice. For example, when a record is to be deleted does the website enforce navigation to another page and thus another server request or can the action be performed asynchronously without requiring a page refresh. For functionality other than deletion, it has been decided that the issue of confirmations is largely negligible in the early stages of the tooling. However, for the deletion of data the website ensures that either form of confirmation can be performed.

The navigation model of the website also requires some consideration. In the past, navigation between through the website was internal and hard coded into the pages themselves. However, newer web technologies such as Apache Struts [29] and JSF [30] use an external navigation model where navigation is controlled by a separate XML file containing navigation rules. A navigation rule describes a movement from one page to another when a particular outcome occurs (see Figure 17). This approach is very useful as it enables page navigation that relies on conditional logic to be managed centrally. The website will use an external navigation model.



```
<navigation-rule>
    <from-view-id>/deletestudent.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>removeStudentConfirmAction</from-outcome>
        <to-view-id>/index.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
```

FIGURE 17: JSF NAVIGATION RULE EXAMPLE

Finally, a decision must be made on the way messages are encoded into pages. Messages refer to strings of text within the website for example, database connection details or the names of attributes and relationships. In the same way to the navigational model, these strings can be encoded internally or externally. Modern web technologies advocate the reading of values from external property bundles. Property bundles are sets of key-value pairs that can be referenced from multiple locations. The benefit of externalising these strings is that they can be maintained centrally in one location. This means that modification of the values is trivial and that any changes to the bundle automatically propagate through to the files that reference it. The website will use external message encoding for reading properties.

3.1.4 Web Application Framework

A web application framework is a set of software APIs / libraries that provide code for generic areas of functionality within a website. This typically includes modules for areas such as testing, transaction management, data access and session management. By building the website on a framework, development can be focussed on the main functional requirements of the system rather than being concerned with the low level implementation of standard features. Frameworks represent a way of significantly reducing software development time by minimising the amount of code that needs to be written. Since this approach makes sense for hand developed websites, it should also be applied to generated ones.

One of the main features the website will incorporate is the automatic management of transactions. Ideally, this would involve not only the opening and closing of transactions but also the handling of exceptions that may occur during the transactions themselves. Automatic session management will also be leveraged in a similar way. For a data intensive website, automatic management of the persistence layer is a key requirement as this represents its largest area of functionality.

3.2 Implementation

This section looks at the implementation of the website. It includes an analysis of the key implementation technologies and discusses how the various components and layers of the website fit together.

3.2.1 Persistence

The architectural design of the persistence layer has already been hinted at in Figure 15 and Figure 16. Figure 18 shows an overview of the concrete implementation of the persistence layer. The left hand side of the diagram presents a conceptual view of the different layers within the tier, the right hand side shows an example instance of a persistent mapping for a Student entity.

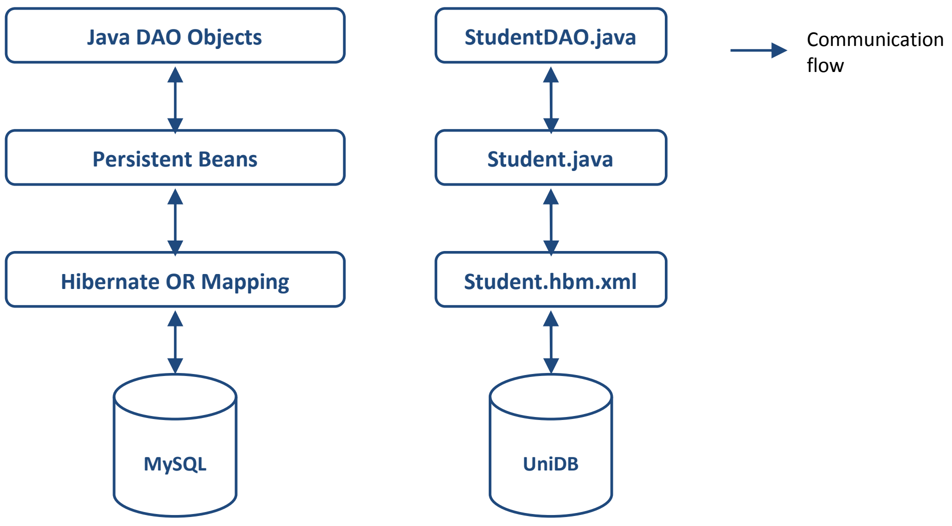
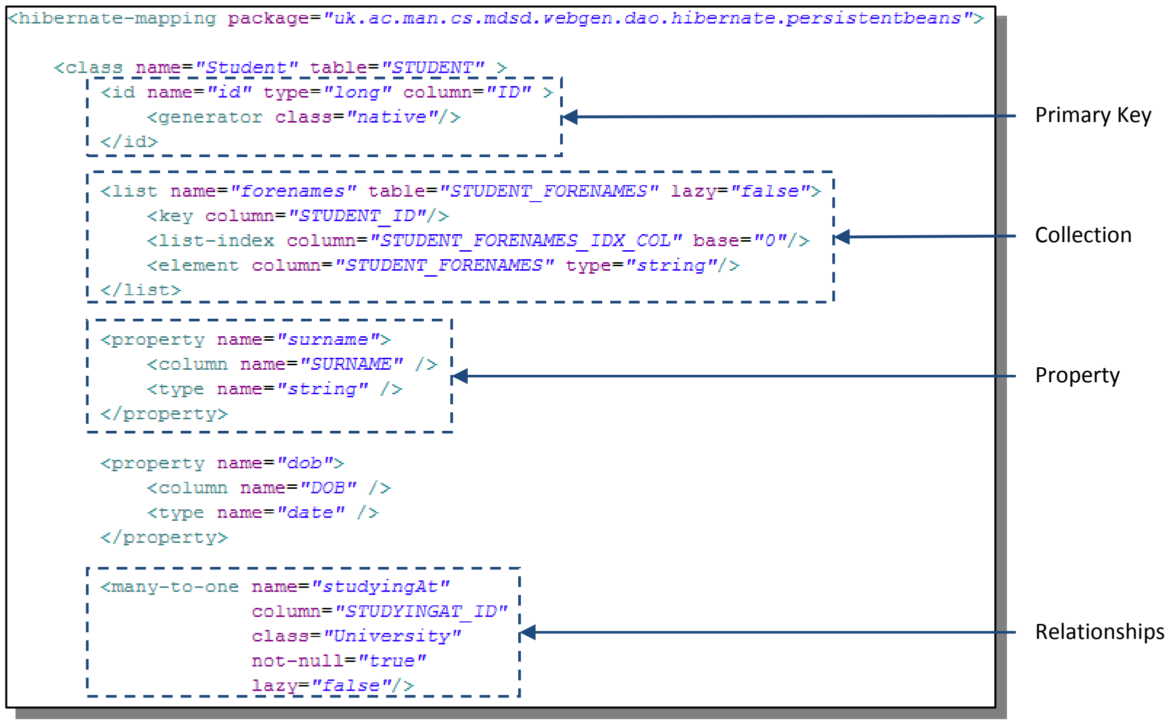


FIGURE 18: WEBSITE PERSISTENCE LAYER IMPLEMENTATION

Java has been selected as the OO language of choice for the implementation of the website. Both the persistent beans and DAO objects are written in Java. Persistent beans are objects that represent the persistent entities within the system. Each bean contains a Java representation of the attributes and relationships of the entity it encapsulates. The website also implements the DAO design pattern. A DAO object is a Java class that contains data access methods for a particular persistent entity. Each entity has its own DAO object for accessing the persistent instances of that entity.

There are numerous OR mapping tools for Java, some of the main ones are Java Data Objects (JDO), Hibernate and TopLink by Oracle. A decision was taken to use Hibernate [31] as the OR mapping tool for the website. Unlike TopLink and JDO, Hibernate is free and open source. This has led to Hibernate possessing a wider user base amongst developers. The API for Hibernate is also much less verbose than those of implementations of JDO. This means that Hibernate code is tidier and typically easier to write. Furthermore, the web application framework that the system is built on provides excellent support for integration with Hibernate meaning that use of the technology is simplified greatly. Further advantages of Hibernate over JDO can be found here [32].

As with other OR technologies such as JDO, Hibernate uses XML mapping files to describe how objects are persisted and to enable the injection of data into the persistent beans. An example snippet showing the main features of a mapping file is shown in Figure 19. This snippet shows part of the mapping for a Student entity. Different XML tags are used to encode the various types of attributes and relationships an entity may possess. The code below specifies that the Student class is uniquely identified by a field called ID. It also states that a Student has forenames that are stored as a list of strings. The encoding for basic properties such as surname and date of birth is then displayed. The final tag describes a many-to-one relationship that Student has with University.



```
<hibernate-mapping package="uk.ac.man.cs.mdsd.webgen.dao.hibernate.persistentbeans">

    <class name="Student" table="STUDENT" >
      <id name="id" type="long" column="ID" >
          <generator class="native"/>
      </id>

      <list name="forenames" table="STUDENT_FORENAMES" lazy="false">
          <key column="STUDENT_ID"/>
          <list-index column="STUDENT_FORENAMES_IDX_COL" base="0"/>
          <element column="STUDENT_FORENAMES" type="string"/>
      </list>

      <property name="surname">
          <column name="SURNAME" />
          <type name="string" />
      </property>

      <property name="dob">
          <column name="DOB" />
          <type name="date" />
      </property>

      <many-to-one name="studyingAt"
                   column="STUDYINGAT_ID"
                   class="University"
                   not-null="true"
                   lazy="false"/>
```

FIGURE 19: HIBERNATE MAPPING EXAMPLE CODE SNIPPET

3.2.2 Business Logic

A detailed breakdown of the middle tier is presented in Figure 20. Although this layer is largely transparent, it does possess some interesting areas of functionality.

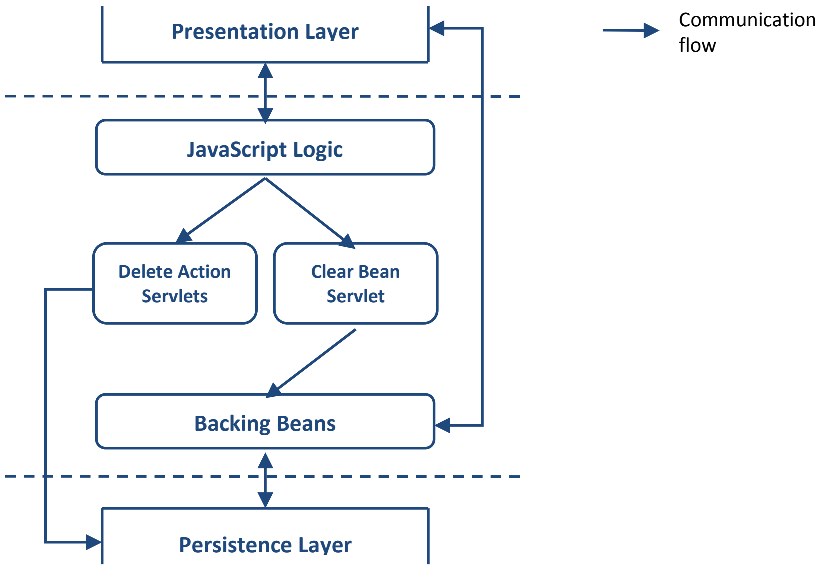


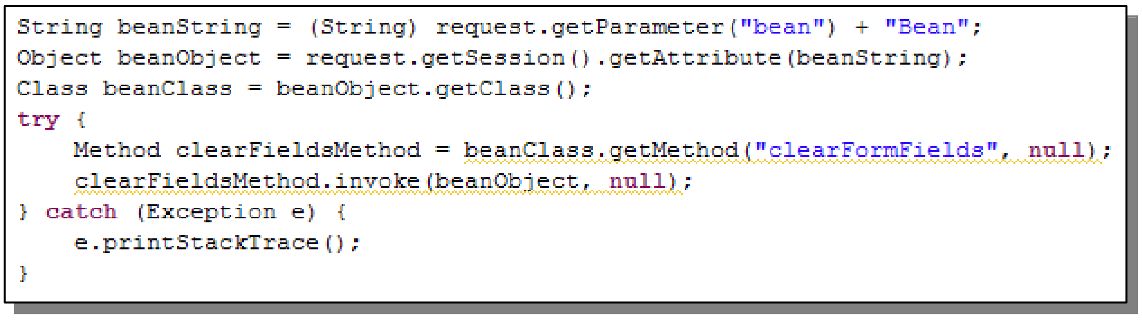
FIGURE 20: WEBSITE BUSINESS LOGIC LAYER IMPLEMENTATION

The main feature of the business logic layer is the backing beans. Backing beans are Java classes that provide information and co-ordinate responses to events that take place in the presentation layer. Each entity has its own backing bean that essentially acts as a wrapper over the persistent bean in the layer below. Backing beans have a similar structure to their persistent counterparts but also contain functionality. The majority of this functionality is implemented through action listeners that respond to events in the presentation layer. The action listener methods perform CRUD database operations via the DAO objects in the persistence layer below.

Servlets are also an important feature within the logic layer. Servlets are Java objects that dynamically process requests and construct responses. They are used to extend and enhance the functionality of web servers by implementing dynamic functionality. Servlets are used in two areas of the website: for performing delete actions and for clearing data within beans. The execution of these Servlets is controlled by a thin layer of JavaScript logic. JavaScript is a client side language. As backing bean and persistence layer technologies reside on the server side, Servlets are used to bridge this gap between client and server, enabling server side functionality to be invoked from the client side.

The Delete Action Servlets contain the functionality to perform a delete operation on a given class instance by accessing that class' DAO object. With delete actions, the deletion is performed asynchronously (using AJAX) without requiring a page request to the server. In order to accomplish this, delete actions use a JavaScript pop up dialog box to request confirmation of the deletion of the chosen entity. If the user confirms, the JavaScript layer invokes the execution of the delete action servlet. Deletion is currently the only data manipulation operation that can be performed via a Servlet as well as through a backing bean. This is because it is the only operation that can be logically performed in two ways.

