A web based modelling framework

(neverfear.org edition)

Author: doug@neverfear.org

Table of Contents

Declaration of originality 5
Copyright Acknowledgement 5
Acknowledgements 6
Abstract 7
Additional Materials on the Accompanying CD 8
Glossary 9
Chapter 1: Introduction 11
  Proposed system and justification 11
Chapter 2: Literature Review 13
  Modelling techniques 13
  Modelling language levels 13
  Modelling concepts 15
Chapter 3: Design 16
  Technology review 16
    Client-side 16
    Server-side 17
  Data structures 20
    Server-side data structures 21
    The XML library file 26
    Client-side data structures 29
  User-Interface components 31
    The modelling element tray 32
    The modelling mechanism tray 32
    The modelling area 32
    The menu bar 32
    The properties window 32
    Justification for layout design 32
  Modelling mechanisms 34
Chapter 4: Implementation 36
  Tomcat installation and set up 36
  WebApp directory layout 36
  Object implementation 37
  Connector implementation 38

The problem of rectangular HTML elements 38
The problem of repositioning connectors 38
The problem of re-rendering the modelling structure during user-interaction 39
Modelling mechanisms implementation 39
Problems with GWT-EXT 39
Problems in Internet Explorer 40
Chapter 5: Testing 41
Manual testing 41
Bounds checks and validation 41
Automated testing 41
Client-side 41
Server-side 41
Chapter 6: Evaluation 43
Comparison between set goals and achieved goals 43
Implementation improvements 45
Chapter 7: Conclusions 46
Summary 46
Further work 46
User interface features 46
Meta-language improvements 46
Framework improvements 47
References 48
Appendices .... **Error! Bookmark not defined.**

Declaration of originality

Copyright Acknowledgement

Signed: Date:

Office Stamp

Acknowledgements

I would like to acknowledge BOC GmbH for the inspiration and assistance in pointing me towards useful sources of information used in this project. I would like to thank Anna Feng and Frank Giannasi of Coventry University for advising me during the creation of this report.

Abstract

Computer modelling has a lot of importance in industry. Nearly all industries that utilise any part of computing also find a use for computer based modelling. The software development sector uses computer aided design modelling tools to conceptualise software implementation, the manufacturing sector may use computer modelling to describe their production processes, the financial sector may model their business processes to optimise efficiency, and so on. Whatever sector has a use of computer modelling the core objectives are always to describe, analyse and optimise their models and better the way their organisation operates.

Each business sector may employ many different modelling ‘languages’ and each language often requires an entirely different software tool. Many firms may also want specific features of a modelling language for their purposes that existing modelling languages do not provide for. A final issue firms may have is that collaboration of modelling between multiple departments of an organisation may be difficult due to interoperability issues that arise from modelling tool heterogeneity.

Within this report I describe and present my solution for these problems. By combining web technologies with my knowledge of modelling theory I have created a flexible prototype software tool that is capable of providing a modelling service to an entire organisation and supports many different modelling languages.

Additional Materials on the Accompanying CD

On the accompanying CD there is all the source code created using this project, test cases, automated test cases resources, two loadable meta-models, a deployable Apache Tomcat webapp, and three class diagrams documenting defined classes.

Glossary

Terms used within this paper are defined below.

AJAX
: *An acronym for "Asynchronous JavaScript and XML". AJAX is a JavaScript development technique employed in web development to request and retrieve data from a web server. In the strictest sense the retrieved data is supposed to be in XML format. However often AJAX is employed to simply retrieve any data from a web server using JavaScript.*

BPM, BPMN
: *An acronym for "Business Process Modelling" and "Business Process Modelling Notation" respectively.*

CASE
: *An acronym for "Computer Aided Software Engineering".*

Connectors & Relations
: *I use connections & relations interchangeably throughout this paper but in the strictest sense the reader should think of relations as a representation of a logical relationship and connectors as a modelling element that represents a relationship.*

JSON
: An acronym for "JavaScript Object Notation". A simple set of data structures that can be represented as a string, transmitted easily and evaluated into a JavaScript object.

Line equation
: *In a 2-dimensional environment the equation for a line is $y = mx + b$ where m is the gradient of the line and b is the intersection of the y-axis when $x = 0$.*

Meta-Model, Modelling Library
: *The definition of a modelling language and all its components.*

Modeller
: *A human user that uses a modelling application to create models.*

Modelling
: *The process of representing a concept or idea in a logical and often graphical way.*

Modelling class
: *A specialised modelling element that represents the logical definition of an isolated entity in the model.*

Modelling element
: *A syntactic component of a modelling language.*

Modelling object/instance
: *An actual instance of a particular modelling class in the model.*

Modelling relation, relation class, connector type
: *A specialised modelling element that represents a logical relation between two modelling objects.*

Modelling language

A way of expressing a modelling concept that includes how elements look graphically and how they function logically.

RPC

An acronym for “Remote Procedure Call”. RPC is a client-server protocol for invoking methods on remote services and returning the results.

XML

An acronym for “Extensible Mark-up Language”. A flexible and structured human-readable data format.

Chapter 1: Introduction

This paper focuses on the concept of modelling and the abstraction of modelling languages. As a result it is useful to have a definition of general modelling. Modelling can be described as the conceptualisation of a system, process or structure in a logical and often graphical way with the objectives being expression, analysis and optimisation of the modelled concept.

Industries that utilise computing find a use for computer based modelling. Each industrial sector has its own specific objectives of modelling. As such there are many modelling languages that have been invented and developed to assist organisations in modelling their concepts. In general this means that organisations have a wide choice of available "standard" pre-defined modelling languages. It also means that often there is a wide choice in software that implements the chosen modelling language. This becomes an issue when an organisation wishes to use two or more modelling languages because often each software tool only implements a single modelling language. As a result organisations may need to purchase, license, install and maintain many different modelling applications throughout the organisation. This can lead to interoperability issues and a loss of efficiency in the organisation. In this project I aim to address this issue.

Given that organisations often have their own specific requirements of a modelling language there is a market for custom modelling language development. To develop customised modelling languages we need to abstract modelling language and introduce the concept of a meta-model.

For software firms dealing with clients who require specific language features there is a concept of customisation. This means that a software firm may develop a generic modelling tool and built-in mechanisms to customise a meta-model to the specific requirements of the client. In this project I implement such a mechanism and application programming interface for developers that need to tweak or re-engineer a modelling language.

As a summary my objectives in this project are to develop a modelling tool that:

- is capable of logically and graphically modelling a concept
- can be used with multiple modelling languages
- can be customised such that new modelling languages can be implemented or existing modelling languages may have features added or removed as per the requirements of the modeller
- can be used to describe, analyse and optimise the modelled concept
- is easy to be used as a collaborative tool between multiple departments of an organisation
- is easy to install, update and maintain throughout the organisation
- is platform independent and can thusly be used by multiple departments of an organisation even if not all departments use the same operating system

Proposed system and justification

My proposed system is a web application that can be installed on a single web server and accessed by all members of that organisation remotely. This satisfies my goals of constructing a system that's easy to maintain throughout the organisation and that can be easily used as a collaborative inter-departmental system.

Platform independence in my system will be achieved by using raw browser-loaded technologies such as JavaScript and HTML. The system should not use technologies that are delivered to users of browsers via plug-ins. Examples of such technologies that should be avoided are:

- Adobe Flash
- Microsoft Silverlight
- Java Applets

The ability to use multiple modelling languages will be implemented by separating the modelling language rules from the actual application and instead loading the language from a meta-model file or library. The meta-model will define the modelling language being used, its modelling rules, logic, the graphical representation and any relevant analyses. This also satisfies the goal that new modelling languages can be created or existing ones modified by creating or modifying the language definition within the meta-model.

Chapter 2: Literature Review

This chapter describes project relevant background knowledge which was acquired by researching the relevant subject areas and from personal experience.

Modelling techniques

"One PhD student started the effort to compile a list of process modeling techniques in use and stopped at the count of 3,000. Many of these have been developed for a specific modeling purpose."

– Jan Recker, Process Modeling in the 21st Century

This quote illustrates just how many process modelling techniques exist. What is not fully emphasised is that the number of modelling techniques for any industry application, not just process modelling is extremely vast. The reason there are so many techniques is because there are so many scenarios in which we need to express an idea or concept to somebody else in a concise and easily understood manner. A lot of failures in project teams can be attributed to miscommunication and modelled communication often plays a part of modern project team communication. It then seems likely that some failures and miscommunications are as a direct result of not understanding how the concept is modelled. This is often due to the application of an inappropriate modelling technique that is not fit for purpose (Recker 2006). When there exists no other modelling technique to appropriately express a concept or idea people create a new modelling technique that best expresses the concept. After considering how many concepts must exist and be understood by people in industry it then seems obvious as to why we have so many modelling techniques.

Due to there being a large variety of modelling techniques we find that there is an even larger variety of software tools implementing a specific modelling technique. In large organisations this can present a problem between two departments using two different modelling tools. Digital communication between these departments when using modelling can be hampered by a lack of software interfaces between the tools. Furthermore an organisation may have several languages they wish to use on a day to day basis. This presents business related problems in that this often means many different software licenses need to be purchased and maintenance is more of an issue both of which can be very costly. This leads to the idea of building a single tool using modelling concepts that can handle many modelling languages. This tool would then be adopted by the entire organisation and should fulfil the modelling needs of that organisation.

Modelling language levels

When building a modelling application such as this it is important to investigate ways within which models and modelling can be abstracted.