When navigating between pages within the website, old form data from previous CRUD operations can remain stored in the backing beans. As a result, navigating to a new page can result in old data being displayed; this is undesirable. The Clear Bean Servlet uses functionality from the Java Reflection API in order to construct a method call that clears the data stored in a given backing bean instance. The Java Reflection API [33] enables analysis and modification of run time behaviour of applications running in a JVM. It is used in this instance to avoid the need to create separate, almost identical Servlets to clear data within each backing bean. The JavaScript that controls the execution of this Servlet is called through the *onclick* argument of a hyperlink. The code snippet in Figure 21 shows the reflective code used in this Servlet. The method is passed a string parameter that is the name of the bean whose data needs to be cleared. It continues to retrieve the backing bean object instance from within the application and invoke the required *clearFormFields* method contained in its class.



```
String beanString = (String) request.getParameter("bean") + "Bean";
Object beanObject = request.getSession().getAttribute(beanString);
Class beanClass = beanObject.getClass();
try {
    Method clearFieldsMethod = beanClass.getMethod("clearFormFields", null);
    clearFieldsMethod.invoke(beanObject, null);
} catch (Exception e) {
    e.printStackTrace();
}
```

FIGURE 21: CLEAR BEAN SERVLET REFLECTIVE CODE SNIPPET

3.2.3 Presentation

Figure 22 shows an overview of the key components within the presentation layer.

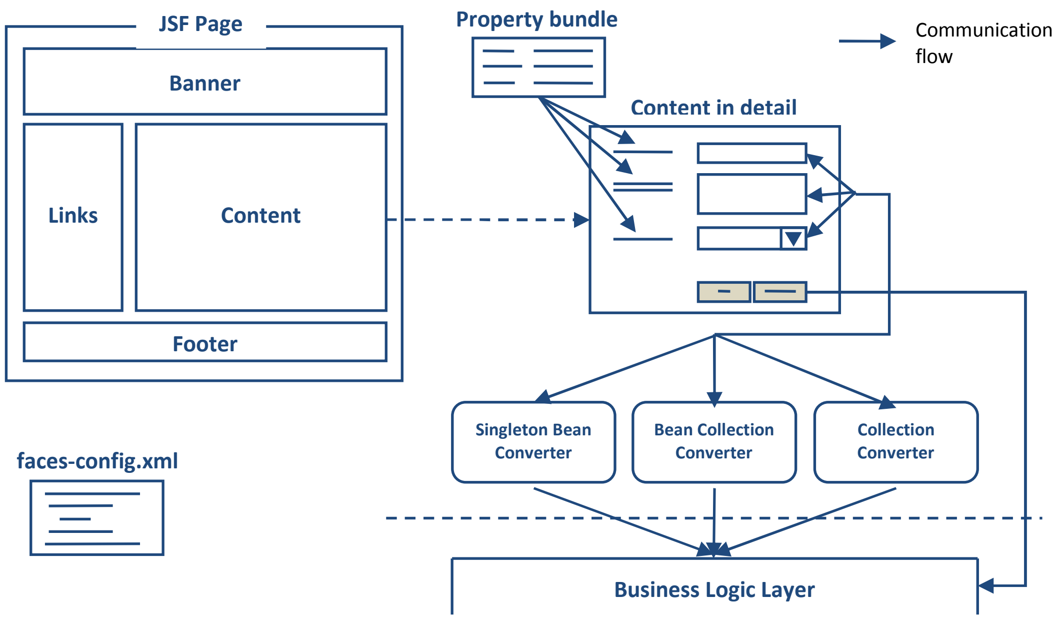


FIGURE 22: WEBSITE PRESENTATION LAYER IMPLEMENTATION

The box in the top left of the diagram represents a JavaServer Faces (JSF) page. JSF is the main technology behind the presentation layer. JSF is a Java based, web application language designed to simplify the development of user interfaces for web applications.

JSF uses JavaServer Pages (JSP) [34] as its default display technology. JSP is a Java technology that allows developers to create dynamically generated websites. The technology enables Java code to be embedded into static content. The JSP syntax also defines additional XML-like tags that act as extensions to traditional HTML. JSPs are compiled into Java Servlets by a JSP Compiler that are subsequently executed when the page is accessed.

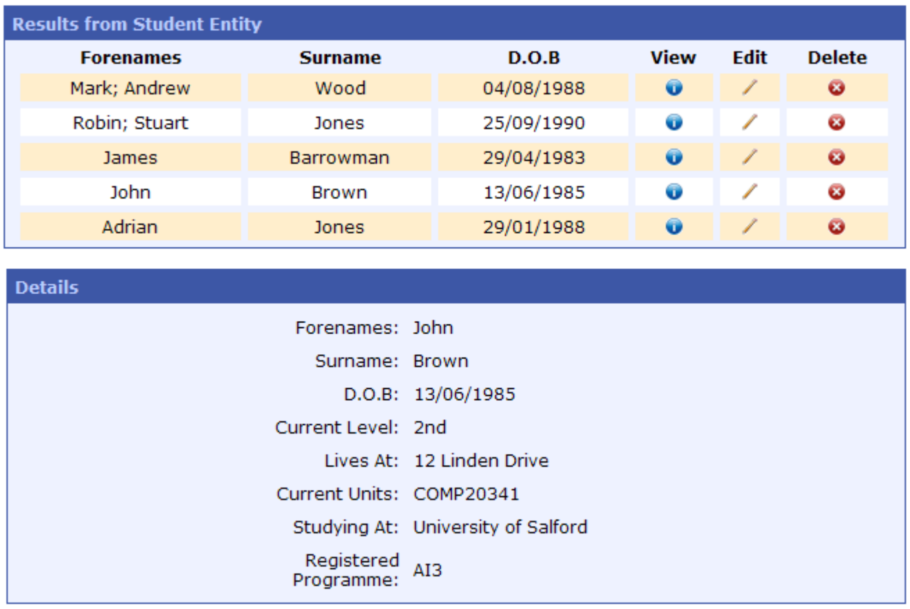
JSF builds on top of JSP and incorporates a number of features:

- A set of APIs for representing user interface (UI) components and managing their state, handling events, input validation, defining page navigation and support for internationalization and accessibility
- A default set of UI components that render low level HTML automatically
- Custom tag libraries that integrate JSF with JSP
- A server side event model

JSF is a relatively new web technology. As with many new web technologies, there was a great deal of hype surrounding the release of JSF. Obtaining reliable information about which web application language is best is difficult as virtually all of it is opinion based. JSF however does seem to have some clear advantages over other frameworks such as Apache Struts. This is mainly due to the customisable UI components JSF includes and the flexibility they provide [35]. JSF also complements the technologies used in the other layers of the system.

The JSF page in Figure 22 is divided into a number of sections, this represents a Facelets template. Facelets is an abbreviation for the JavaServer Faces View Definition Framework [36]. It is a templating language for structuring the view of JSF pages. A Page is split into the following five sections: Page Title, Banner, Content, Links and Footer. These sections each contain exactly what their name suggests.

Content within pages is in the form of CRUD functionality. Read functionality can be performed in two different ways: reading all of the features stored on one particular instance of an entity, or reading a subset of the features stored on all instances of an entity. Each Read function presents the data in a slightly different way as shown below in Figure 23. The top screenshot shows a subset of the data stored on all of the Students. The bottom screenshot shows all of the data stored on one of the Students.



Results from Student Entity

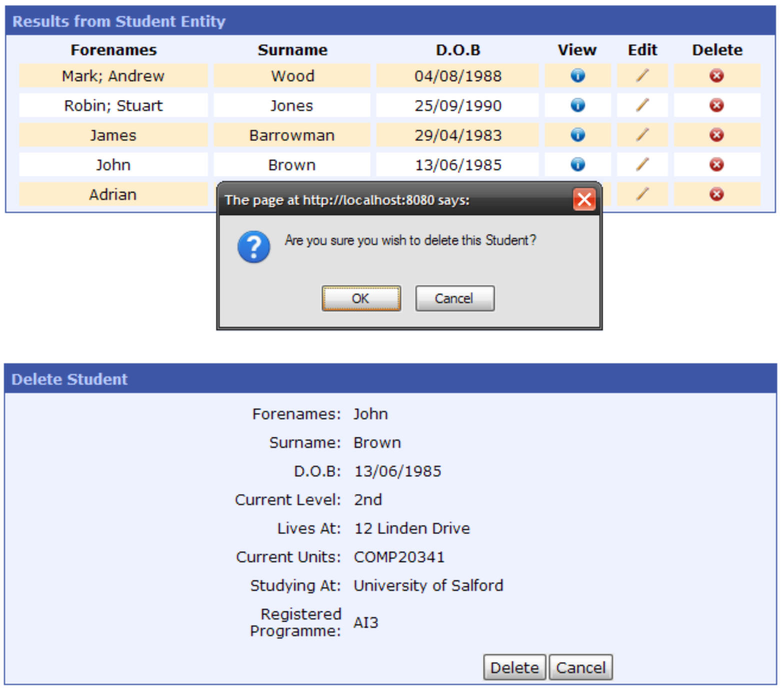
Forenames	Surname	D.O.B	View	Edit	Delete
Mark; Andrew	Wood	04/08/1988	ⓘ	✎	✖
Robin; Stuart	Jones	25/09/1990	ⓘ	✎	✖
James	Barrowman	29/04/1983	ⓘ	✎	✖
John	Brown	13/06/1985	ⓘ	✎	✖
Adrian	Jones	29/01/1988	ⓘ	✎	✖

Details

Forenames:	John
Surname:	Brown
D.O.B:	13/06/1985
Current Level:	2nd
Lives At:	12 Linden Drive
Current Units:	COMP20341
Studying At:	University of Salford
Registered Programme:	AI3

FIGURE 23: DIFFERENT IMPLEMENTATIONS OF READ FUNCTIONALITY

Deletion can also be performed in two different ways: by visiting a separate page before the deletion takes place, or deleting directly from a set of data using a pop up dialog. The two different methods of deletion are captured in Figure 24. The top screenshot shows the deletion of a Student directly from list of Students. The screenshot beneath it displays all of the data on the Student and displays a button to perform the deletion.



Results from Student Entity

Forenames	Surname	D.O.B	View	Edit	Delete
Mark; Andrew	Wood	04/08/1988	ⓘ	✎	✖
Robin; Stuart	Jones	25/09/1990	ⓘ	✎	✖
James	Barrowman	29/04/1983	ⓘ	✎	✖
John	Brown	13/06/1985	ⓘ	✎	✖
Adrian				✎	✖

The page at http://localhost:8080 says:

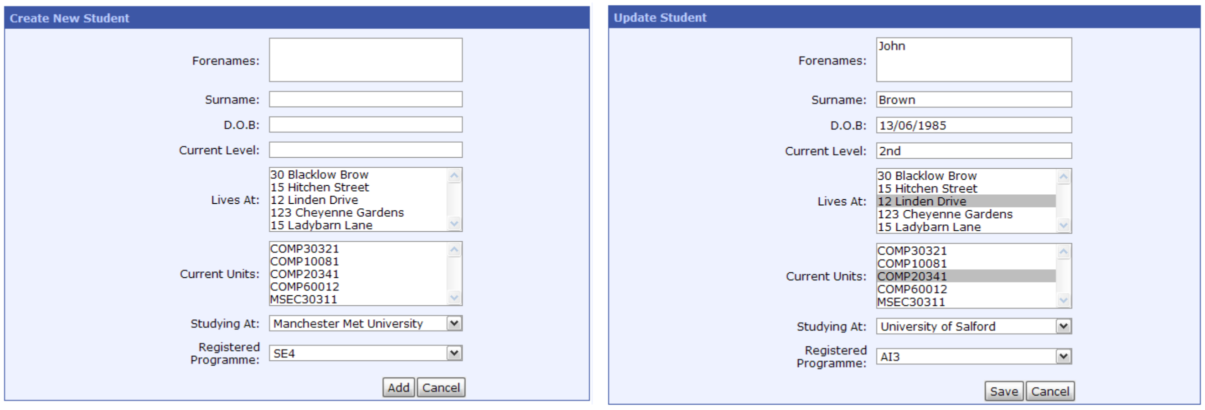
Are you sure you wish to delete this Student?

Delete Student

Forenames:	John
Surname:	Brown
D.O.B:	13/06/1985
Current Level:	2nd
Lives At:	12 Linden Drive
Current Units:	COMP20341
Studying At:	University of Salford
Registered Programme:	AI3

FIGURE 24: DIFFERENT IMPLEMENTATIONS OF DELETE FUNTIONALITY

The Create and Update functionality can currently only be performed in one way. Screenshots of these units of functionality are presented on the left and right of Figure 25 respectively for completeness.



Create New Student

Forenames:	
Surname:	
D.O.B:	
Current Level:	
Lives At:	30 Blacklow Brow; 15 Hitchen Street; 12 Linden Drive; 123 Cheyenne Gardens; 15 Ladybarn Lane
Current Units:	COMP30321; COMP10081; COMP20341; COMP60012; MSEC30311
Studying At:	Manchester Met University
Registered Programme:	SE4

Update Student

Forenames:	John
Surname:	Brown
D.O.B:	13/06/1985
Current Level:	2nd
Lives At:	30 Blacklow Brow; 15 Hitchen Street; 12 Linden Drive; 123 Cheyenne Gardens; 15 Ladybarn Lane
Current Units:	COMP30321; COMP10081; COMP20341; COMP60012; MSEC30311
Studying At:	University of Salford
Registered Programme:	AI3

FIGURE 25: IMPLEMENTATION OF CREATE AND UPDATE FUNCTIONALITY

The functionality behind each of the buttons on the pages is delegated to the business logic layer. Each button has a corresponding action listener in the relevant backing bean that responds to button presses. The attribute labels within pages are all references to external property bundles that contain the string associated with each attribute.

One of the key features of the JSF framework is Conversion. This term refers to the process of ensuring that data is of the correct type or in the correct format for displaying and processing. It typically involves conversion between strings that appear UI components and the underlying objects they represent. As well as providing standard converters for various data types, JSF also defines a Converter interface enabling developers to define their own converters. The website makes use of three classes of custom converters: Singleton Bean Converters, Bean Collection Converters and Primitive Collection Converters. Each of these is designed to map between object and string representations for the entities and primitive data types within the system to ensure they are displayed correctly.

Configuration files are arguably the most important collection of files within the website. These files wire the system together and enable the different layers of the web application to interact. The faces-config.xml file is the key configuration tool in JSF. It acts as a registry that glues the various parts of a web application together. The file contains details about the managed beans within the system, i.e. the persistent beans, and the custom converters that are used. It is also used to define the navigational flow through the website using navigation rules such as those shown in Figure 17.

3.2.4 Web Application Framework

There is now a myriad of Java Web Application Frameworks available, some of the most well known frameworks being JBoss Seam, Spring [37], Tapestry and Apache Wicket. Many of these frameworks are very similar and therefore deciding which one to use is often difficult. The decision was made to use the very popular framework, Spring; an application framework for Java and .NET. Spring is an open source project that contains a number of modules that provide services for improving software development. The Spring Framework was chosen as it contains all of the required features specified during the design of the website. Spring also complements the technologies used in each layer of the architecture.

Although not shown on any of the diagrams, three packages of the Spring framework have been used throughout the website: Spring ORM, Spring Web and Spring Context [38]. Spring ORM integrates with Hibernate in the persistence layer, managing the data source, sessions & transactions and providing APIs to implement DAO objects. These features are managed using Spring’s main configuration file; applicationContext.xml. The Spring Web and Spring Context packages both provide support for locating objects and services within the website thereby permitting interaction between them. Each of these packages provides an abstraction that minimises the amount of code required and ensures that the low level features of the system are managed automatically.

Chapter 4 Tool Architecture Design

4.1 Overview

Now that the structure and implementation of data intensive websites is understood, the tooling used to generate them can be analysed. Before detail is presented on how the generation tool is built, it is important to have a clear idea of the general flow and architecture of the model driven tooling; this is shown in Figure 26.

The tooling begins with the definition of the website metamodel; this describes the concepts within a data intensive website. A graphical editor is then used to a define website model. A Website PIM defined using the editor then undergoes a series of M2M transformations to convert the PIM into two main PSMs: Persistent and Website. The Persistent PSM is used in an M2T transformation to generate the bottom two layers of the website. In order to enable some degree of customisation of the website prior to generation, models that encapsulate Website Options and Choices have been defined. The customisation detail within these model instances is combined with the original Website PIM to produce the Website PSM. This model is then used to generate the textual artifacts within the presentation layer.

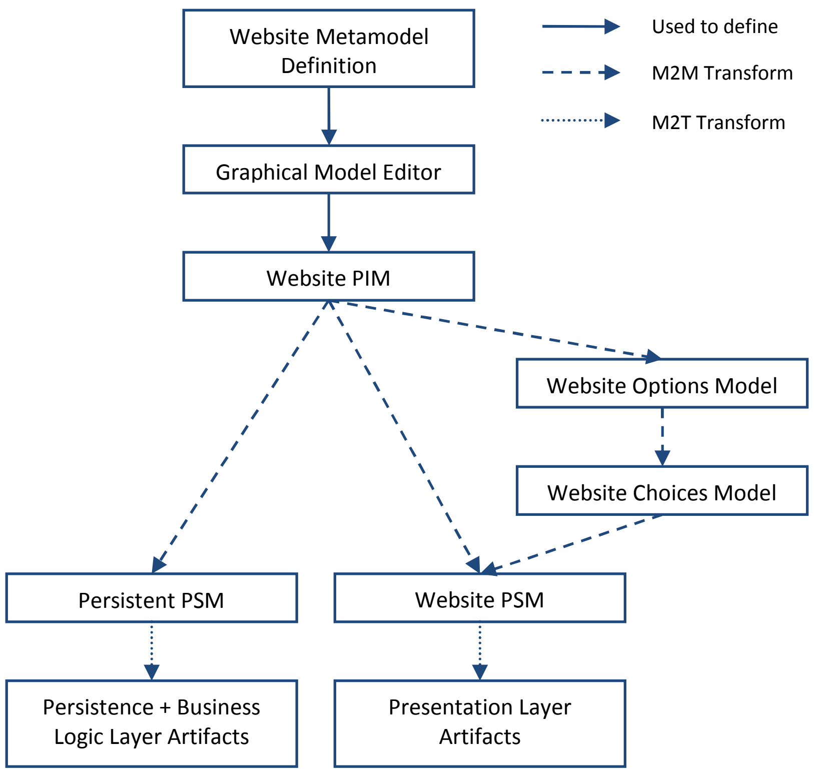


FIGURE 26: MODEL DRIVEN TOOLING ARCHITECTURE

This chapter continues to analyse the design of the components within the model driven tooling.

4.2 Tooling Metamodels: PIMs

The generation tool uses model definitions that capture the concepts of customisable data intensive websites.

4.2.1 Website PIM

The main model of the system, the Website PIM, is a model definition for data intensive websites. The model can be separated into two distinct areas: persistence and web. The persistence area refers to model elements that represent the underlying data model within the website. Conversely, the web area refers to the elements that affect the actual functionality of the website. To make this analysis more comprehensible, each area is described separately but it is important to realise that they are part of the same model.

4.2.1.1 Persistence Area

Figure 27 shows a hierarchical breakdown of the persistence area of the metamodel. Concepts written in italics are abstract.

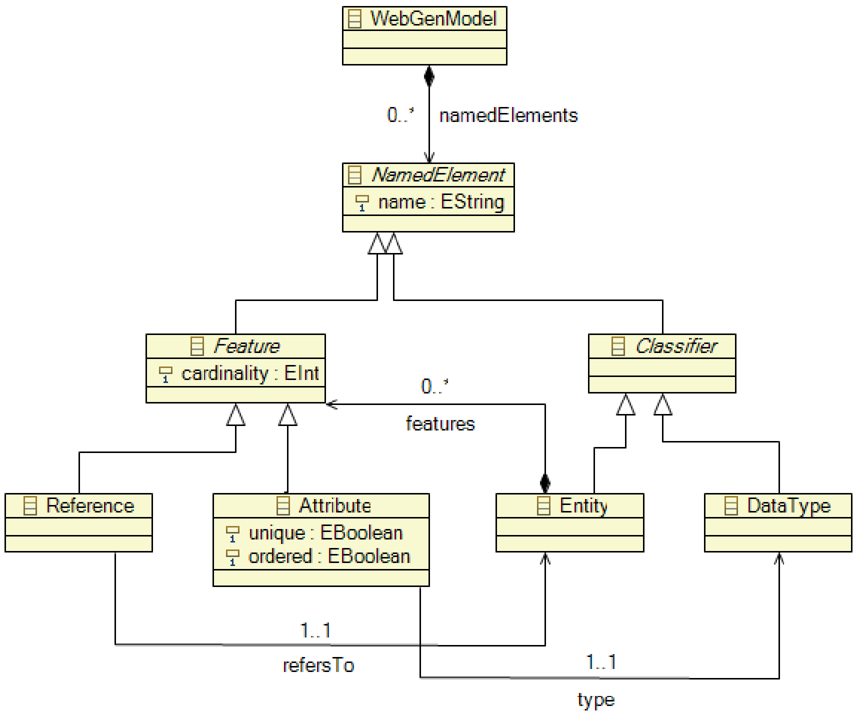


FIGURE 27: HIERARCHICAL STRUCTURE OF PERSISTENCE AREA

The persistent area of the model contains named elements that can be split into two categories: Classifiers and Features. A Classifier defines the type of an object, i.e. it describes an element that has common features or attributes. The model has two notions of Classifier: Entity and Data Type. A Feature represents a distinguishing characteristic of an item. Features have an associated cardinality that describes how the Feature relates to its containing item. A Feature is an abstract concept that embodies an Attribute or a Reference.

An Entity represents some unique, real world concept that is distinguishable from another Entity. Entities contain a collection of zero or more Features that each have a name that is unique within their containing Entity. A Data Type is a property of Attribute that defines the type of data the Attribute represents. Each Data Type has a unique name in the context of the model. Note that the model has no inbuilt types, thus Data Types within the model represent primitive types.

An Attribute is a typed property of a particular Entity. The individual elements within Attributes that represent collections i.e. those with cardinality greater than one, may be ordered or unique. A Reference describes a one-way relationship between two Entities.

4.2.1.2 Web Area

Figure 28 presents a hierarchical breakdown of the web area of the metamodel. Concepts written in italics are abstract.

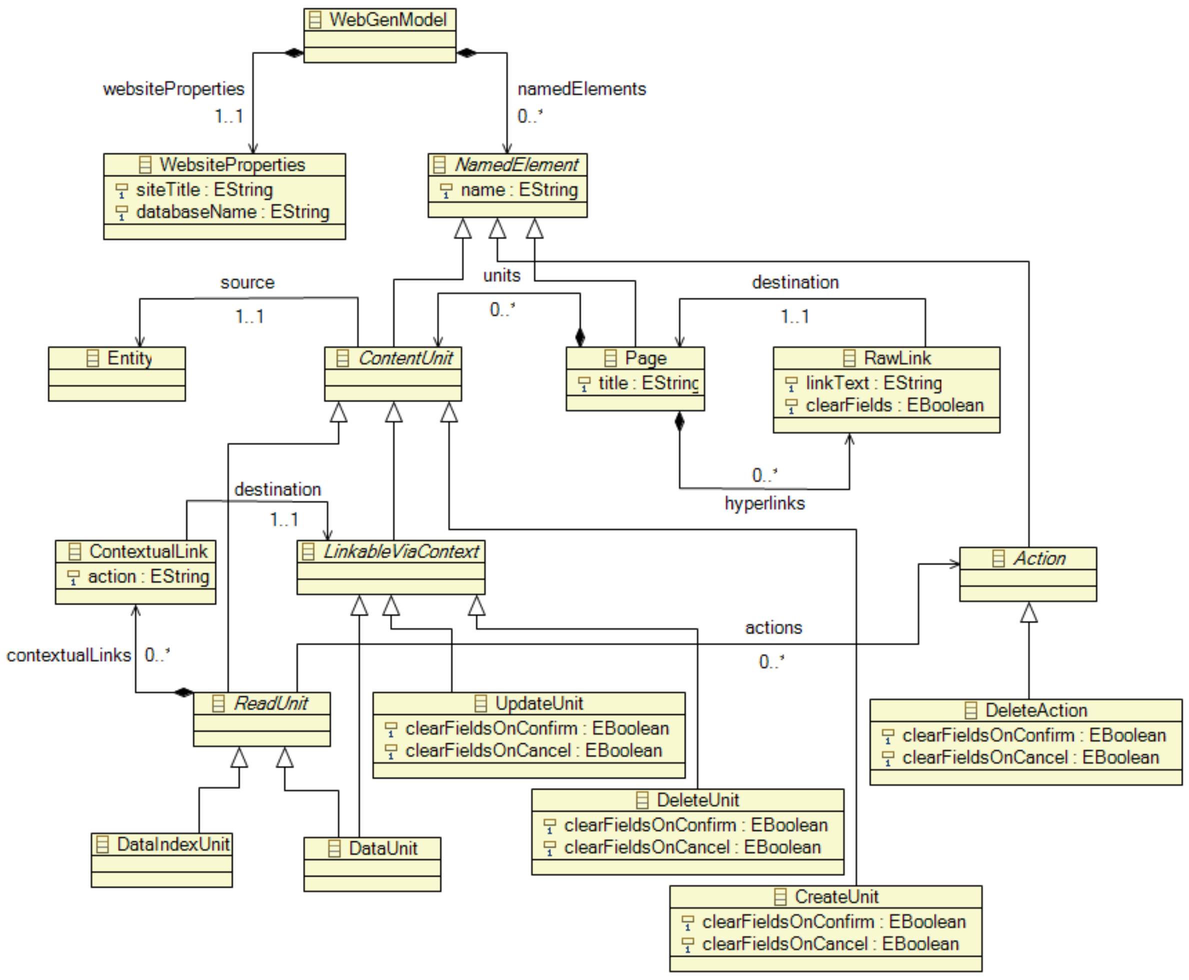


FIGURE 28: HIERARCHICAL STRUCTURE OF WEB AREA

The model's web area contains a number of named elements that represent the structure and functionality of a website: Pages, Content Units, Actions, Links and Website Properties.

A Page is an abstraction of a web page. As well as having a name, Pages also have a title and collections of zero or more Content Units and Links.

A Content Unit is an abstract concept for a unit of functionality within a page. Content Units are based on a particular source Entity and embody one of the four CRUD operations. Therefore the model contains concepts of Create Unit, Read Unit, Update Unit and Delete Unit. The model also reflects the two methods of providing Read functionality that was discussed in the website implementation (section 3.2.3). These are distinguished by Data Units and Data Index Units, each of which can have Actions associated with them. Data Units represent all details of one instance of a persistent Entity, where as Data Index Units represent a subset of details about all instances of a persistent Entity.

Data, Update and Delete Units are all instances of Content Units that are Linkable via Context. Essentially this means that Pages containing these units can be linked together by passing data between them. However, there are rules that determine how these Content Units can be linked together; Figure 29 shows these. Read Units are always the source of any linking of this form; they contain a collection of zero or more Contextual Links.

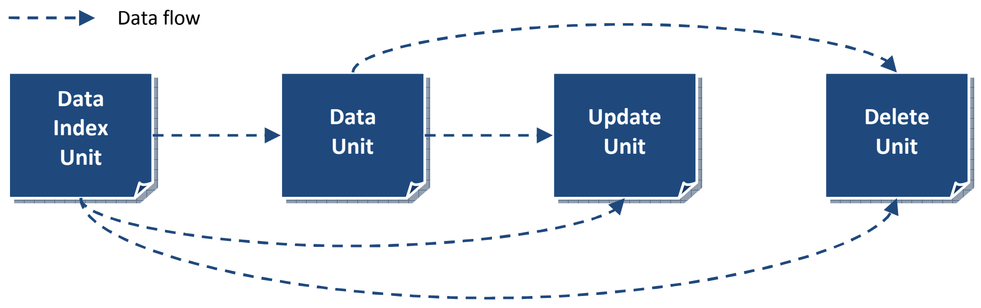


FIGURE 29: CONTEXTUAL LINK RULES BETWEEN CONTENT UNITS

The functionality behind Create, Update and Delete Units can either be performed or cancelled. For each outcome, the Unit requires a reference to a destination Page that will be loaded after the outcome is performed (N.B this has been removed from the diagram for clarity). Furthermore, they also require knowledge of whether or not data that was previously supplied to them should be cleared when the outcome is performed. Update Units also enable a selection of Features of an Entity to be updated, rather than simply all of the Features.

The concept of Actions in the model is a subtle one. Logically, an Action performs some operation on data that uses a simple confirm or cancel mechanism. Action is an abstract concept that the current model only supports one concrete instance of: Delete Action. Delete Actions have identical functionality to Delete Units, however the steps involved in performing the operation are different. Delete Actions and Delete Units reflect the two different ways of performing a deletion in the website as explained in section 3.2.3.

Links represent navigation between Pages. Links come in two forms: Raw Links and Contextual Links. A Raw Link is an abstraction of a direct hyperlink to a Page. Raw Links have a title and a reference to a destination Page. Just like some of the Content Units, Raw Links require knowledge of whether or not to clear any stored data that had been passed between pages. Contextual Links represent a data flow between Content Units that are Linkable via Context.

Finally, the model contains a concept called Website Properties. Website Properties is a singleton element that defines the overall title of the website and the name of the database schema. It is a utility concept to capture extra information that is not appropriate as attributes of any existing elements.