From my research a commonly held structure of modelling is a four-level model. Below are two diagrams describing the differences between the levels and the abstraction that occurs at each modelling level. The first is from a technical report published in 2002 by Geisler, Klar & Pons titled "Dimensions and Dichotomy in Metamodelling"

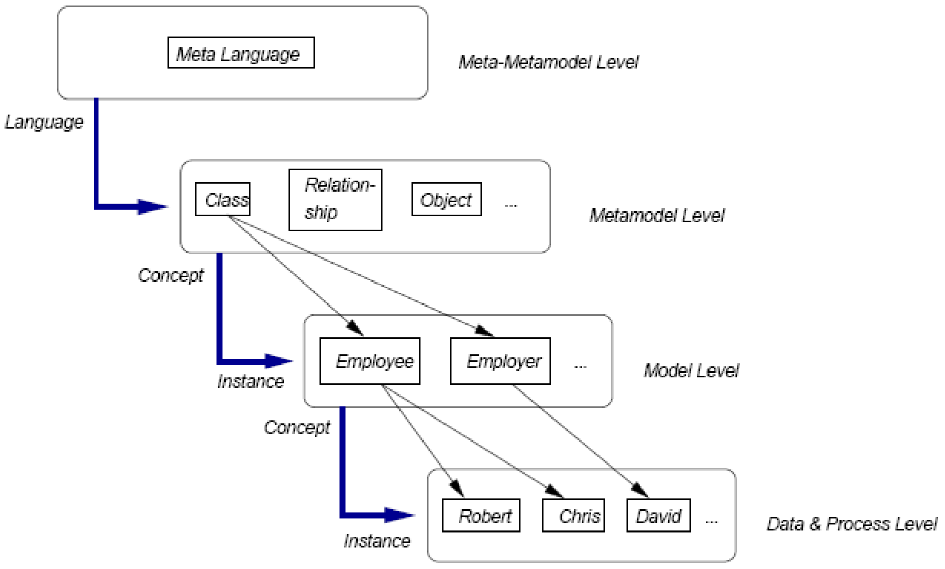


Figure 1 (Geisler, Klar & Pons 2002)

At the bottom it shows the actual model that the end users create. The next level up is the modelling language itself containing the available modelling elements. The next level up from that is the meta-modelling language that contains generic constructs that exist within any modelling language. The top level is the meta-meta-modelling language that describes the meta-language itself.

Another illustration of modelling levels comes from a paper published in 2002 title "Metamodelling platforms" authored by Kühn & Karagiannis.

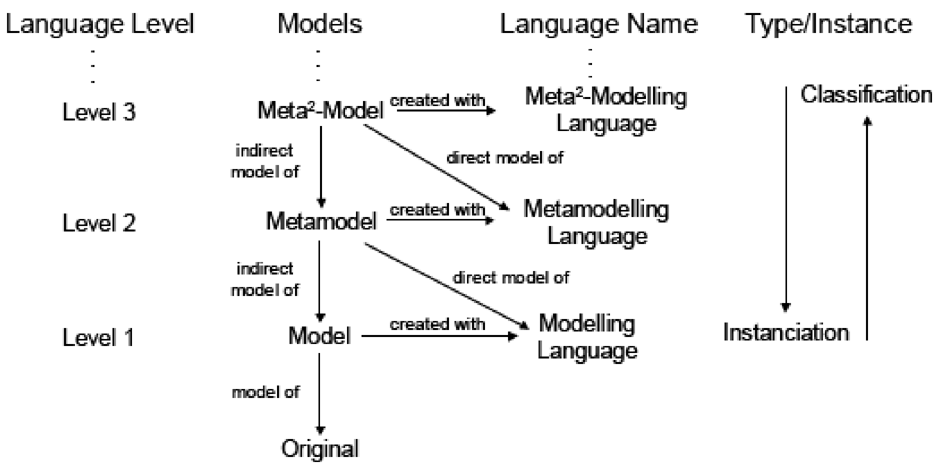


Figure 2 (Kühn & Karagiannis 2002)

This diagram illustrates further relationships between the modelling levels, including the distinction between instantiation and classification.

My application should implement the meta-modelling language in order to create modelling language. In addition it should be possible using the meta-modelling language to create a meta^2-modelling language using the idea of modelling mechanisms discussed later in this chapter.

Modelling concepts

Kühn & Karagiannis in their 2002 paper titled "Metamodelling platforms" define several components of modelling methods and modelling concepts which are relevant to my project.

They define the following modelling components:

- A modelling language which contains the building blocks of models including valid syntax, semantics and notation (which may be textual or graphical)
- A modelling procedure which describes the steps for creating a model from the modelling language
- Modelling mechanisms & algorithms which operate over the model as defined by the rules of the modelling language

For the purpose of this project I will be only focusing on the modelling language, mechanisms and algorithms as the modelling procedure is outside the scope of this project.

The modelling language defines (Kühn & Karagiannis 2002):

- A modelling syntax that defines:
  - Modelling elements
  - Modelling rules (such as multiplicity, relational logic, etc)
- Modelling semantics which describes the meaning of modelling elements
- A modelling notation which describes how modelling elements visually look

The modelling mechanisms & algorithms are used to evaluate and analysis models created using the given modelling language (Kühn & Karagiannis 2002). Kühn & Karagiannis broke down these mechanisms into three subcategories of mechanisms:

- Generic mechanisms implemented on the meta^2-model
- Specific mechanisms implemented on a specific meta-model. An example of such a mechanism could be an execution cost analysis on BP (business process) models. This analysis could not be applied to a UML (unified modelling language) model and therefore is classified as a meta-model specific mechanism
- Hybrid mechanisms which is implemented at the meta^2-model level and adapted to specific meta-models

An example of hybrid mechanism could be simple relational logic such as whether or not a particular type of relationship can exist in the modelling language. The mechanism would be implemented on the meta^2-model level but the details of which can be customised by the specific meta-model. The meta^2-model level would implement such relational consistency checks and the meta-model would define the specific rules for consistency such as whether or not a particular modelling element classification can be legally related to another modelling element classification and what sort of multiplicity that relationship has.

Chapter 3: Design

This chapter describes the planned software implementation to satisfy the project objectives. It discusses the issues that need to be considered and design decisions I have made. This chapter begins with a discussion of technologies, and procedures with a discussion of data structures that must be created, user-interface components and the implementation of modelling mechanisms.

Technology review

Client-side

As mentioned in the introduction, plug-in based technologies should be avoided. A graphical modelling application will need to be able to graphically render a model. Using raw browser technologies this leaves only a few options.

The first of which is to find a raw browser technology that can handle dynamic creation of graphics. The second option is to generate images of the current model on a server-side basis and update the image in the browser.

This second option introduces many problems relating to the asynchronous nature of client-server web architectures, specifically that highly interactive graphical applications tend not to be successfully implemented by constantly posting rendering requests over a non-instantaneous communication channel and waiting for responses. Such an application may experience lag and experience rendering delays. Graphical update requests may even fail entirely leaving the user with a broken application.

Previously the first option would have been very difficult to implement but with the recent introduction to all major browsers of a new JavaScript element called the canvas element (also known as the HTMLCanvasElement), the first option is made significantly more appealing. The canvas element is a HTML element that is implemented in the browser as an image element but where the graphical contents of the image are dynamically drawn by JavaScript. This is the option I've chosen but because raw-browser technologies still have limited graphics capabilities I will need to build a code base to handle rendering the basic constructs of modelling languages such as arrows and connectors.

The next issue is that of client-side meta-model storage. The client could make requests to the meta-model services for every occasion when modelling rules are required or the client could store the rules in a simple form locally. Since the client could potentially need to make several requests every second that the user is doing something the latter option should be taken.

The question of how to store the modelling language definition (the meta-model) has a few options also. I could store the definition as XML, as a JSON string or I could instruct the server-side components to generate a JavaScript object hierarchy that translates the server-side data structures identically and methods. I have chosen to use server-side generated JSON to represent the meta-model. The reason for my choice is that XML is very heavy and requires extra JavaScript code to parse the XML document and generating JavaScript code to create JavaScript objects to duplicate the server-side data structures is a too complex solution. Once evaluated, JSON strings are instantly translated into simple JavaScript objects. An example of a JSON string being evaluated and then accessing an element of the resulting data structure is given below.

```
var json = {"name": "Bob", "childTest": {"name": "Alice"}};
alert(json.name + " has a child named " + json.childTest.name);
```

This should open a popup dialog with the text "Bob has a child named Alice". As you can see from the example I can treat JSON created objects exactly as any other JavaScript object.

A final issue is that of actual user-interface implementation. There are two options I'm considering for this issue. One of these options is to implement the interface and UI interactivity by hand using a combination of HTML and JavaScript. The other option is to implement the UI using a web toolkit such as GWT (Google Web Toolkit) or GWT-EXT (Google Web Toolkit with ExtJS extensions). GWT and GWT-EXT is a framework for developing browser interfaces in Java. The Java is then compiled to a collection of HTML and JavaScript files rather than compiling to a Java class file. Using this option would result in the user-interface implementation being entirely written in Java. My original choice for this was to implement the UI in GWT-EXT but I changed my design after encountering some implementation problems (See *Chapter 4: Implementation* for more information). So I choose to implement the UI using HTML and JavaScript.

Server-side

The application will also need a server side. This will be used to store and load the meta-model and provide each client application (the client-side browser application) about the rules, logic and notations of the modelling language. This comes in two parts, namely a web server and a dynamic server side technology (such as a server-side programming language). Each of these parts is closely related as certain web servers only provide certain dynamic technologies.

Some of my options for this component are:

- The Apache Web Server which provides a lot of CGI based modular dynamic technologies, such as:
    - PHP
    - Python
    - Perl
- The Apache Tomcat Server which is a Java based web server that provides Java related technologies such as:
    - Java Servlets which can be thought of as very flexible web services
    - Java Server Pages which are dynamically loaded scripted pages using the Java language and grammar.

- The Internet Information Server which is a Microsoft based web server that utilises Windows technologies such as:
    - The .NET framework

Given that one of my objectives is platform independence for both the client and the server components the IIS option can be excluded. Out of the remaining two distributions of the apache server it comes down largely to the technologies they provide me. I have opted to go for the Apache Tomcat server because of the rich API and the combination of servlet and server page technologies. As a result a lot of my implementation will revolve around Java.