4.2.2 Configurable Options PIM

The tooling also makes use of another PIM; the Configurable Options model. This model is designed to enable the highlighting / flagging of model elements that represent customisable features. Note this model is not specific to the domain of data intensive websites. Customisation options could have been encoded directly into the Website PIM itself, however, the chosen approach is advantageous for two reasons. Firstly, by separating the domains of data intensive websites and customisation a separation of concerns is achieved allowing either domain model to be modified in isolation without affecting the other. Secondly, by moving the notion of configuration into a separate model, this model can be reused for any other systems that require customisation.

The structure of the model is very simple, it contains one concept; Option. An Option holds a reference to an element that can be customised. The model can therefore be viewed as a set of pointers to customisable model elements within any domain model.

In the initial iteration of the customisable tooling, only Reference elements from the Website PIM are configurable. References are eventually translated into collections of the referenced Entity. Clearly, in website technology there are numerous ways to display collections of values e.g. radio buttons, check boxes, lists. The tooling allows the developer to select the way in which these collections are displayed.

4.3 Graphical Editor

The tooling requires a graphical model editor to allow the definition of website models. This section discusses the design decisions that have been taken to ensure the requirements of the graphical editor are met.

4.3.1 Clear and Intuitive Presentation

The editor is required to have a clear and intuitive UI to ensure that the definition of website models is straightforward. Below is a list of design decisions that will be encoded into the editor's implementation.

- Coloured model elements: all model elements will have a unique colour, elements that are contained within other elements will a slightly lighter shade of colour than that of their parent. This will make it easier to distinguish types of elements and improved the overall attractiveness of the diagram.
- Appropriate use of icons: any icons that the editor uses will be meaningful and appropriate for the purpose they are intended. This will improve the clarity and ease of understanding of the diagram editor.
- Grouping related tools: the editor will logically separate the tool definition area into the following groups; Persistence, Website, Create Units, Read Units, Update Units, Delete Units and Actions. This will allow the required tool to be found easily.
- Different connection styles and colours: different types of connection on the diagram will use different styles or colours so that distinguishing between them is simple. For instance, Raw Links will use a solid line whereas Contextual Links will use a dashed line. Similarly the Confirm Action and Cancel Action connections will use green and red lines respectively.
- Use of containment: the editor should ensure that elements that are contained e.g. Attributes within Entities, are nested within their parent element. This requirement makes the diagram more intuitive.

4.3.2 Validating Model Input

In order to protect the back end generators of the tooling, the graphical editor will enable some form of validation to be performed to ensure that any website model definitions are complete. The editor will use batch validation as opposed to live validation. The definition of website models requires a series of elements and attributes to be specified in order for the model to validate. Using live validation will therefore cause many errors to be displayed during the early stages of model production; this would prove annoying for the user. In reality, the decision between batch and live validation is largely irrelevant provided that some validation of the model is performed prior to generation.

4.3.3 Minimising User Interaction

The process of transforming a model of a website into a generated implementation is accomplished in a series of steps. Where possible, these various stages should be encapsulated and hidden from the user in order to make the generation process more seamless. This will be achieved by providing three contextual menu items to perform the generation process: Generate Persistence and Logic, Generate Configurable Options and Generate Website. When clicked, these items will cause the transformations for the relevant parts of the model as indicated by their name. The separate generation of configurable options is necessary to allow the user to make decisions on the customisable features within the website. However, the decision to keep persistence and logic generation separate to website generation facilitates a separation of concerns allowing either area of the model to change without requiring a regeneration of the entire model.

4.4 Tooling Metamodels: PSMs

Before M2T transformations can take place, a website model must be transformed into a set of PSMs. This is done to ensure that the final transformations are simplified as much as possible; ideally the mapping from model to text is close to one to one. This section examines the design of the PSMs within the tooling.

4.4.1 Persistent PSM

The Persistent PSM is a metamodel that is specific to Hibernate; the main persistence layer implementation technology. It originates from the persistence area of the original Website PIM. Figure 30 shows a hierarchical breakdown of the Persistent PSM. Concepts written in italics are abstract.

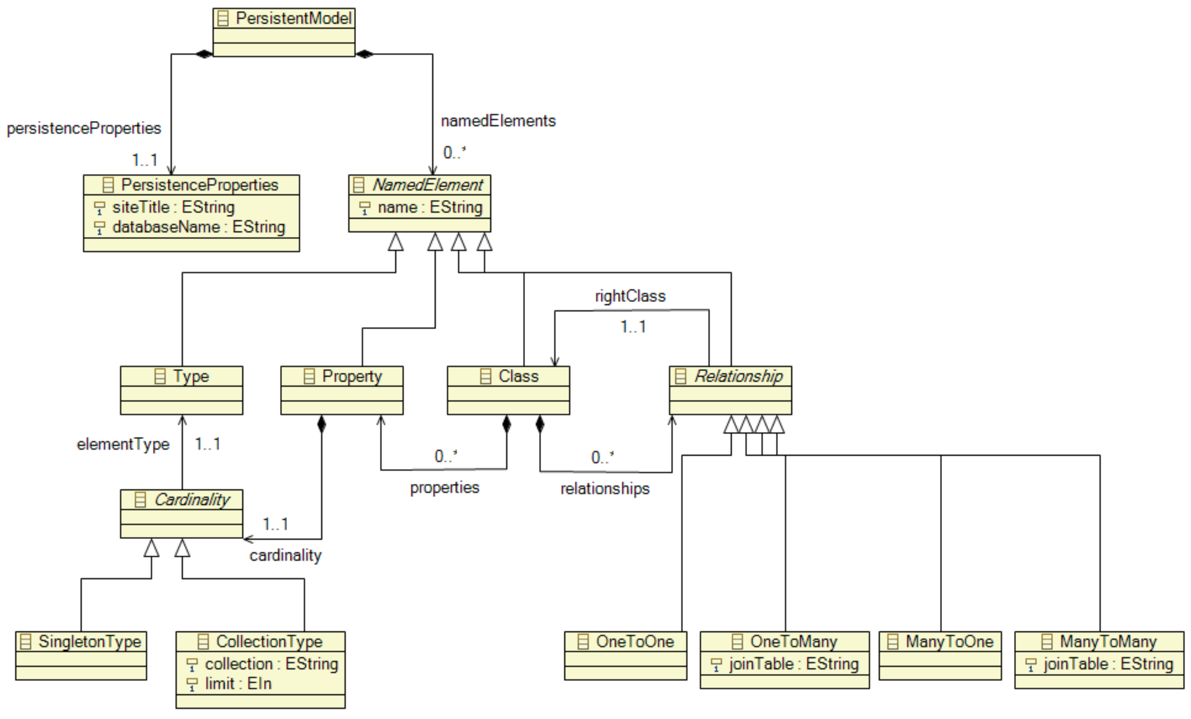


FIGURE 30: HIERARCHICAL STRUCTURE OF PERSISTENT PSM

The main element in this model is a Class. A Class represents a Java class / Object that is to be persisted. Classes contain Properties and Relationships.

Relationship is an abstract concept that contains a reference to another Class. Instances of Relationships fall into one of four categories: One-to-One, One-to-Many, Many-to-One or Many-to-Many. Here, *One* and *Many* refer to the cardinalities of the participating Classes within the Relationship. One-to-Many and Many-to-Many Relationships require the use of an extra table to produce a join.

A Property of a Class represents a primitive variable of a Java Object. A Property has a Cardinality; either Singleton (one) or Collection (greater than one). Cardinalities refer to a Type that describes.

the type of data this Property stores. A Collection is one of four types: Set, List, Bag or Ordered Set. Each collection has distinguishing *unique* and *ordered* properties on the elements it contains; see Figure 31. Collections may also have an upper bound or limit on the number of elements they can contain.

Collection	Unique	Ordered
Ordered Set	True	True
Set	True	False
List	False	True
Bag	False	False

FIGURE 31: UNIQUE AND ORDERED PROPERTIES ON COLLECTIONS

The Persistent model also contains Persistence Properties. This element has the same purpose as the Website Properties concept in the Website PIM. The Persistence Properties are the name of the website and the name of the database schema.

4.4.2 Website PSM

The Website PSM is derived from the Website PIM. The design of this model is intended to incorporate concepts that are specific to the implementation technology of the presentation layer; JSF. However, any truly JSF specific details are too close to code level and thus modelling them defeats the whole purpose for which the model was intended. As a result of this, the Website PSM only required minor changes from the original Website PIM. Firstly, the concepts from the persistence area have been updated to match those used in the Persistent PSM. This is done to simplify the M2T transformations. Furthermore, Relationships within the Website PSM now store an extra piece of information representing the developer's customisation decision over how to display the feature. By incorporating this information into the model, the entire generation of the presentation layer can take place from this model alone. The structure of the model is virtually identical to the composition of Figure 30 and Figure 28. The diagram of this model has been excluded as it is simply not possible to present it in a manageable form.

4.4.3 Configurable Choices PSM

The Configurable Choices PSM is a model intended to capture the decisions of the user for customisable features of a website. It is derived from the Configurable Options PIM. Figure 32 below presents a structural overview of this model.

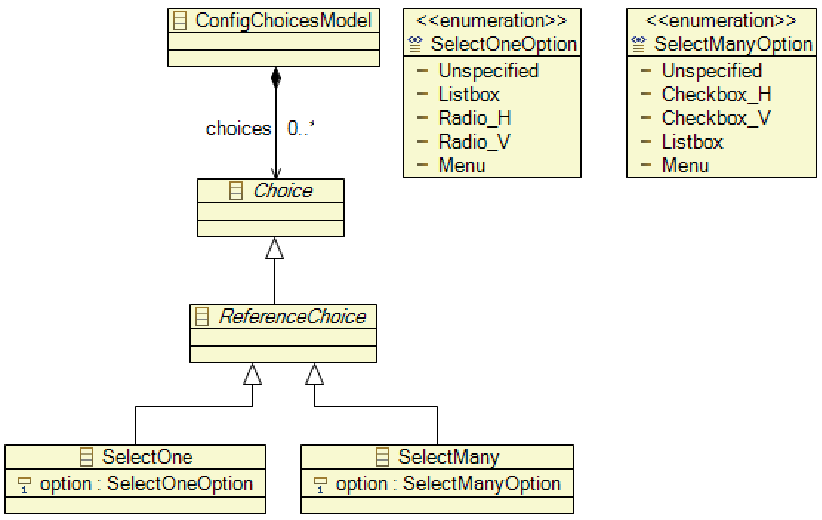


FIGURE 32: HIERARCHICAL STRUCTURE OF CONFIGURABLE CHOICES PSM

The main element in this model is the abstract concept of a Choice. A Choice represents a decision about a specific customisable website feature. Due to the current level of customisation in the tooling, there is only one instance of a Choice; a Reference Choice. Although not shown on the diagram, a Reference Choice contains a pointer to its associated Reference in the Website PIM. There are two possible types of Reference Choice; Select One and Select Many. In this context, *One* and *Many* refer to the number of items that can be selected from the UI component when it is rendered on the page. This part of the model is JSF specific; JSF defines 3 tags for displaying each type of collection. For collections where one item is to be selected JSF defines selectOneListbox, selectOneMenu and selectOneRadio. Similarly, for collections where multiple items may be selected JSF defines selectManyListbox, selectManyMenu and selectManyCheckbox tags. Furthermore, Checkbox and Radio buttons can be laid out horizontally or vertically. The model captures these options and ensures that each Select One or Select Many Reference Choice contains exactly one decision. It is also possible to use a default configuration whereby choices associated with References can be left unspecified.

4.5 Transformations

In order to map the website PIM into a set of textual artefacts a series of transformation must be performed. This section analyses the design for the tooling's M2M and M2T transformations.

4.5.1 Model-to-Model Transformations

The tooling uses four M2M transformations to convert from the PIMs into the PSMs.

4.5.1.1 Website PIM to Persistent PSM

Before the persistence and business logic layers can be generated, a transformation must be performed that maps the Website PIM into a Hibernate specific Persistent model. This transformation uses the concepts in the persistence area of the Website Generation PIM. Figure 33 below shows the mapping from elements in the Website PIM to elements in the Persistent PSM.

Website PIM Element	**Persistent PSM Element**	**Comment**
Entity	Class	Hibernate is Object-Relational therefore Classes represent persistent entities.
Attribute (Cardinality = 1)	Property (Singleton)	Attributes with cardinality of 1 map to Properties that have Singleton Cardinality.
Attribute (Cardinality > 1)	Property (Collection)	Attributes that represent collections map to Properties that have Collection Cardinality. The type of collection is based on the values of the Attribute's unique and ordered properties as described in Figure 31.
Reference (Cardinality = One-to-One)	Relationship (One-to-One)	In Hibernate, references between classes are known as Relationships.
Reference (Cardinality = One-to-Many)	Relationship (One-to-Many)	As above. Join tables are named via a concatenation of the names of two Classes involved.
Reference (Cardinality = Many-to-One)	Relationship (Many-to-One)	As above.
Reference (Cardinality = Many-to-Many)	Relationship (Many-to-Many)	As above. Join tables are named via a concatenation of the names of two Classes involved.
Data Type	Type	Hibernate refers to a notion of Type rather than Data Type, this is simply a renaming.
Website Properties	Persistence Properties	Website properties shall be known as persistence properties in the context of the Persistent PSM

FIGURE 33: WEBSITE PIM TO PERSISTENT PSM MAPPING

4.5.1.2 *Website PIM to Configurable Options PIM*

In order to enable website customisation, a transformation is required that creates an instance of the Configurable Options model that describes the customisable features of the input model. Due to current level of customisation available, this transformation is trivial. It simply maps Reference elements in the Website PIM to Option elements in the Configurable Options model, storing a pointer to the original element.

4.5.1.3 *Configurable Options PIM to Configurable Choices PSM*

Once an instance of a Configurable Options model has been created, it must then be transformed into a Configurable Choices model that allows the user to make concrete decisions on customisable features for their website. Again, due to the current level of customisation the tooling allows, the mapping for this transformation is quite simple (see Figure 34).

Configurable Options PIM Element	Configurable Choices PIM Element	Comment
Option (Element with option = Reference and Cardinality = One-to-One or Cardinality = Many-to-One)	Select One Reference Choice	Specific JSF UI components can be selected based on the cardinality of the Reference.
Option (Element with option = Reference and Cardinality = One-to-Many or Cardinality = Many-to-Many)	Select Many Reference Choice	As above.