The next technology to consider is how to store the library. The meta-model will especially be structured data that contains the modelling language definition. This can be divided further into two categories of structured data storage:

- Databases. Examples of databases:
    - Microsoft SQL databases
    - MySQL databases
    - Oracle databases
    - PostgreSQL databases
- Structured data files. Examples of my options structured data files:
    - XML
    - JSON
    - Custom developed raw file structure

My choice in this category was to implement the meta-model with XML. The reason for this is that one of my goals is to allow customisation of the meta-model to implement or modify new or existing language features. As such if I were to implement this as a database I would also require a customisation tool to allow the developers to access the language definition in the database and make changes. Whereas with XML developers can open the XML file in any standard XML editor and customise the definition by treating it as a XML configuration file.

Creating my own XML parser would be long-winded as such I investigated Java based APIs for XML parsing and found that there are two major standard XML parsing APIs and very many minor XML parsing APIs (Harold, 2002). I decided to only consider the two major standard XML parsers because I only need general XML parsing to transform the XML data into usable Java data structures. These two APIs are named:

- SAX (Simple API for XML)
- DOM (Document Object Model)

The main differences are that SAX parses a XML document as it is read from an input stream and triggers event handlers to do the actual processing where as DOM parses the XML document into a DOM (document object model) tree and then returns the result back to the calling application. Harold in his 2002 book titled “Processing XML with Java” sums SAX up as follows:

"This makes SAX very fast and very memory efficient (since it doesn't have to store the entire document in memory). However, SAX programs can be harder to design and code because you normally need to develop your own data structures to hold the content from the document."

In the same publication Harold sums DOM up as follows:

"This makes DOM much more convenient when random access to widely separated parts of the original document is required. However, it is quite memory intensive compared to SAX, and not nearly as well suited to streaming applications."

Since I already planned to develop my own data structures and because of the low memory foot print of SAX, for this project I employ SAX as my XML parser.

Data structures

The issue of the modelling library data structures needs to be addressed. The values when designing these data structures are efficiency of access and separation of logically distinct concepts. I choose to implement my data structures as a series of objects orientated data structures. By doing this I can abstract the physical storage structures from the conceptual elements of the data structures.

The first step is to identify the components of a modelling language. From the literature review I can identify:

- A meta-model that encapsulates the complete modelling definition
- Modelling elements that represents the syntactic building blocks of the modelling language
- Modelling algorithms or analyses

Modelling elements can further be specialised into objects classes and object relationships (Geisler, Klar & Pons 2002). There is a distinction between the two in this project.

Server-side data structures

The UML overview for the server-side data structures looks like this:

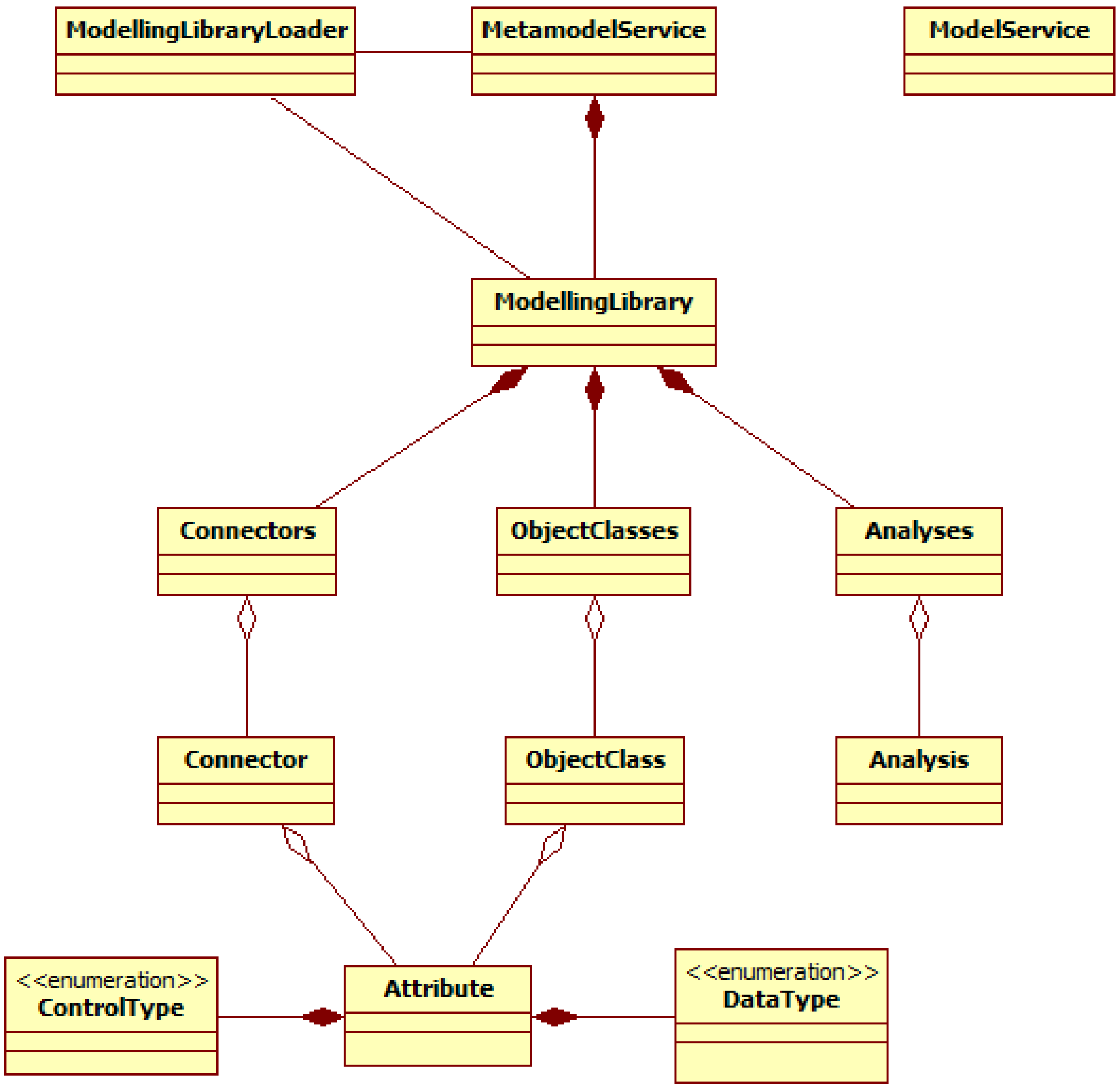
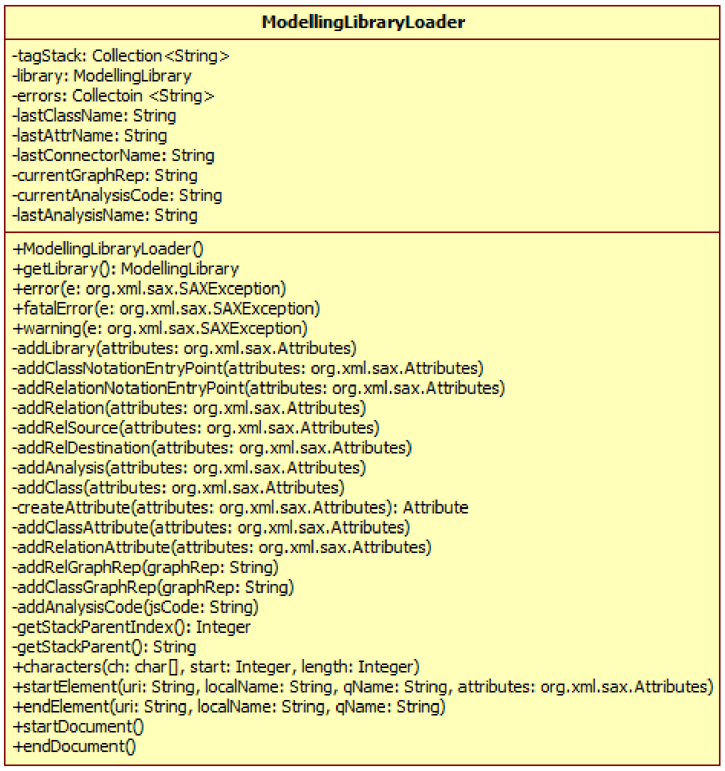


Figure 3

A detailed class diagram of the server objects can be found in appendix A. It is important to note that the ModelService is an independent service that is responsible for model file management and doesn't need to know about the meta-model. The remaining classes are all part of the MetamodelService.

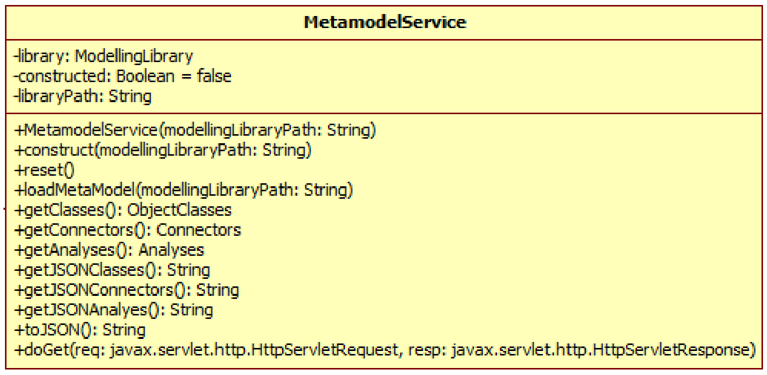
The ModellingLibraryLoader class



```
ModellingLibraryLoader
-tagStack: Collection<String>
-library: ModellingLibrary
-errors: Collectoin <String>
-lastClassName: String
-lastAttrName: String
-lastConnectorName: String
-currentGraphRep: String
-currentAnalysisCode: String
-lastAnalysisName: String
+ModellingLibraryLoader()
+getLibrary(): ModellingLibrary
+error(e: org.xml.sax.SAXException)
+fatalError(e: org.xml.sax.SAXException)
+warning(e: org.xml.sax.SAXException)
-addLibrary(attributes: org.xml.sax.Attributes)
-addClassNotationEntryPoint(attributes: org.xml.sax.Attributes)
-addRelationNotationEntryPoint(attributes: org.xml.sax.Attributes)
-addRelation(attributes: org.xml.sax.Attributes)
-addRelSource(attributes: org.xml.sax.Attributes)
-addRelDestination(attributes: org.xml.sax.Attributes)
-addAnalysis(attributes: org.xml.sax.Attributes)
-addClass(attributes: org.xml.sax.Attributes)
-createAttribute(attributes: org.xml.sax.Attributes): Attribute
-addClassAttribute(attributes: org.xml.sax.Attributes)
-addRelationAttribute(attributes: org.xml.sax.Attributes)
-addRelGraphRep(graphRep: String)
-addClassGraphRep(graphRep: String)
-addAnalysisCode(jsCode: String)
-getStackParentIndex(): Integer
-getStackParent(): String
+characters(ch: char[], start: Integer, length: Integer)
+startElement(uri: String, localName: String, qName: String, attributes: org.xml.sax.Attributes)
+endElement(uri: String, localName: String, qName: String)
+startDocument()
+endDocument()
```

This class can be thought of a load adaptor utilised by MetamodelService. Its job is to load and construct a ModellingLibrary instance containing the meta-model definition and pass it to the MetamodelService. It does this using the SAX Java XML parser and inheriting the org.xml.sax.helpers.DefaultHandler class. This allows the loader to be used as a parse handler for the XML document being read by SAX. Once the SAX parser has finished then the resulting meta-model can be retrieved by calling getLibrary().

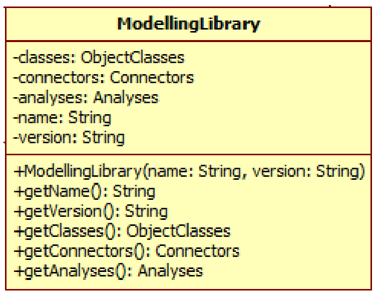
The MetamodelService class



```
MetamodelService
-library: ModellingLibrary
-constructed: Boolean = false
-libraryPath: String
+MetamodelService(modellingLibraryPath: String)
+construct(modellingLibraryPath: String)
+reset()
+loadMetaModel(modellingLibraryPath: String)
+getClasses(): ObjectClasses
+getConnectors(): Connectors
+getAnalyses(): Analyses
+getJSONClasses(): String
+getJSONConnectors(): String
+getJSONAnalyes(): String
+toJSON(): String
+doGet(req: javax.servlet.http.HttpServletRequest, resp: javax.servlet.http.HttpServletResponse)
```

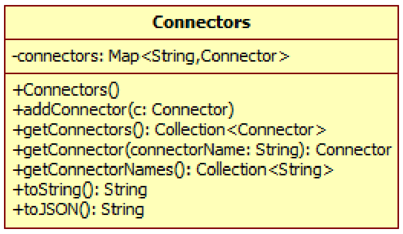
This class implements a static service that once initialised stores the loaded meta-model and dispatches JSON format strings either as a direct method call by JSP or by use of the GET http method.

The ModellingLibrary class


```
ModellingLibrary
-classes: ObjectClasses
-connectors: Connectors
-analyses: Analyses
-name: String
-version: String
+ModellingLibrary(name: String, version: String)
+getName(): String
+getVersion(): String
+getClasses(): ObjectClasses
+getConnectors(): Connectors
+getAnalyses(): Analyses
```

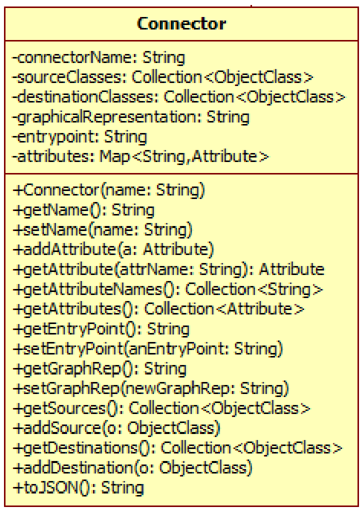
This represents the rules and definitions of a single modelling language composed of connectors (also known as relations), classes and analyses.

The Connectors class


```
Connectors
-connectors: Map<String,Connector>
+Connectors()
+addConnector(c: Connector)
+getConnectors(): Collection<Connector>
+getConnector(connectorName: String): Connector
+getConnectorNames(): Collection<String>
+toString(): String
+toJSON(): String
```

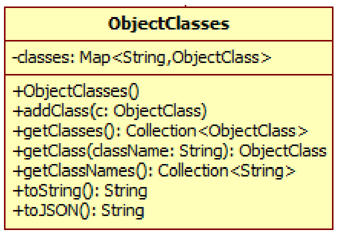
This class represents a collection of Connector instances and provides simple searching methods.

The Connector class


```
Connector
-connectorName: String
-sourceClasses: Collection<ObjectClass>
-destinationClasses: Collection<ObjectClass>
-graphicalRepresentation: String
-entrypoint: String
-attributes: Map<String,Attribute>
+Connector(name: String)
+getName(): String
+setName(name: String)
+addAttribute(a: Attribute)
+getAttribute(attrName: String): Attribute
+getAttributeNames(): Collection<String>
+getAttributes(): Collection<Attribute>
+getEntryPoint(): String
+setEntryPoint(anEntryPoint: String)
+getGraphRep(): String
+setGraphRep(newGraphRep: String)
+getSources(): Collection<ObjectClass>
+addSource(o: ObjectClass)
+getDestinations(): Collection<ObjectClass>
+addDestination(o: ObjectClass)
+toJSON(): String
```

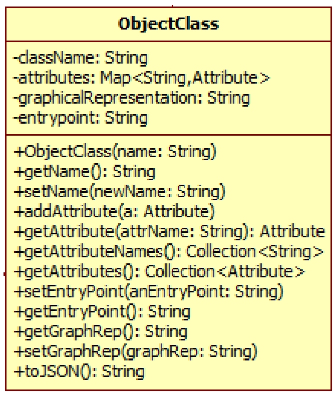
This class represents a single connector or relation type. It contains notation information used to generate the graphical representation on the modelling area at runtime. It also contains rule logic concerning what source and destination classes a connector may legally connect to.

The ObjectClasses class


ObjectClasses
-classes: Map<String,ObjectClass>
+ObjectClasses()
+addClass(c: ObjectClass)
+getClasses(): Collection<ObjectClass>
+getClass(className: String): ObjectClass
+getClassNames(): Collection<String>
+toString(): String
+toJSON(): String

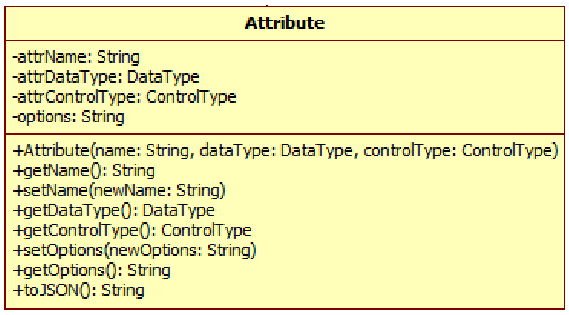
This class represents a collection of ObjectClass instances and provides simple searching methods.

The ObjectClass class


ObjectClass
-className: String
-attributes: Map<String,Attribute>
-graphicalRepresentation: String
-entrypoint: String
+ObjectClass(name: String)
+getName(): String
+setName(newName: String)
+addAttribute(a: Attribute)
+getAttribute(attrName: String): Attribute
+getAttributeNames(): Collection<String>
+getAttributes(): Collection<Attribute>
+setEntryPoint(anEntryPoint: String)
+getEntryPoint(): String
+getGraphRep(): String
+setGraphRep(graphRep: String)
+toJSON(): String

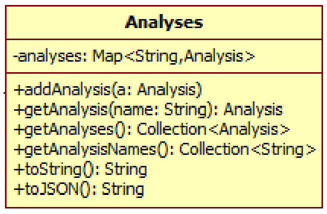
This class represents a single modelling class. It contains notation information used to generate the graphical representation on the modelling area at runtime. It also contains a list of all valid descriptors (attributes) for the element.

The Attribute class


Attribute
-attrName: String
-attrDataType: DataType
-attrControlType: ControlType
-options: String
+Attribute(name: String, dataType: DataType, controlType: ControlType)
+getName(): String
+setName(newName: String)
+getDataType(): DataType
+getControlType(): ControlType
+setOptions(newOptions: String)
+getOptions(): String
+toJSON(): String

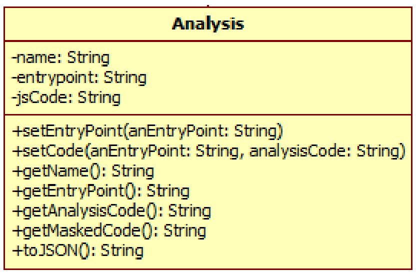
This class represents an attribute or descriptor of an element. Each attribute has a name, a control type (text input box, select box, radio field, etc) and a data type which is used for input validation. The options field is a comma delimited string of options used as the possible options for select boxes and radio fields.