FIGURE 34: CONFIGURABLE OPTIONS PIM TO CONFIGURABLE CHOICES PSM MAPPING

4.5.1.4 *Website PIM to Website PSM*

Before generation of the presentation layer of the website can take place, a Website PSM must be constructed that contains the details of the configured choices for the website with the original Website PIM. As was mentioned in section 4.4.2, the Website PSM contains virtually identical concepts to its PIM. Consequently, the M2M transformation that occurs is made up of the same rules from Figure 33 plus a one-to-one mapping with the remainder of the elements. The configured choice for each Reference is encoded into the Website PSM's concept of Reference to simplify artefact generation in the next stage.

4.5.2 Model-to-Text Transformations

The final stage of the generation process transforms the PSMs into textual artefacts. This generation is divided into two separate sections; Persistence & Logic and Website. Although both of the sections rely on one another, it is logical to keep to the two notions separate to enable either area of the model to be altered without needing to regenerate the entire system. Due to the skilful design of the PIMs and PSMs, the M2T transformation mappings are largely one to one. Consequently, the inclusion of the mapping details in this report is unjustified.

Chapter 5 Tool Architecture Implementation

5.1 Overview

The implementation of the tooling architecture is built from components of EMF. As was discussed in Chapter 2, one of the major disadvantages of existing website generators is that they are built upon bespoke frameworks. There is currently no implementation of a website generator on EMF; this is one of the points the tooling aims to address.

5.2 PIMs and PSMs

The concepts within the various models used in the tooling have been described in Chapter 4. The implementation of these models is a translation of their specifications into the textual modelling syntax, Emfatic (see 2.3.3.2). This process is largely straightforward and so details of the translations are ignored. The Emfatic syntax for the main models within the system: Website PIM, Persistent PSM and Website PSM are included as Appendices A, B and C respectively for the interested reader.

5.3 Graphical Editor

The graphical editor for defining website models has been implemented using GMF. This section looks at how some of the features of GMF have been used to implement the design specification set out in section 4.3. A screenshot of the finished editor is shown below in Figure . The content of the model displayed in this screenshot is not important; it is the look and feel of the editor that is most relevant.

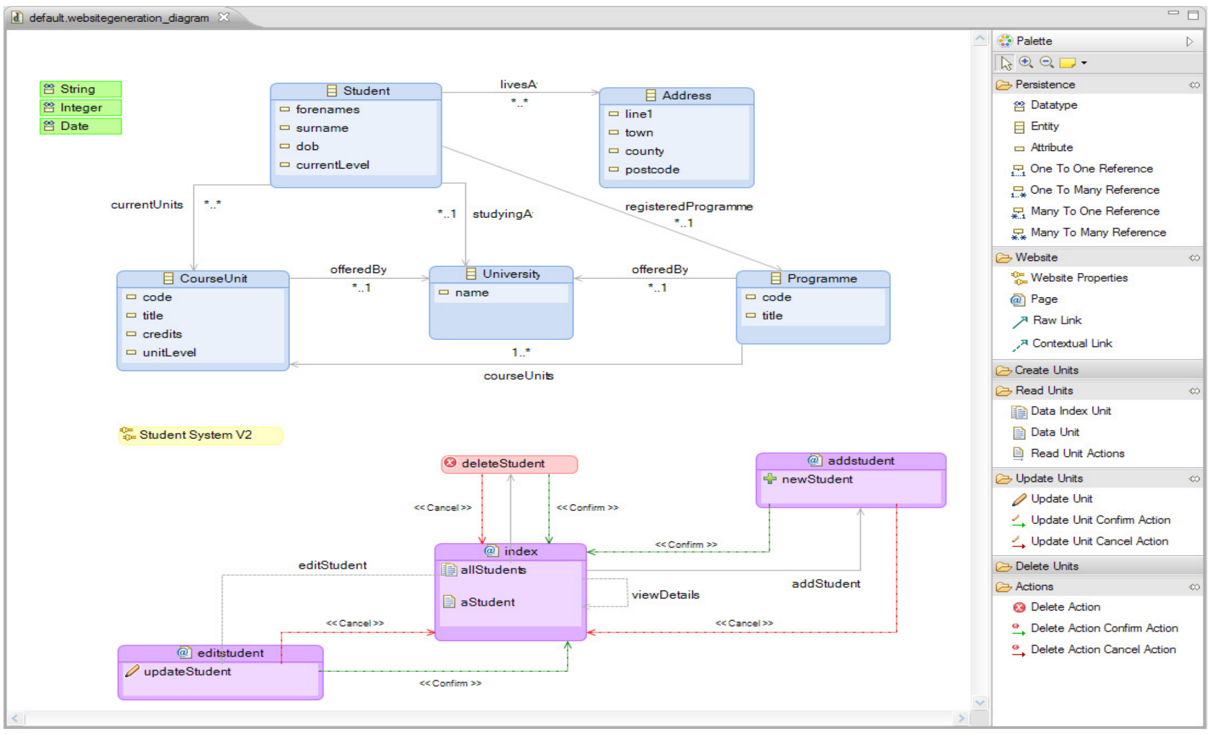


FIGURE 35: GRAPHICAL MODEL EDITOR

5.3.1 Clear and Intuitive Presentation

The look and feel of the default GMF editor has been modified by making alterations to the gmftool, gmfmap and gmfgraph models that the framework uses as input.

The gmftool model controls what appears within the Palette on the right hand side of the editor. By default, all tools within the Palette are displayed together in an unordered list. However, this implementation has been modified so that the tools are displayed in separate, collapsible groups as specified in the design of the editor. All changes to the colour and styling of the editor have been made by editing the gmfgraph model. This covers the colouring of different model elements and the styling of connectors. Many of the icons within the editor have been re-used from standard Eclipse icon packs, however, some have been drawn manually for the purpose of this tool. The gmfmap file has been modified significantly to ensure the structure of website models is intuitive. By default, GMF connects related items together with lines e.g. the Data Type of an Attribute is shown by drawing a line between the elements on the diagram. Clearly, for a model of this size, such an approach is undesirable and would result in a very hard to follow diagram. Therefore, the gmfmap has been customised to use connections to represent navigational flow only. Containment has also been introduced so that elements that are contained within other elements e.g. Content Units within Pages, are displayed that way on the diagram.

5.3.2 Validating Model Input

Use of the EMF Validation Framework has been integrated with GMF to ensure that batch validation of models can be performed. The various constraints on the model are propagated through from the original Website PIM definition that the editor is built on. A typical validation flow is presented below. Figure 36 shows part of a model definition that contains errors: the name of the Student entity contains a white space and the Address entity is duplicated.

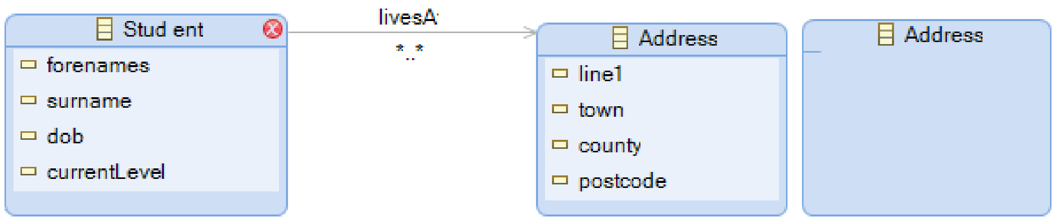
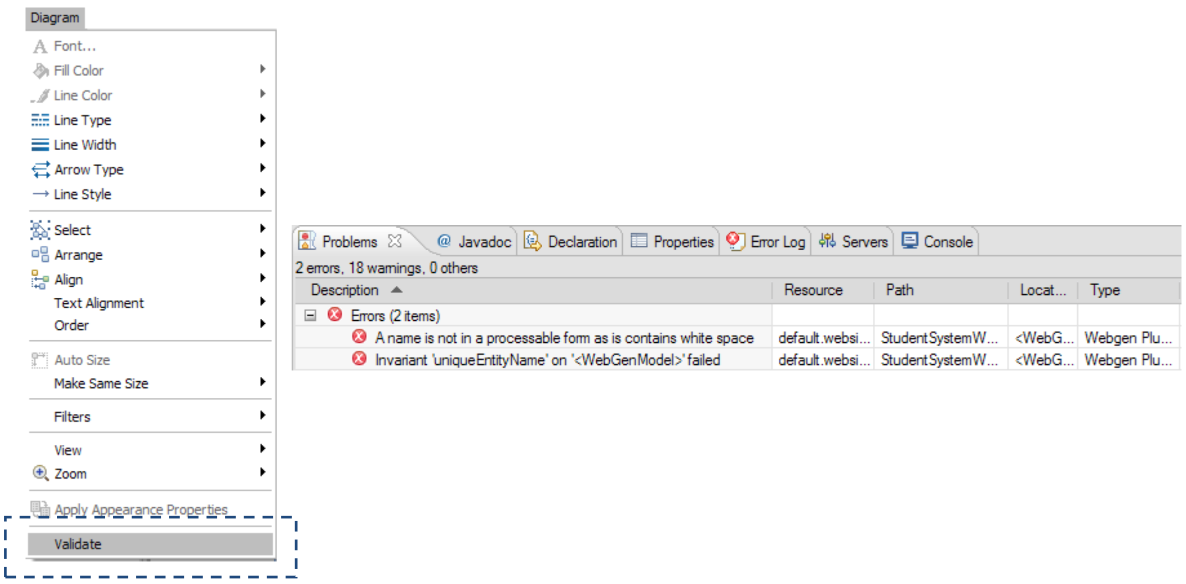


FIGURE 36: EXAMPLE MODEL SNIPPET CONTAINING VALIDATION ERRORS

The validation is performed on the model by selecting the Validate option from the Diagram menu in the editor (left of Figure 37). The resultant Problems view is shown that contains an explanation of the why the model validation failed (right of Figure 37).



```
2 errors, 18 warnings, 0 others
Description                                              Resource        Path              Locat...  Type
⊟ Errors (2 items)
    A name is not in a processable form as is contains white space   default.websi...  StudentSystemW...  <WebG...  Webgen Plu...
    Invariant 'uniqueEntityName' on '<WebGenModel>' failed           default.websi...  StudentSystemW...  <WebG...  Webgen Plu...
```

FIGURE 37: VALIDATION PROCESS OF THE GRAPHICAL EDITOR

5.3.3 Minimising User Interaction

In order to minimise the amount of user interaction required to generate the website, a series of Eclipse plug-ins have been developed to execute the required transformations programmatically. These plug-ins have then been integrated as pop up menu items that appear when particular model instances are clicked. Figure 38 shows the menu items: the left menu is displayed when the file that contains the Website PIM is selected, the right menu is displayed when the Configurable Choices PSM is selected.

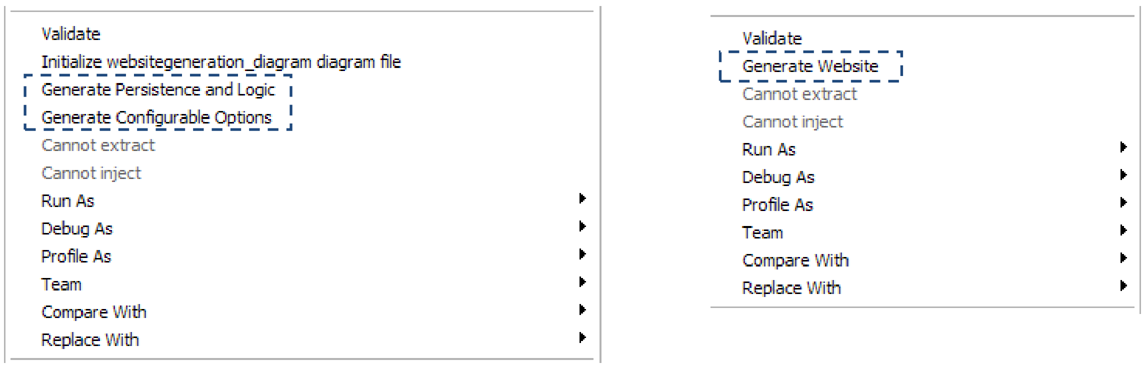


FIGURE 38: CONTEXTUAL MENUS WITHIN GRAPHICAL EDITOR

The underlying sequence of transformations that each of these plug-ins executes is shown in Figure 39. The first step generates the persistence and logic by performing an M2M transformation on the Website PIM to create an instance of the Persistent PSM. The bottom two layers of the website architecture are then generated from this model through M2T. Next, the configurable options of the website model are produced by transforming the Website PIM into an instance of the Configurable Options model. This is then transformed into an instance of the Configurable Choices model which can be edited by the user. The final step is the generation of the website itself. The Configurable Choices model instance is combined with the Website PIM to produce an instance of the Website PSM. The remaining artifacts are then produced entirely from this model via M2T.

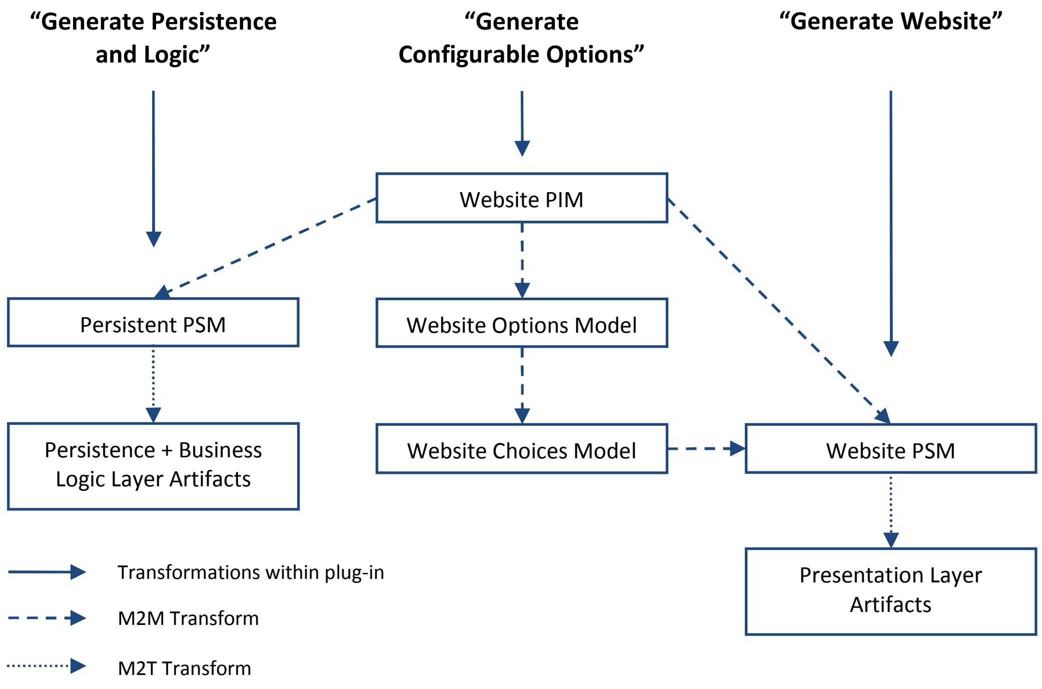

"Generate Persistence and Logic"
"Generate Configurable Options"
"Generate Website"
Website PIM
Persistent PSM
Website Options Model
Persistence + Business Logic Layer Artifacts
Website Choices Model
Website PSM
Presentation Layer Artifacts
Transformations within plug-in
M2M Transform
M2T Transform

FIGURE 39: SEQUENCE OF TRANSFORMATIONS WITHIN PLUG-INS

5.4 Model to Model Transformations

All of the M2M transformations within the tooling are written in ATL. In a similar way to the model definitions, the content of the M2M transformations are simply a translation of their specification described in section 4.5.1 into ATL.