The Analyses class


Analyses
-analyses: Map<String,Analysis>
+addAnalysis(a: Analysis)
+getAnalysis(name: String): Analysis
+getAnalyses(): Collection<Analysis>
+getAnalysisNames(): Collection<String>
+toString(): String
+toJSON(): String

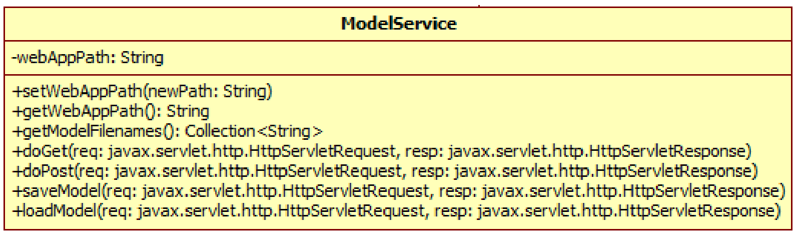
This class represents a collection of Analysis instances and provides simple searching methods.

The Analysis class



This class stores all relevant information for each defined analysis. This includes a name, the JavaScript code and the callable JavaScript entry point into that code. A discussion on how analyses and modelling mechanisms will function in this project is discussed later.

The ModelService class


ModelService
-webAppPath: String
+setWebAppPath(newPath: String)
+getWebAppPath(): String
+getModelFilenames(): Collection<String>
+doGet(req: javax.servlet.http.HttpServletRequest, resp: javax.servlet.http.HttpServletResponse)
+doPost(req: javax.servlet.http.HttpServletRequest, resp: javax.servlet.http.HttpServletResponse)
+saveModel(req: javax.servlet.http.HttpServletRequest, resp: javax.servlet.http.HttpServletResponse)
+loadModel(req: javax.servlet.http.HttpServletRequest, resp: javax.servlet.http.HttpServletResponse)

This class represents is the persistency layer of my system. It controls the storage and retrieval of saved models. It is a service class, implements HttpServlet and supports two HTTP request methods "POST" and "GET".

The POST method is used for saving models. The body of a POST request contains the JSON of the run-time model and a filename parameter which must not contain the characters "/", "\" or "." for security reasons.

The GET method has two uses. The first use is to retrieve a previously saved model. It requires a filename parameter and returns the model as a JSON string. The second use is to retrieve a list of all models stored on the server. This will be used later to allow the user to select a model to load from a drop down box.

The XML library file

The XML library file structure and layout is directly related to the Java object data structures and applies established XML design patterns and ends up with the following XML tree structure.

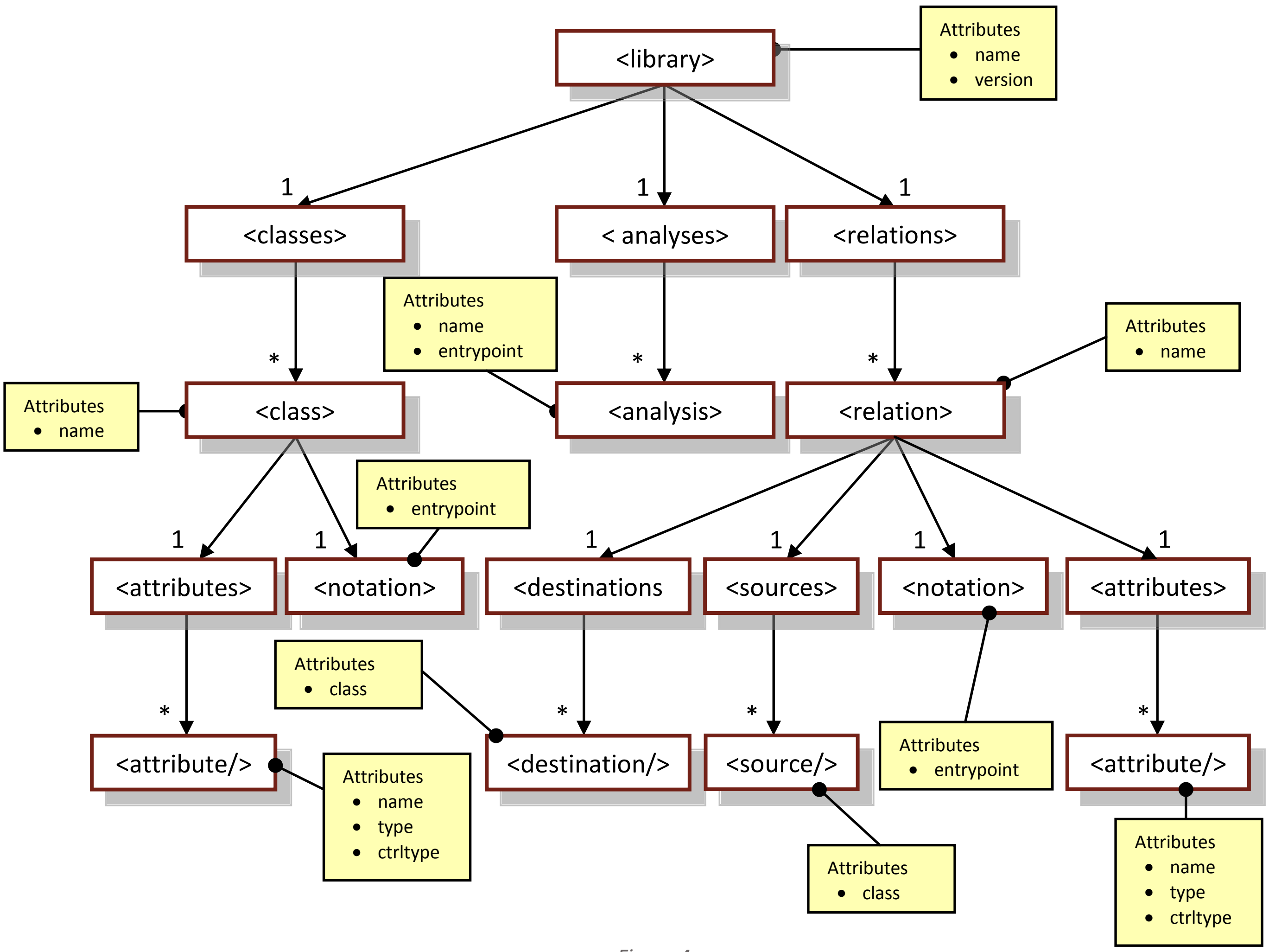


Figure 4

The <library> node

The <library> node is the top level definition of a specific meta-model library. The <library> node has two attributes, *name* which is the library name (e.g. "Business process modelling") and *version* which is a definition version for that library. The *version* attribute is included for specific modelling libraries developed for a client organisation that may over the course of its maintenance need several updates, fixes or extensions and a version string is useful for tracking changes. Both attributes are string attributes.

The <classes> node

The <classes> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <class> elements.

The <class> node

The <class> node represents a class definition within the library. It has one string attribute called *name* and contains two child nodes, <attributes> and <notation>.

The <notation> node

The notation node defines graphical representation code for objects of this class. This node has one attribute named *entrypoint* that is the name of a JavaScript class definition. The body of this node is JavaScript code containing the entrypoint definition. The JavaScript entrypoint function is a JavaScript class definition that inherits from the super class “Object” and may optionally override the default draw() method with the modelling element specific graphical representation code.

The <attributes> node

The <attributes> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <attribute> elements. This node can be found as a child node of both <class> and <relation>.

The <attribute> node

This node defines an attribute of a class or relation. This node can be found as a child node of both <class> and <relation>. This is node has no body and just contains three attributes named *name*, *type*, and *ctrltype*. The *name* attribute is a string attribute that contains the name of a modelling class attribute. The *type* attribute contains the data type of this modelling attribute. Valid values for *type* are:

- STRING
- TIME
- DATE
- INTEGER
- ENUM
- FLOAT

The *ctrltype* attribute contains what sort of input field should be displayed the user. Valid values for *ctrltype* are:

- TEXT
- SELECTBOX
- RADIO
- CHECKBOX

The <analyses> node

The <analyses> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <analysis> elements.

The <analysis> node

This defines a specific meta-model modelling mechanism that can be executed by the user. An analysis node has two attributes named *name* and *entrypoint*. The *name* attribute is the analysis or mechanism name and the *entrypoint* attribute is a JavaScript callable function that executes the analysis or mechanism. The body of this node is JavaScript code containing the entrypoint definition that is executed in the browser.

The <relations> node

The <relations> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <relation> elements.

The <relation> node

This node defines a logical relationship between modelling objects. This node has one XML attribute named *name* that is the name of this relation (e.g. “Sequence”). The <relation> node contains four child nodes named <attributes>, <notation>, <sources> and <destination>.

The <notation> node

The notation node defines graphical representation code for relation instances of this relation. This node has one attribute named *entrypoint* that is the name of a JavaScript class definition. The body of this node is JavaScript code containing the entrypoint definition. The JavaScript entrypoint function is a JavaScript class definition that inherits from the super class “Connector” and may optionally override the default draw() method with the modelling element specific graphical representation code.

The <destinations> node

The <destinations> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <destination> elements.

The <destination> node

This node represents what modelling class types the current relation definition may terminate at when instantiated. This node contains no body data and a single attribute named *class* which should co-respond to an existing <class> node name.

The <sources> node

The <relations> node applies the collection element XML pattern (Toivo Lainevool 2004). Its only purpose is to define a collection of child <source> elements.

The <source> node

This node represents what modelling class types the current relation definition may originate from when instantiated. This node contains no body data and a single attribute named *class* which should co-respond to an existing <class> node name.

Client-side data structures

Client-side code will be written in JavaScript using JavaScript objects and will contain utility functions for manipulation of the interface, classes, relations and analyses. Most of the utility code will be used by the client core code but other routines could be used by modelling mechanisms to facilitate their execution.

Modelling element structures

All instantiated modelling elements must inherit from one of two client-side classes. These are:

- **ObjectClass**: This is the super class for all modelling objects
- **Connector**: This is the super class for relations between modelling objects

Both super classes provide methods and event handlers for graphical representation and user interaction with the objects. These objects can be represented in UML as follows:

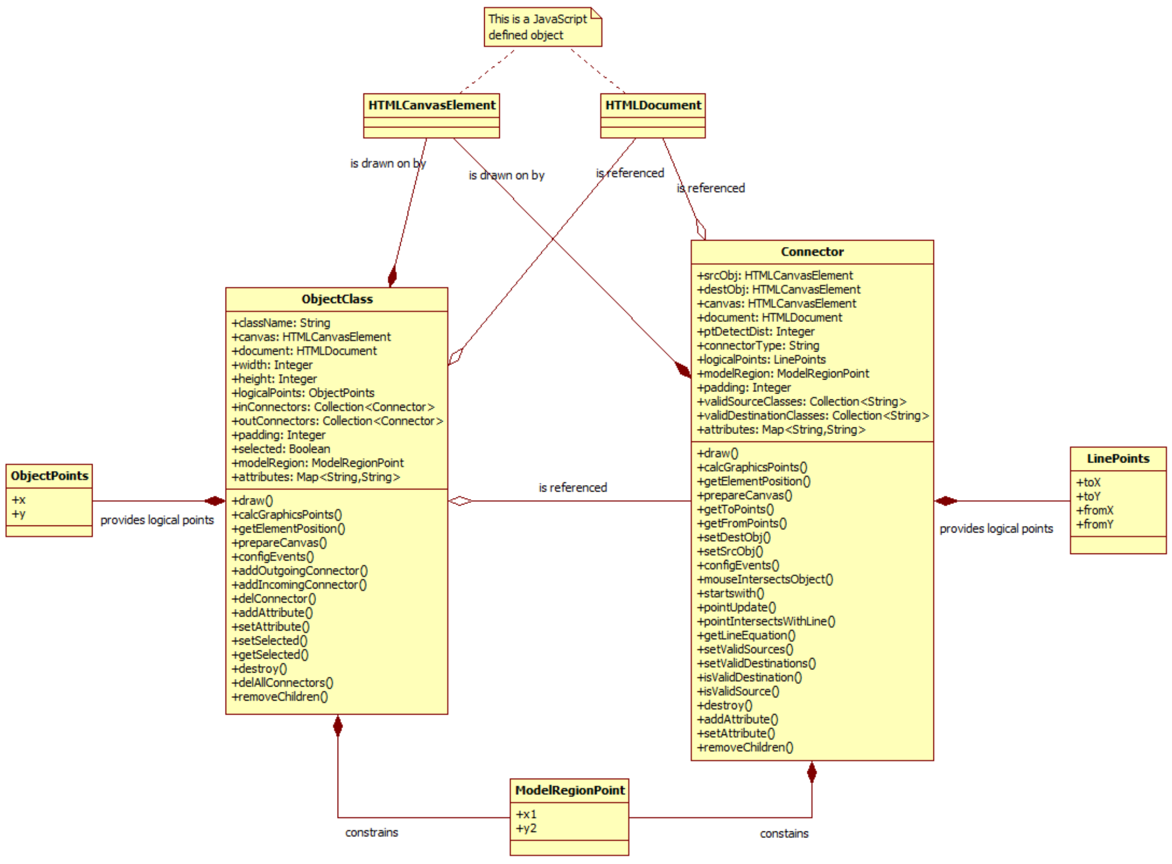


Figure 5

By default the object super class draws itself on the canvas as a white 100 pixels by 100 pixels square with a black border and the connector class draws itself as a single black line from point (100,100) to (190, 190) on the modelling area. A modelling language developer uses these two classes to create customised connectors and relations. Generally speaking they simply inherit the super class and override the draw method to specify a customised notation. Some more complex modelling elements that use labels may need to override other methods such as calcGraphicsPoints() or prepareCanvas().

Each class has one HTMLCanvasElement set to the attribute *canvas* at construction. Only one HTMLCanvasElement may be assigned to an object or connector and may not be assigned to any other instance.

Connectors additionally store a HTMLCanvasElement as a source and destination object (in the class attributes named “srcObj” and “destObj”). When a HTMLCanvasElement is assigned to either of these attributes that connector instance is considered to be logically connected between the source object and the destination object. When a connection is created the JavaScript ObjectClass instance, that represents the source or destination object, will have addOutgoingConnector() or addIncomingConnector() invoked by the connector instance. The argument to either of these calls will be the calling connector instance. This is to say that when a logical connection is made, the connector instance stores a source and destination object and both of these objects also have the connector instance referenced as an incoming or outgoing relation. This allows modelling mechanisms to determine what objects are connected to what objects via relations very easily.

Run-time meta-model structure

The meta-model itself is stored as the JSON object that meta-mode web service provides. The JSON object exists within the modelling application for the duration of the run and is considered read-only for retrieving information about the modelling language. The structure of the JSON object is a simplified version of the server-side data structures. The structure of the JSON meta-model object can be represented as:

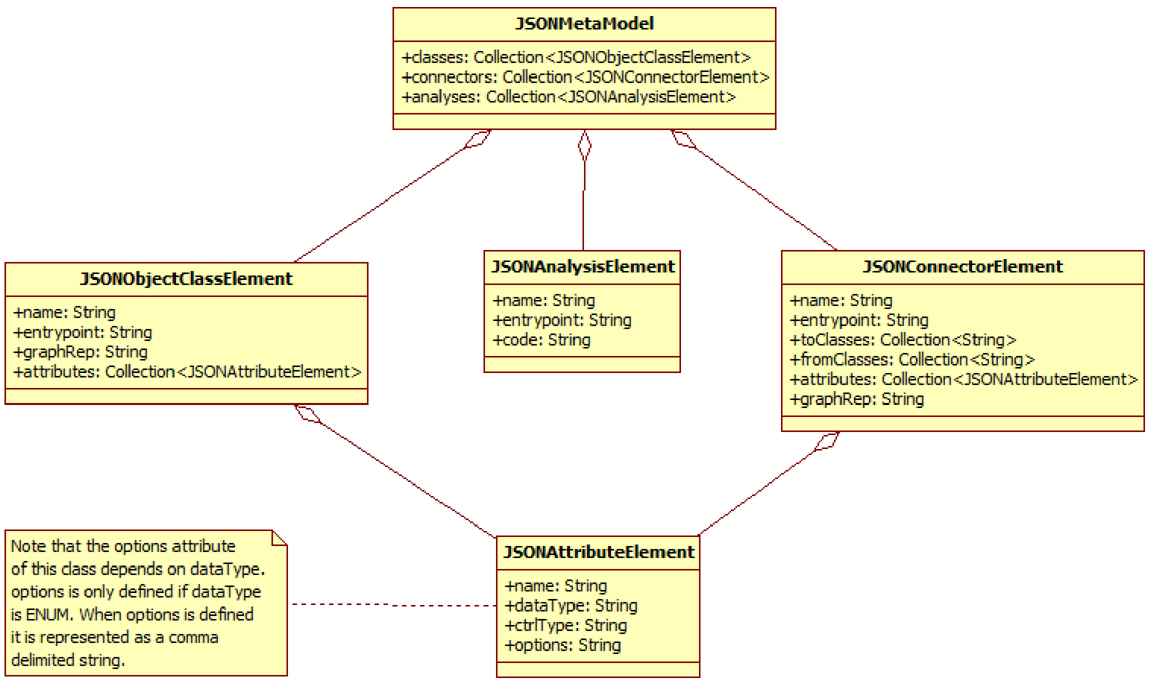

JSONMetaModel
+classes: Collection<JSONObjectClassElement>
+connectors: Collection<JSONConnectorElement>
+analyses: Collection<JSONAnalysisElement>
JSONObjectClassElement
+name: String
+entrypoint: String
+graphRep: String
+attributes: Collection<JSONAttributeElement>
JSONAnalysisElement
+name: String
+entrypoint: String
+code: String
JSONConnectorElement
+name: String
+entrypoint: String
+toClasses: Collection<String>
+fromClasses: Collection<String>
+attributes: Collection<JSONAttributeElement>
+graphRep: String
Note that the options attribute of this class depends on dataType. options is only defined if dataType is ENUM. When options is defined it is represented as a comma delimited string.
JSONAttributeElement
+name: String
+dataType: String
+ctrlType: String
+options: String