5.5 Model to Text Transformations

The generation of textual artifacts is controlled by JET. The tooling uses a total of 28 different template files each containing approximately 200 lines of code. Separate templates are used for each artefact that is generated. During execution, JET uses XPath commands embedded within the templates to extract values from the model and insert them into the required artefact. As was mentioned in section 4.5.2 the mapping from PSMs to textual artefacts is largely one to one as the system has been designed correctly. Consequently, the low level implementation details of the templates are not worthy for inclusion in this report.

Chapter 6 Testing

Testing is important in all software development. The tooling required two separate areas of testing: firstly to establish that the correct textual artifacts were produced based on the content of the model and secondly to ensure that the generated website functioned correctly. This section provides an overview of how the testing of each area was performed.

6.1 Testing the Model Architecture

Testing the model architecture was a lengthy process. It was necessary to check that for each JET template, all possible generation branches were executed as expected. In order to accomplish this each template was first split into logical sections based on the branch conditions within the file. Small scale test models were then produced that would cause generation of a particular branch of content. The generated files were then checked manually to see if the correct content was produced.

6.1.1 Example Scenario – Hibernate Mapping

To illustrate the methods used to test the model architecture, the testing of the Hibernate mapping file generation is presented.

The Hibernate mapping template can be logically split into three sections: singleton properties, collection properties and relationships. Singleton properties are the most basic requiring the least amount of code. The model in Figure 40 shows a collection of basic user defined Data Types and an Entity containing a number of singleton Attributes.

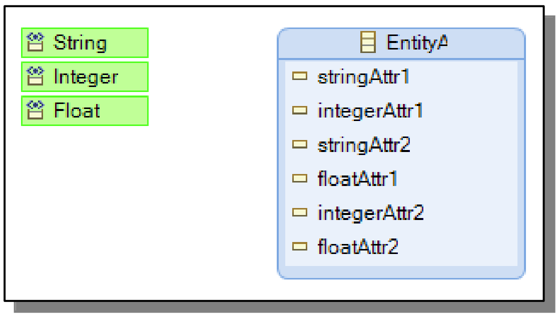


FIGURE 40: TEST MODEL - SINGLETON PROPERTIES

Collection properties have cardinalities greater than 1 and can be one of four collection types based on their ordered and unique values (see Figure 31). The test model for this class of properties, shown in Figure 41, is similar to the model to test the singleton properties. It ensures that each type of collection should be generated by varying the unique and ordered properties on the Attributes in the obvious way.

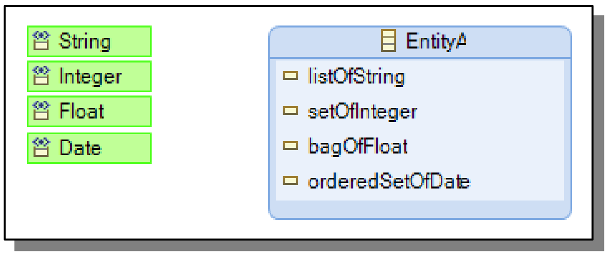


FIGURE 41: TEST MODEL - MULTITON PROPERTIES

Relationships can also be broken down into four types: One-to-One, One-to-Many, Many-to-One and Many-to-Many. The test model for the relationships is shown in Figure 42. The model shows four Entities and four relationships between them; each relationship type is tested.

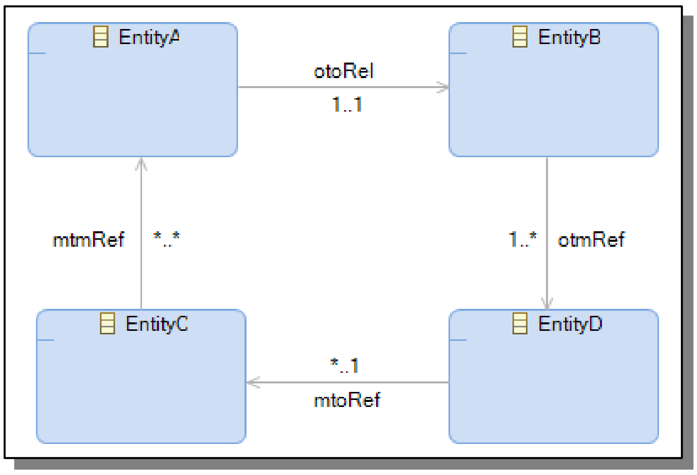


FIGURE 42: TEST MODEL - RELATIONSHIPS

Each of these models was put through the required transformations to produce a Hibernate mapping file. These files were then analysed manually to check whether the correct code had been generated. Furthermore, the SQL that Hibernate generated for each mapping file was checked to ensure the database was built in the expected way. This process was repeated as necessary when changes to the template file were made.

6.2 Testing the Generated Architecture

Testing the generated websites was very much a use case driven process performed in a similar way to the testing of the model architecture. Functionality was broken down into CRUD operations that were each tested in isolation and then combined. Website navigation was tested by physically performing the actions. Although the testing of the website was largely informal, significant effort was made to ensure it was rigorous and thorough. A better approach might have been to develop a framework to test the generated websites automatically. However, time restrictions on the project made such an objective difficult to achieve if the original requirements of the tooling were also to be met.

Chapter 7 Results

This chapter presents a high level overview of the functionality of the tooling. It aims to display the main features of the system and what they mean for website developers.

7.1 Direct Model to Website Mapping

Website models defined using the tooling's graphical editor are transformed into fully functional websites containing the exact functionality specified within the model. For a relatively simple model, such as that shown in Figure 43, generation takes less than 5 seconds and produces around 5,000 lines of code.

Figure 43 shows a simple model of a website for a student system. It records details about students, addresses, course units, programmes and universities. The website definition at the bottom of the diagram is focussed on providing CRUD functionality on students stored in the database. It shows 5 pages; one main index page containing a data index unit for students and separate pages for each of the CRUD functions. Contextual links are used to link the read and delete pages to the main index unit. A raw link is used to connect the addStudent page to the index.

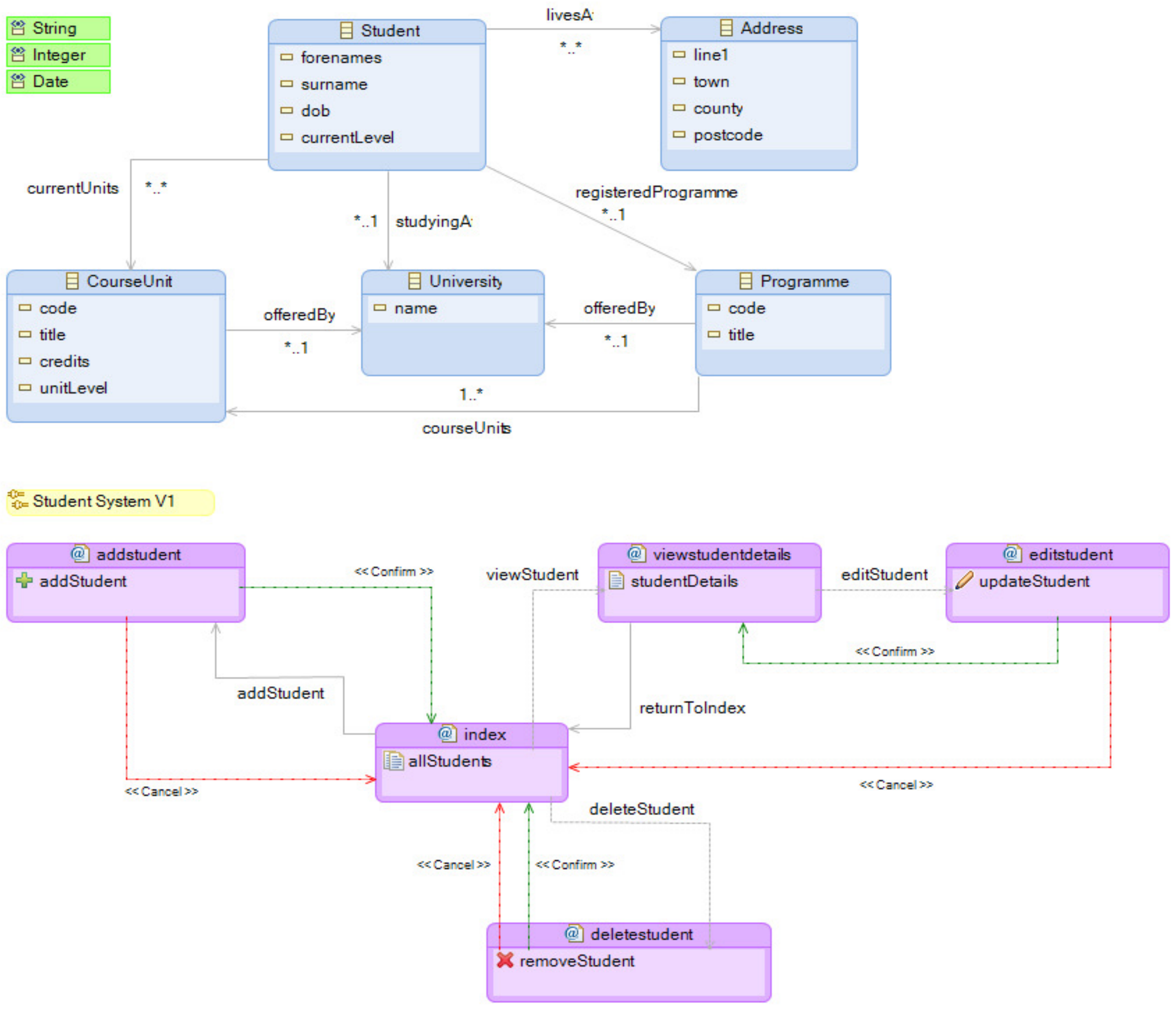
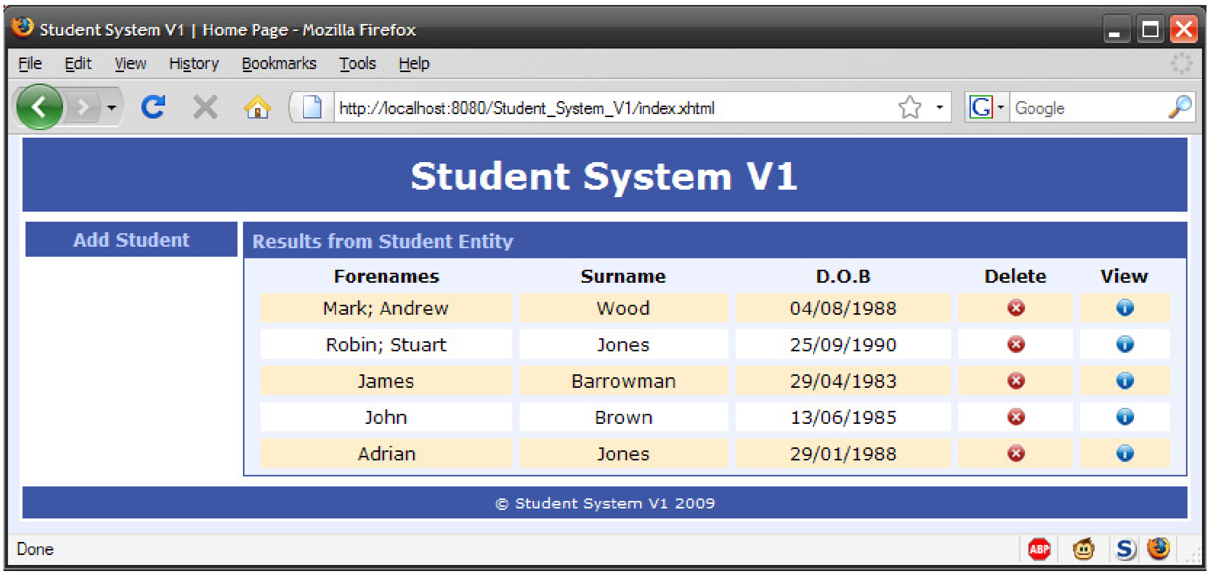


FIGURE 43: STUDENT SYSTEM V1 MODEL

The index page of the website generated from this model is shown in Figure 44. The screenshot shows the data index unit displaying all the students in the system. Next to each student record is a link to either delete or view a particular student's full details. A hyperlink is displayed down the left hand side that goes to the addStudent page. Notice how the generated website is an exact implementation of the model that is fully functional immediately after generation.


Student System V1 | Home Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/Student_System_V1/index.xhtml
Google

Student System V1

Add Student

Results from Student Entity

Forenames	Surname	D.O.B	Delete	View
Mark; Andrew	Wood	04/08/1988	⊗	ⓘ
Robin; Stuart	Jones	25/09/1990	⊗	ⓘ
James	Barrowman	29/04/1983	⊗	ⓘ
John	Brown	13/06/1985	⊗	ⓘ
Adrian	Jones	29/01/1988	⊗	ⓘ

© Student System V1 2009

Done

FIGURE 44: STUDENT SYSTEM V1 GENERATED WEBSITE

7.2 Flexibility and Configuration

Flexibility is one of the key strengths of the tooling; it aims to ensure constraint free development and avoid developers being forced into making particular design decisions. A brief analysis of how this is done is presented below.

Figure 45 shows another simple model definition of a student system. It is in fact identical in terms of functionality to the model in Figure 43. The model uses the same persistence definitions but a slightly different website definition. In this model, only three pages are specified compared to five in the previous model. Notice how different content units have been composed on the same index page for displaying a list of students followed by an individual student's full details. Delete functionality in this model is performed in-place without requiring navigation to another page. This is accomplished through the use of a Delete Action rather than Delete Unit. Functionality for adding new students and updating student details are defined in a similar way.