Figure 6

Run-time model structures

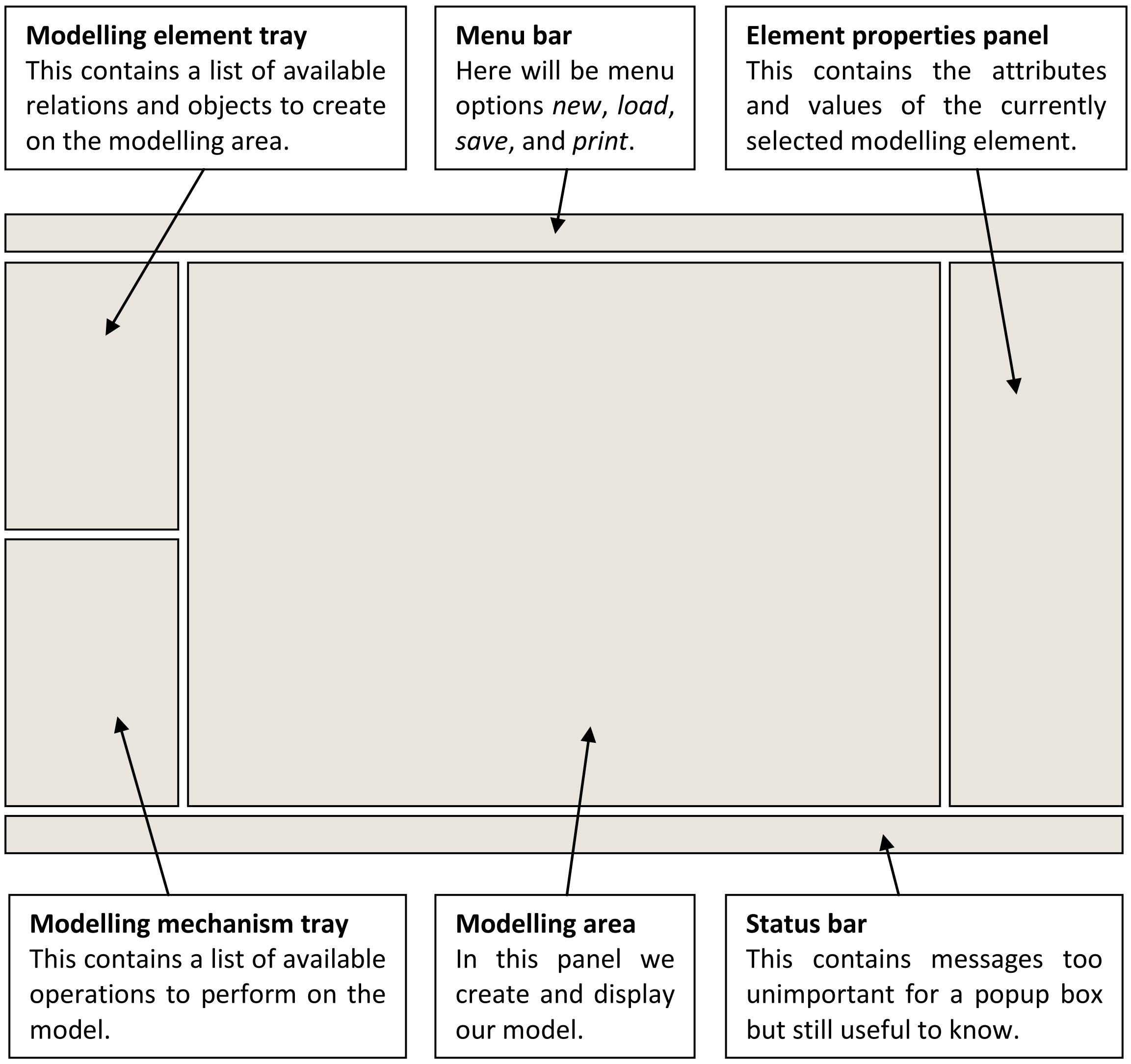
The next data structure is the run-time model data. Since both connector and object classes store the logical structure of the model the client application will use two collections to store a list of all relations and objects. This will allow modelling mechanisms to locate the modelling elements. Some simple searching functions will be created. An example of such a routine is a search function to find the respective JavaScript object given a HTMLCanvasElement object. This search iterates through the list and returns to the first JavaScript object that is composed of the given canvas class.

User-Interface components

For the interface design there are six distinct components. These are:

- A modelling area
- A UI mechanism for the creation of new instances of modelling elements
- A UI mechanism for displaying descriptions of modelling elements
- A UI mechanism for executing modelling mechanisms (analyses, etc.)
- A method of displaying messages to the user such as errors, warnings, etc.
- A UI mechanism for loading and saving models

I combined these requirements and came up with a design.



Modelling element tray
This contains a list of available relations and objects to create on the modelling area.

Menu bar
Here will be menu options *new*, *load*, *save*, and *print*.

Element properties panel
This contains the attributes and values of the currently selected modelling element.

Modelling mechanism tray
This contains a list of available operations to perform on the model.

Modelling area
In this panel we create and display our model.

Status bar
This contains messages too unimportant for a popup box but still useful to know.

Figure 7

The modelling element tray

The modelling element tray is a list of clickable buttons that display the name of a modelling element. The user can click on a button to create the named modelling element in the modelling area. This tray will be sub divided into a section for object classes and another section for relation classes.

The modelling mechanism tray

The modelling mechanism tray is a list of clickable buttons that display the name of a modelling mechanism. The user can click on a button to execute the mechanism on the current model. The results from a mechanism will be displayed according to the mechanism implementation. In most cases this will be a message dialog giving the summary of the operation however a modelling mechanism developer could create new HTML documents or not given any output at all.

The modelling area

The modelling area or region is positioned in the centre. It is the largest UI component because models can become very large. As a result this region will have horizontal and vertical scroll bars to allow the modeller to create very large models several screens wide.

The menu bar

The menu bar is a short horizontal list of general operations to perform on the model. Such as “New Model”, “Load Model”, “Save Model” and “Print Model”. The menu bar will be classically positioned at the top of the application UI

The properties window

The properties window will contain a dynamically created table of attributes and their values for the currently selected modelling element. This table will additionally allow editing and saving of those attributes and so input fields, drop down boxes and radio lists will be dynamically created and set by default to the current attribute value.

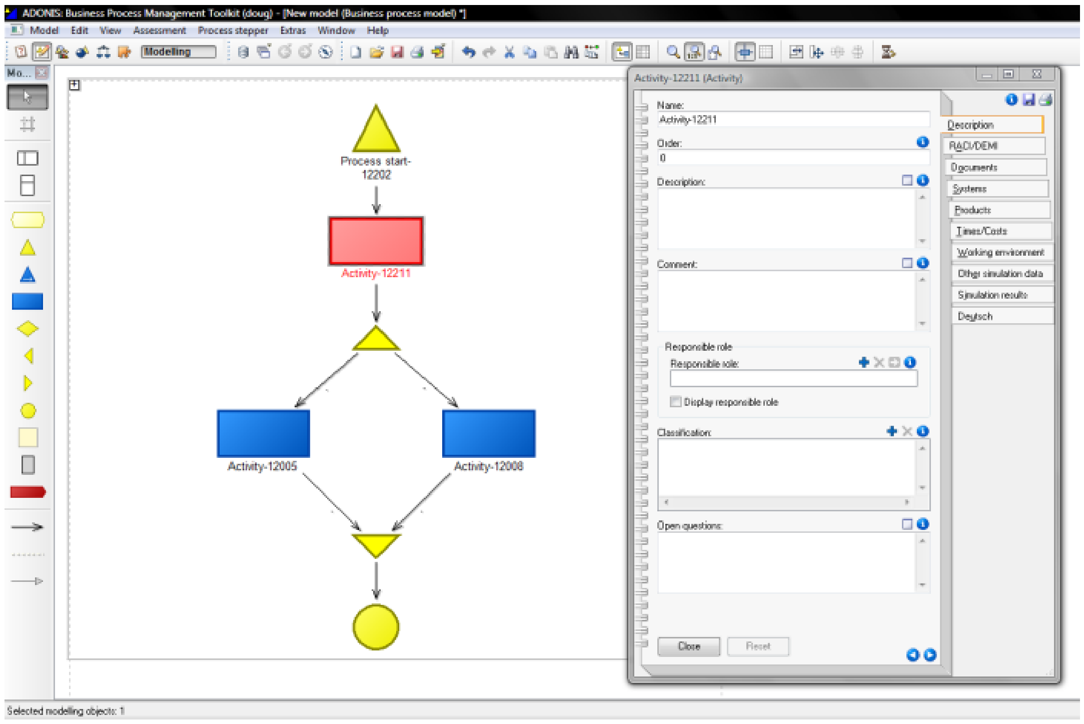
Justification for layout design

Here I will justify the layout of my design. In order to do this it is useful to investigate how other applications that have large editable regions, support the creation of elements on that region and support property editing on elements in the large editable region. A similar layout to these applications will make it easy for new users to learn how the interface for this project works.

I discovered many types of application that meet the given description of UI components. There are other modelling applications (e.g. Rational Rose, Star UML, ADONIS, Oryx-Editor, etc.), graphics applications (e.g. Adobe Photoshop), animation applications (e.g. Adobe Flash, etc.), and publishing applications (e.g. Microsoft Publisher).

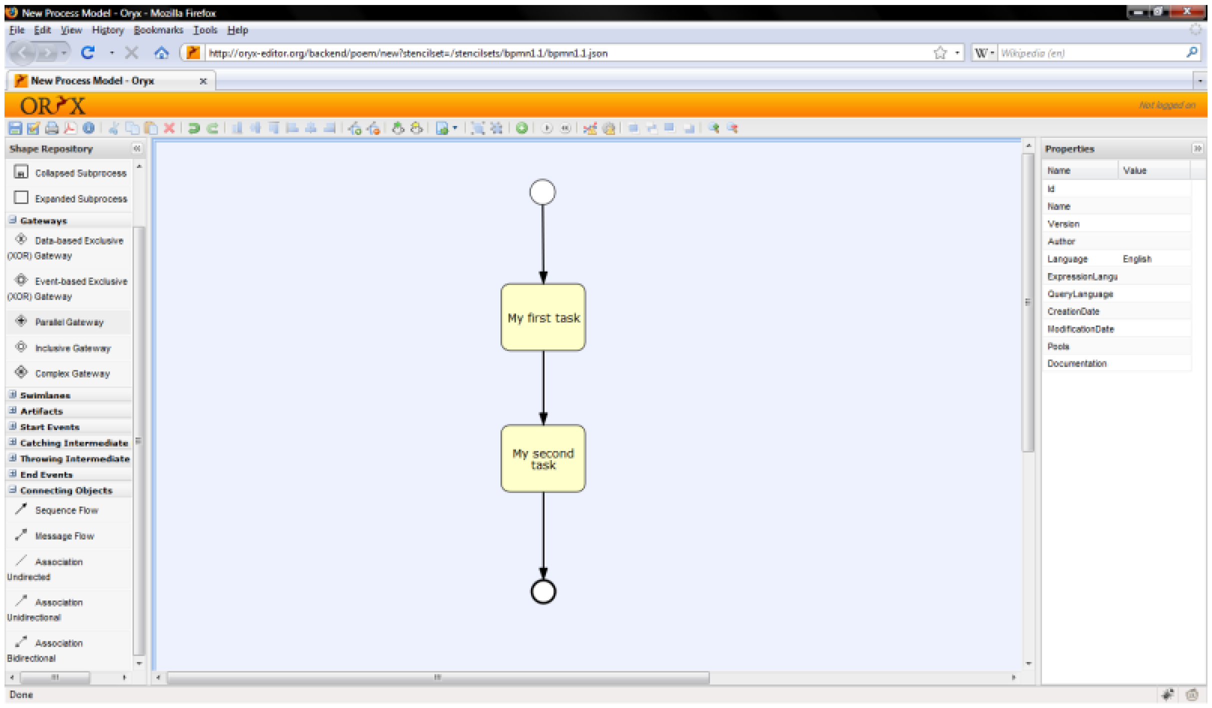
Below I have included a screenshot and brief explanation of the layout of a selection of modelling tools to demonstrate that my proposed layout is indeed a common one and that as a result it should be easy for users to adapt to my application UI.