Figure 46 shows a screenshot of the index page for the website that was generated from this slightly different model. The page contains the two content units for displaying details of students along with the links to perform data manipulation. Clicking on the Delete icon of a particular record would display a pop up dialog requesting confirmation of the action rather than navigating to a separate page. This example demonstrates how the tooling allows the definition of logically the same functionality in multiple different ways.

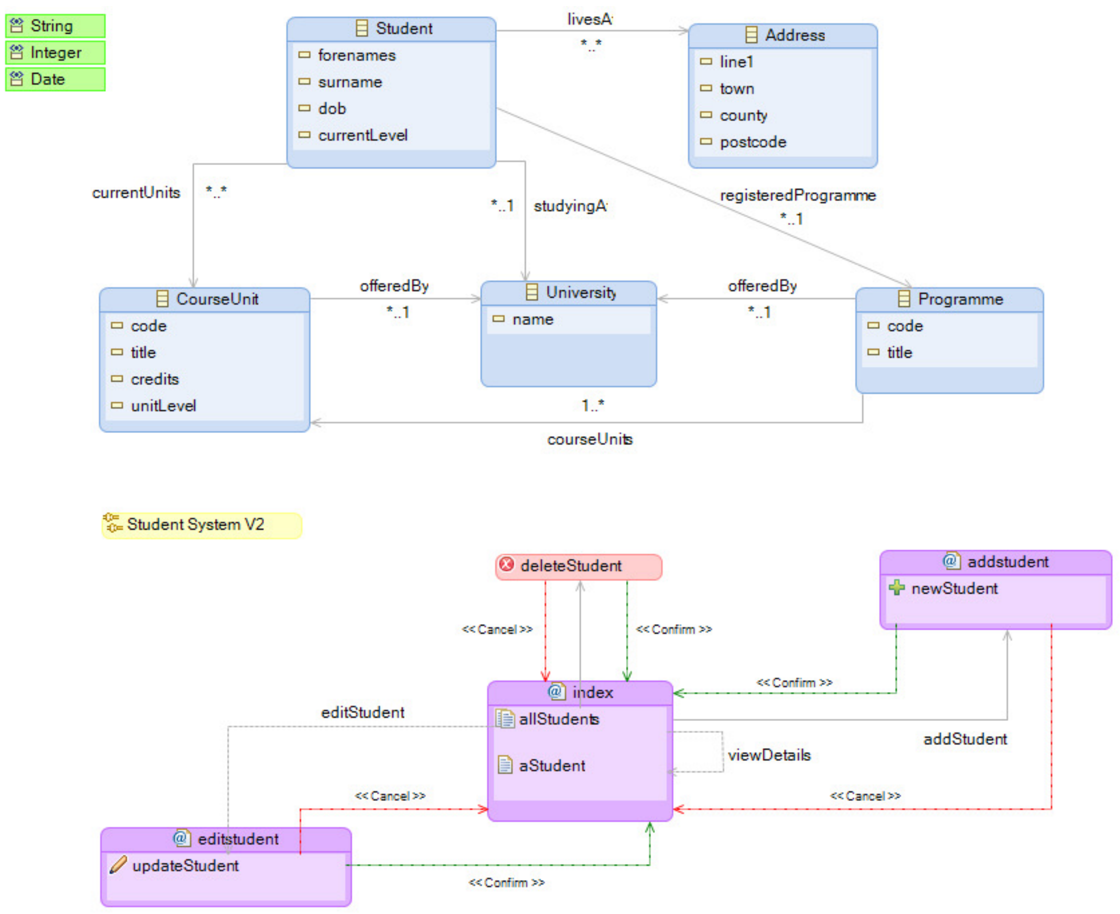
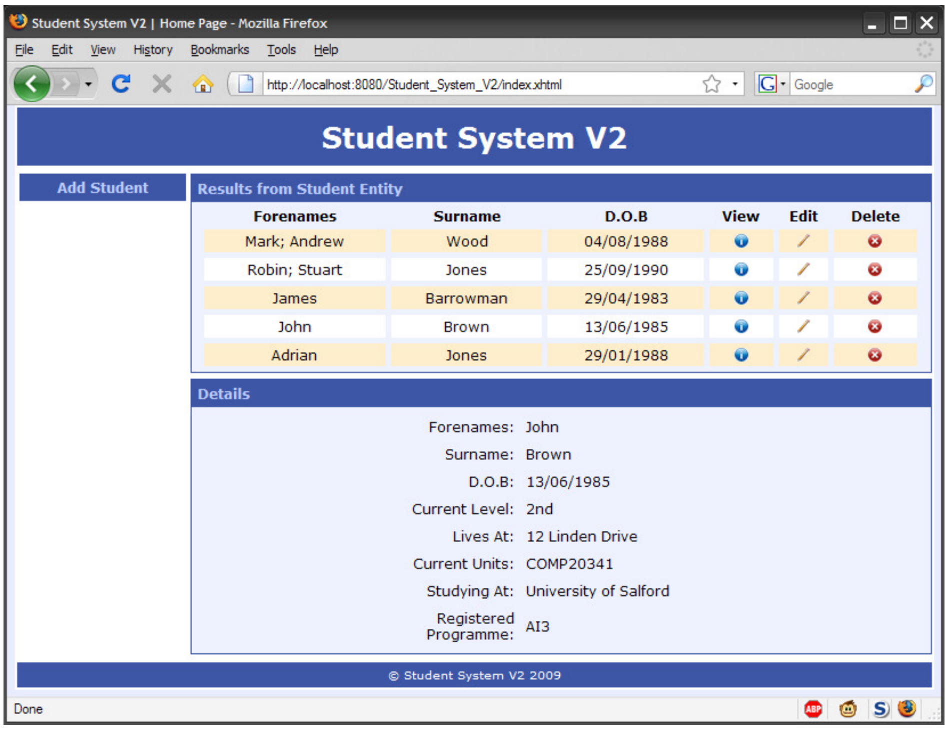


FIGURE 45: STUDENT SYSTEM V2 MODEL


Student System V2 | Home Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/Student_System_V2/index.xhtml
Google
Add Student
Done

Student System V2

Results from Student Entity

Forenames	Surname	D.O.B	View	Edit	Delete
Mark; Andrew	Wood	04/08/1988	●	✎	✖
Robin; Stuart	Jones	25/09/1990	●	✎	✖
James	Barrowman	29/04/1983	●	✎	✖
John	Brown	13/06/1985	●	✎	✖
Adrian	Jones	29/01/1988	●	✎	✖

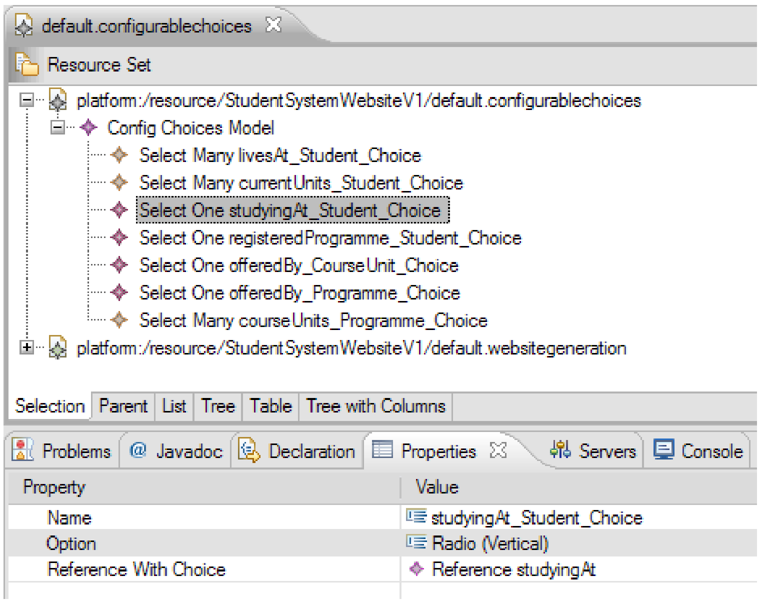
Details

Forenames:	John
Surname:	Brown
D.O.B:	13/06/1985
Current Level:	2nd
Lives At:	12 Linden Drive
Current Units:	COMP20341
Studying At:	University of Salford
Registered Programme:	AI3

© Student System V2 2009

FIGURE 46: STUDENT SYSTEM V2 WEBSITE

As well as allowing the definition of different website structures, the tooling also facilitates the customisation of individual user interface components. Currently this configurability is constrained to choices relating to how References are displayed on the screen. Figure 47 is a screenshot of the Configurable Choices model editor being used prior to generation for the model in Figure 43. It specifies that the Reference studyingAt will be rendered as a set of radio buttons stacked vertically. A further modification has been made so that the Reference livesAt is displayed as a vertical stack of check boxes. The result of this configuration is shown in the screenshot in Figure 48; the relevant customised areas are highlighted.


default.configurablechoices

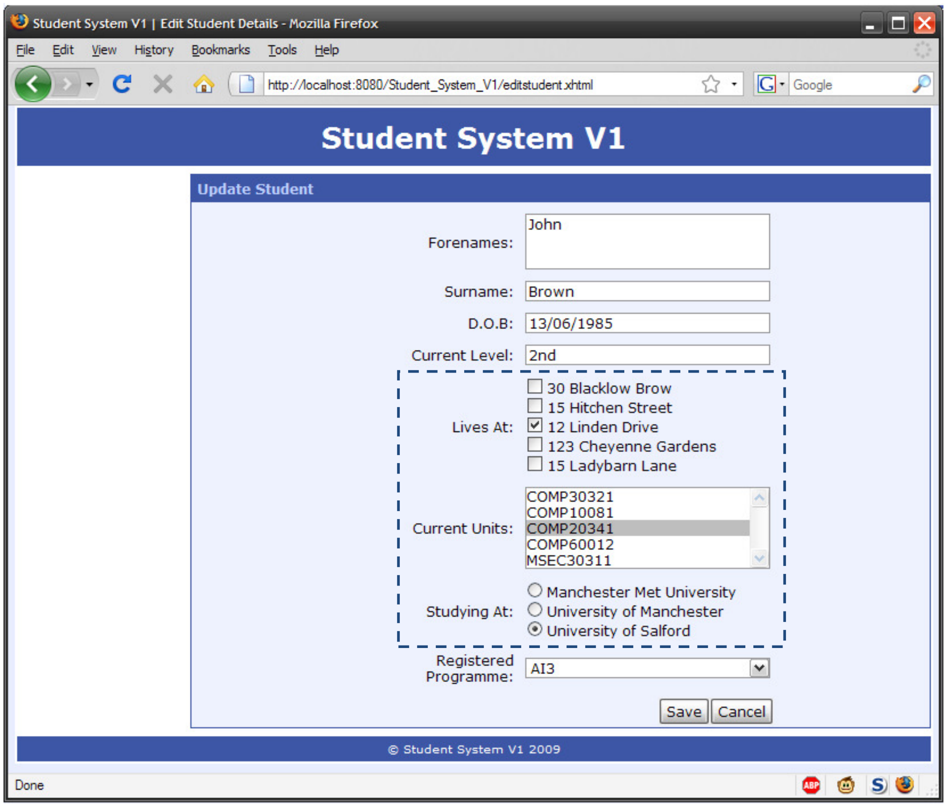
Resource Set

- platform:/resource/StudentSystemWebsiteV1/default.configurablechoices
  - Config Choices Model
    - Select Many livesAt_Student_Choice
    - Select Many currentUnits_Student_Choice
    - Select One studyingAt_Student_Choice
    - Select One registeredProgramme_Student_Choice
    - Select One offeredBy_CourseUnit_Choice
    - Select One offeredBy_Programme_Choice
    - Select Many courseUnits_Programme_Choice
- platform:/resource/StudentSystemWebsiteV1/default.websitegeneration

Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Properties Servers Console

Property	Value
Name	studyingAt_Student_Choice
Option	Radio (Vertical)
Reference With Choice	Reference studyingAt

FIGURE 47: CUSTOMISING THE WEBSITE


Student System V1 | Edit Student Details - Mozilla Firefox

File Edit View History Bookmarks Tools Help
http://localhost:8080/Student_System_V1/editstudent.xhtml
Google

Student System V1

Update Student

Forenames:	John
Surname:	Brown
D.O.B:	13/06/1985
Current Level:	2nd
Lives At:	☐ 30 Blacklow Brow ☐ 15 Hitchen Street ☑ 12 Linden Drive ☐ 123 Cheyenne Gardens ☐ 15 Ladybarn Lane
Current Units:	COMP30321 COMP10081 COMP20341 COMP60012 MSEC30311
Studying At:	○ Manchester Met University ○ University of Manchester ● University of Salford
Registered Programme:	AI3

FIGURE 48: CUSTOMISED GENERATED WEBSITE

Chapter 8 Conclusion

This chapter presents an overview of the achievements of the project in relation to the original requirements. It goes on to suggest ways in which the tooling could be extended or improved.

8.1 Achievements

The tooling generates websites capable of performing all areas of CRUD functionality. In the case of Read and Delete functions, these can be achieved in multiple ways leading to increased flexibility. The generated websites are functionally complete and can be deployed onto a server without requiring modification. The styling of the generated websites is simple, clear and functional; however, their attractiveness is largely subjective. This is one aspect of the tooling that could be improved.

The graphical editor for defining website models is simple and reasonably efficient to use. The use of UI plug-ins ensures that the required user interaction to generate the website is minimal. The editor can also validate the model definition ensuring that the system is robust. The implementation of the graphical editor was undoubtedly a success.

In the current version of the tooling, some website customisation is available that determines how certain UI elements are displayed. Unfortunately, due to time constraints, the level of configurability is not as high as it potentially could be, however the principles behind the customisation work well. Possible extensions to the customisation are discussed in section 8.2.

The websites generated by the tooling are built from some of the latest web technology and embody best practice patterns. In terms of functionality, a hand written implementation of the same website would be no better.

Overall the project has been very successful with all of the initial requirements being met to a significant extent. Given the learning curve involved with this project, this is a substantial achievement.

8.2 Further Work

The tooling was successful in meeting all of the original project requirements; it provides a solid working system for the definition of data intensive websites. As a result, it represents a good foundation for a range of extensions at different levels within the architecture.

The tool currently generates websites written in JSF. A possible extension would be to write a new set of generator templates to enable websites to be generated in different implementation technologies e.g. ASP.NET or PHP. The developer would then be able to select the web language of their choice. This extension could be appled to any of the other layers in the architecture, for example a JDO persistence layer could be used instead of Hibernate.

Currently, the customisation of configurable choices is performed through the default reflective model editor that EMF provides. A better solution would be to generate a user interface for editing the contents of this model perhaps written in a technology such as Java Swing.

The level of configurability could also be extended by adding new concepts to the configurable choices model. A range of different Facelets templates could be defined allowing the developer to select alternative layout options. Similarly, some notion of a colour scheme choice could be incorporated into the model to allow websites with different look and feel to be produced easily.

The use of property bundles for attribute names could be extended to provide support for multiple languages and internationalization.

At the lowest level, the website generation model itself could be modified to allow the definition of more sophisticated websites. For instance, adding support for authenticated websites would be a useful extension i.e. websites where certain areas are password protected.

References

1. **Netcraft Ltd.** March 2009 Archives - Netcraft. *Netcraft Ltd - Internet Research, Anti-Phishing and PCI Security Services.* [Online] http://news.netcraft.com/archives/2009/03/index.html.

2. **Microsoft.** FrontPage 2003 Help and How-to - Microsoft Office Online. *Office Online Home Page - Microsoft Office Online.* [Online] http://office.microsoft.com/en-gb/frontpage/default.aspx.

3. **Adobe.** web design software, HTML editor | Adobe Dreamweaver CS4. *Adobe.* [Online] http://www.adobe.com/products/dreamweaver/.

4. **R, Crooks.** HTML Editors vs. Authoring Tools. *Graphics Unleashed.* [Online] http://www.unleash.com/articles/homesite/.

5. **Code Generation Network.** Code Generation Network - Generators By Platform. *Code Generation Network.* [Online] http://www.codegeneration.net/generators-by-platform.php?target=12&lang=4.

6. **AndroMDA.org.** AndroMDA.org - What is AndroMDA? *AndroMDA.org.* [Online] http://galaxy.andromda.org/index.php?option=com_content&task=blogcategory&id=0&Itemid=42.

7. **Politecnico di Milano.** webml.org - Overview. *webml.org.* [Online] http://www.webml.org/webml/page3.do?ctx1=EN.

8. **Web Models s.r.l.** WebRatio - Supported Standards. *WebRatio.* [Online] http://www.webratio.com/portal/contentPage.do?seu169akey.att1=155&link=oln212.redirect.

9. **Giormov, Dimitar.** Texo - Eclipsepedia. *Eclipsepedia.* [Online] http://wiki.eclipse.org/Texo.

10. **A, Brown.** An introduction to Model Driven Architecture. *developerWorks : IBM's resource for developers and IT professionals.* [Online] http://www.ibm.com/developerworks/rational/library/3100.html.

11. **Kleppe A, Warmer J, Bast W.** *MDA Explained The Model Driven Architecture: Practice and Promise.* s.l. : Addison-Wesley, 2003. 032119442X.

12. **Object Management Group.** MDA. *Object Management Group.* [Online] http://www.omg.org/mda/index.htm.

13. **Cicchetti A, Di Ruscio D, Pierantonio A.** ATL Use Case - A Metamodel Independent Approach to Difference Representation. *Eclipse.org.* [Online] http://www.eclipse.org/m2m/atl/usecases/MMIndApproachtoDiffRep/.

14. **Object Management Group.** Object Management Group - UML. *Object Management Group.* [Online] http://www.uml.org/.

15. —. OMG's MetaObject Facility (MOF) Home Page. *Object Management Group.* [Online] http://www.omg.org/mof/.

16. —. MDA FAQ. *Object Management Group.* [Online] http://www.omg.org/mda/faq_mda.htm#uml%20role%20in%20mda.

17. —. CORBA, XML and XMI Resource Page. *Object Management Group.* [Online] http://www.omg.org/technology/xml/.

18. —. UML 2.0 OCL Specification. *Object Management Group.* [Online] www.omg.org/docs/ptc/03-10-14.pdf.

19. **Eclipse.org.** ATL Project. *Eclipse.org.* [Online] http://www.eclipse.org/m2m/atl/.

20. —. Eclipse Modeling - M2T - Home. *Eclipse.org.* [Online] http://www.eclipse.org/modeling/m2t/?project=jet.

21. **W3C.** XML Path Language (XPath). *World Wide Web Consortium - Web Standards.* [Online] http://www.w3.org/TR/xpath.

22. **Eclipse.org.** Project Summary - modeling. *Eclipse.org.* [Online] http://www.eclipse.org/projects/project_summary.php?projectid=modeling.

23. —. Project Summary - modeling.emf. *Eclipse.org.* [Online] http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emf.

24. —. The Eclipse Modeling Framework (EMF) Overview. *Eclipse documentation - Current Release.* [Online] http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html.

25. —. Eclipse Modeling - EMFT - Home. *Eclipse.org.* [Online] http://www.eclipse.org/modeling/emft/?project=emfatic.

26. **A, Carpenter.** Model-Driven Software Development - Emfatic (School of Computer Science - The University of Manchester). *Model-Driven Software Development (School of Computer Science - The University of Manchester).* [Online] http://www.cs.man.ac.uk/~andy/sectionShow.php?id=29.

27. **Eclipse.org.** Graphical Modeling Framework. *Eclipse.org.* [Online] http://www.eclipse.org/modeling/gmf/.

28. **S, Ambler.** The Object-Relational Impedance Mismatch. *Agile Data.* [Online] http://www.agiledata.org/essays/impedanceMismatch.html.

29. **The Apache Software Foundation.** Apache Struts - Welcome. *Apache Struts.* [Online] http://struts.apache.org/.

30. **Sun Microsystems.** JavaServer Faces Technology. *Sun Microsystems.* [Online] http://java.sun.com/javaee/javaserverfaces/.

31. **hibernate.org.** hibernate.org - Hibernate. *hibernate.org.* [Online] https://www.hibernate.org/.

32. **Javalobby.** JDO vs Hibernate. *Javalobby | The heart of the Java developer community.* [Online] http://www.javalobby.org/java/forums/t7836.html?start=0#81697882.

33. **Sun Microsystems.** Trail: The Reflection API (The Java Tutorials). *The Java Tutorials.* [Online] http://java.sun.com/docs/books/tutorial/reflect/index.html.

34. —. JavaServer Pages Technology - Documentation. *Sun Microsystems - Sun Developer Network (SDN).* [Online] http://java.sun.com/products/jsp/docs.html.

35. —. JavaServer Faces - JSF versus Struts. *Sun Forums.* [Online] http://forums.sun.com/thread.jspa?threadID=553126&forumID=427.

36. **java.net.** facelets: JavaServer Facelets. *java.net - The Source for Java Technology Collaboration.* [Online] https://facelets.dev.java.net/.

37. **SpringSource.org.** SpringSource.org. *SpringSource.* [Online] http://www.springsource.org/about.

38. —. Chapter 1. Introduction. *SpringSource.* [Online] http://static.springframework.org/spring/docs/2.5.x/reference/introduction.html#introduction-overview.

Appendix A – Website PIM Emfatic Implementation

```
@namespace(prefix="websitegeneration", uri="http://uk.ac.man.cs.mdsd.woodm6/WebsiteGeneration")
package WebsiteGeneration;

@inv(name=uniqueEntityName, spec="self.namedElements->select(e|e.oclIsTypeOf(Entity))->isUnique(name)")
@inv(name=uniqueDatatypeName, spec="self.namedElements->select(d|d.oclIsTypeOf(DataType))->isUnique(name)")
@inv(name=uniquePageName, spec="self.namedElements->select(p|p.oclIsTypeOf(Page))->isUnique(name)")
@inv(name=uniqueLinkName, spec="self.namedElements->select(l|l.oclIsTypeOf(Link))->isUnique(name)")
class WebGenModel {
  val NamedElement[*] namedElements;
  val WebsiteProperties[1] websiteProperties;
}

abstract class NamedElement {
  attr String[1] name;
}

// ------------------------ Persistence Area ------------------------ //
abstract class Classifier extends NamedElement {

}

@inv(name=validCardinality, spec="self.cardinality >= -3 and self.cardinality <> 0")
abstract class Feature extends NamedElement {
  attr int[1] cardinality = "1";
}

@inv(name=uniqueAttributeNameWithinEntity, spec="self.features->select(a|a.oclIsTypeOf(Attribute))-
>isUnique(name)")
@inv(name=uniqueReferenceNameWithinEntity, spec="self.features->select(r|r.oclIsTypeOf(Reference))-
>isUnique(name)")
class Entity extends Classifier {
  val Feature[*] features;
}

class DataType extends Classifier {

}

class Attribute extends Feature {
  ref DataType[1] type;
  attr boolean[1] ~unique = false;
  attr boolean[1] ~ordered = false;
}

class Reference extends Feature {
  ref Entity[1] refersTo;
}

// ------------------------ Web Area ------------------------ //
class WebsiteProperties {
  attr String[1] siteTitle;
  attr String[1] databaseName;
}

@inv(name=uniqueContentUnitNameWithinPage, spec="self.units->isUnique(name)")
class Page extends NamedElement {
  attr String[1] title;
  val ContentUnit[*] units;
  val RawLink[*] hyperlinks;
}

abstract class ContentUnit extends NamedElement {
  ref Entity[1] source;
}

abstract class LinkableViaContext extends ContentUnit {

}
```

```
abstract class ReadUnit extends ContentUnit {
  val ContextualLink[*] contextualLinks;
  ref Action[*] actions;
}

class CreateUnit extends ContentUnit {
  ref Page[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref Page[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

class DataUnit extends ReadUnit, LinkableViaContext {

}

@inv(name=validIncludedFeatures, spec="self.source.features->includesAll(self.includedFeatures)")
class DataIndexUnit extends ReadUnit {
  ref Feature[+] includedFeatures;
}

@inv(name=validIncludedFeatures, spec="self.source.features->includesAll(self.includedFeatures)")
class UpdateUnit extends LinkableViaContext {
  ref Feature[*] includedFeatures;
  ref Page[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref Page[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

class DeleteUnit extends LinkableViaContext {
  ref Page[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref Page[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

abstract class Action extends NamedElement {

}

class DeleteAction extends Action {
  ref Page[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref Page[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

abstract class Link extends NamedElement {

}

class RawLink extends Link {
  ref Page[1] destination;
  attr String[1] linkText;
  attr boolean[1] clearFields;
}

class ContextualLink extends Link {
  ref LinkableViaContext[1] destination;
  attr String[1] action;
}
```

Appendix B – Persistent PSM Emfatic Implementation

```
@namespace(prefix="persistentpsm", uri="http://uk.ac.man.cs.mdsd.woodm6/PersistentPSM")
package PersistentPSM;

class PersistentModel {
  val NamedElement[*] namedElements;
  val PersistenceProperties[1] persistenceProperties;
}

abstract class NamedElement {
  attr String[1] name;
}

class Class extends NamedElement {
  val Property[*] properties;
  val Relationship[*] relationships;
}

class Property extends NamedElement {
  val Cardinality[1] cardinality;
}

abstract class Relationship extends NamedElement {
  ref Class[1] rightClass;
}

class OneToOne extends Relationship {

}

class OneToMany extends Relationship {
  attr String[1] joinTable;
}

class ManyToOne extends Relationship {

}

class ManyToMany extends Relationship {
  attr String[1] joinTable;
}

class Type extends NamedElement {

}

abstract class Cardinality {
  ref Type[1] elementType;
}

class SingletonType extends Cardinality {

}

class CollectionType extends Cardinality {
  attr String[1] collection;
  attr int[1] limit;
}

class PersistenceProperties {
  attr String[1] siteTitle;
  attr String[1] databaseName;
}
```

Appendix C – Website PSM Emfatic Implementation

```
@namespace(prefix="websitepsm", uri="http://uk.ac.man.cs.mdsd.woodm6/WebsitePSM")
package WebsitePSM;

class WebsiteModel {
  val NamedElement[*] namedElements;
  val WebsiteProperties[1] websiteProperties;
}

abstract class NamedElement {
  attr String[1] name;
}

// ----------------------- Persistence PSM ----------------------- //
class Class extends NamedElement {
  val Property[*] properties;
  val Relationship[*] relationships;
}

class Property extends NamedElement {
  val Cardinality[1] cardinality;
}

abstract class Relationship extends NamedElement {
  ref Class[1] rightClass;
  attr String[?] configChoiceDecision;
}

class OneToOne extends Relationship {

}

class OneToMany extends Relationship {
  attr String[1] joinTable;
}

class ManyToOne extends Relationship {

}

class ManyToMany extends Relationship {
  attr String[1] joinTable;
}

class Type extends NamedElement {

}

abstract class Cardinality {
  ref Type[1] elementType;
}

class SingletonType extends Cardinality {

}

class CollectionType extends Cardinality {
  attr String[1] collection;
  attr int[1] limit;
}

// ----------------------- Enhanced Web Area ----------------------- //
class WebsiteProperties {
  attr String[1] siteTitle;
}

class WebPage extends NamedElement {
  attr String[1] title;
  val ContentUnit[*] units;
  val RawLink[*] hyperlinks;
}

abstract class ContentUnit extends NamedElement {
  ref Class[1] source;
}

abstract class LinkableViaContext extends ContentUnit {

}

abstract class ReadUnit extends ContentUnit {
  val ContextualLink[*] contextualLinks;
  ref Action[*] actions;
}

class CreateUnit extends ContentUnit {
  ref WebPage[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref WebPage[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

class EnhancedDataUnit extends ReadUnit, LinkableViaContext {
}

class EnhancedDataIndexUnit extends ReadUnit {
  ref Property[*] includedProperties;
  ref Relationship[*] includedRelationships;
}

class EnhancedUpdateUnit extends LinkableViaContext {
  ref Property[*] includedProperties;
  ref Relationship[*] includedRelationships;
  ref WebPage[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref WebPage[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

class DeleteUnit extends LinkableViaContext {
  attr String[?] filter;
  ref WebPage[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref WebPage[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

abstract class Action extends NamedElement {

}

class DeleteAction extends Action {
  ref WebPage[1] confirmAction;
  attr boolean[1] clearFieldsOnConfirm;
  ref WebPage[1] cancelAction;
  attr boolean[1] clearFieldsOnCancel;
}

abstract class Link extends NamedElement {

}

class RawLink extends Link {
  ref WebPage[1] destination;
  attr String[1] linkText;
  attr boolean[1] clearFields;
}

class ContextualLink extends Link {
  ref LinkableViaContext[1] destination;
  attr String[1] action;
}
```