ADONIS



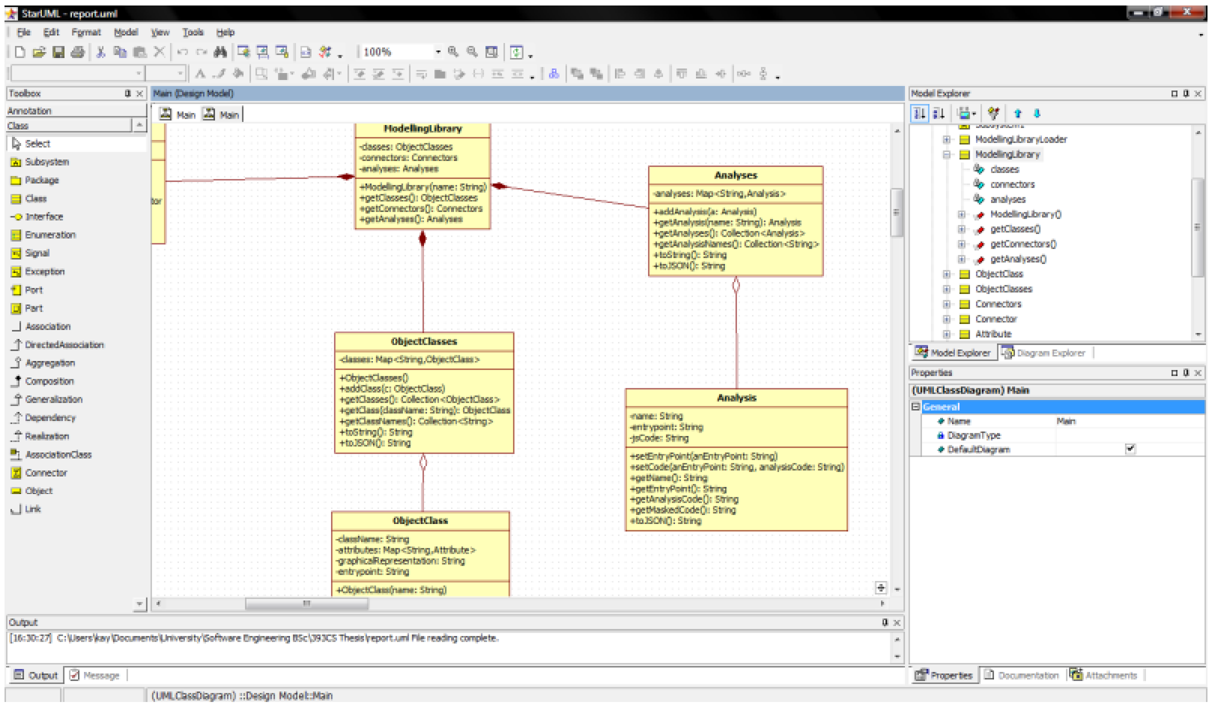
ADONIS is a business process modelling (BPM) focused application that implements the meta^2-modelling concept. As you can see the modelling elements are on the left and are represented as symbols. There is a large modelling region in the middle. Properties are given as a pop-up "notebook" dialog which can be positioned anywhere by the user.

Oryx-Editor



This is another business process modelling application except this one deals primarily with the Business Process Modelling Notation (BPMN) standard. This application like the one described in this project is a web application. The modelling elements are available on the left, there is the modelling area in the middle and element properties are on the right.

StarUML



StarUML is a CASE tool application that is capable of creating class diagrams and use case diagrams. This application has the modelling elements available in the panel on the left, a central modelling area and a properties panel on the right.

Modelling mechanisms

All modelling mechanisms in my project have a name, an entrypoint and some JavaScript code. The entrypoint is the name of a callable JavaScript function to invoke in order to begin execution of the mechanism. The name is a human-friendly name that represents the mechanism. There are two types of mechanisms briefly touched upon in the preceding design sections. These were:

- GraphRep (also known as the modelling element notation) and
- Analyses

GraphReps are classified as mechanisms because of their implementation rather than their conceptual place in this project. GraphReps use the same mechanism for execution as the analyses. That is to say that both have entrypoints and some executable JavaScript code associated with them.

When the web application index page is generated it will first print all the GraphReps and analyses in the current meta-model near the beginning of the page, making all relevant JavaScript code available throughout the duration of the applications execution.

The second phase to implementing the modelling mechanisms is that the buttons in the modelling element tray and the modelling mechanisms tray must be associated with the co-responding entrypoint. For modelling elements this will be a GraphRep. For analyses this will be an analysis entrypoint. When the user clicks on the button the appropriate entrypoint function is invoked and the mechanism is executed.

As a special case when the modelling element buttons are clicked there are some extra operations that must be carried out such as adding the newly created modelling element to the run-time model data structures.

Chapter 4: Implementation

This chapter describes issues and factors involved with implementing this project.

Tomcat installation and set up

In order to implement the prototype I first need to create an environment for running it and testing it. The Apache Tomcat web server can be downloaded from http://tomcat.apache.org/. Once downloaded the installation file is executed and the Tomcat is installed on the development system. The Tomcat should run as a service and the Tomcat Monitor should also be running. Running the monitor means that the Tomcat and be shutdown and restarted whenever there are changes to the server-side class files during development.

WebApp directory layout

The second step is to set up a skeleton WebApp folder on the Tomcat. A WebApp is a directory structure of a small web application running on the Tomcat. The Tomcat may have many WebApps installed and any code that runs on the Tomcat must be associated with a WebApp and a WepApp directory.

Each WebApp requires a “WEB-INF” directory which stores back-end resources used by the WebApp during its execution. Within this directory there is two sub directories named “lib” and “classes”. The “lib” directory should contain all third party libraries used by the application. The “classes” directory should contain the server-side Java package structure and compiled Java classes. Additionally within the “WEB-INF” directory there must be a “web.xml” file. This holds the global configuration for the WebApp.

The WebApp now has the basic structure required for the project. The next step is to install any required software libraries. The Java programming libraries “xml-apis.jar” and “xercesImpl.jar” were copied into the “lib” directory.

Once the application has been implemented the file structure of the WebApp directory looks like this:

```
webapps
`- WebBasedModelling
   |- WEB-INF
   |  |- lib
   |  |  |- xercesImpl.jar
   |  |  `- xml-apis.jar
   |  |- classes
   |  |  `- WebBasedModelling
   |  |     |- metamodel
   |  |     |  |- Analyses.class
   |  |     |  |- Analysis.class
   |  |     |  |- Attribute.class
   |  |     |  |- Connector.class
   |  |     |  |- Connectors.class
   |  |     |  |- ControlType.class
   |  |     |  |- DataType.class
   |  |     |  |- ModellingLibrary.class
   |  |     |  |- ModellingLibraryLoader.class
   |  |     |  |- ObjectClass.class
   |  |     |  `- ObjectClasses.class
   |  |     `- service
   |  |        |- MetamodelService.class
   |  |        `- ModelService.class
   |  `- web.xml
   |- AnalysisAPI.js
   |- ClassAPI.js
   |- connector.js
   |- index.jsp
   |- InterfaceAPI.js
   |- library.xml
   |- object.js
   |- RelationAPI.js
   `- style.css
```

Object implementation

As described in the design section, one of the JavaScript classes that must be implemented is the ObjectClass class and is used as a super-class for all future developed modelling object classes. It provides basic functions for manipulating, drawing, and interacting with modelling objects. Implementing this was straight forward.

The logical structure of the model is maintained by both the objects and the connectors and when a connection is made both the involved objects and connectors have to update their internal data structures. Objects contain a list of all incoming and outgoing connectors. When a connection is made the JavaScript object that represents that relation is added to either the incoming or outgoing lists depending on whether or not the object instance is the source or destination of the connector. Since it is the connector's responsibility to construct and deconstruct a logical relationship to objects all that the object class needs to do is implement methods to add and remove incoming and outgoing connectors.

Additionally when the object is re-positioned on the modelling area, the handler for re-positioning must call a method of all incoming and outgoing connectors named pointUpdate() which refreshes the internal point data of all connectors associated with this object.

Connector implementation

As described in the design section, one of the classes that must be implemented in JavaScript is called Connector and is used as a super-class for all further developed modelling relations. It provides draw basic functions for manipulating, drawing, and interacting with modelling relations. Implementing this was not so straight forward. The reasons for this are listed and discussed below:

- Browser HTML elements are all rectangular, presenting a problem for user-interactivity on line shapes (e.g. that events handlers are triggered when an event occurs for the entire rectangular region)
- The user must be able to move the connector around the modelling area and reposition it
- Connector must be able to "attach" and "detach" itself to a modelling object and then redraw it's position when either of the modelling objects (remember there is a source and a destination object for each connector) is moved by the user

The problem of rectangular HTML elements

Browsers typically use a "block" like structure for positioning and creating elements. This meant that a line in a browser would also be physically implemented as a block or rectangular HTML element. This presents a problem in that event listeners for user-inactivity events (such as clicks, mouse drags, etc.) are attached to a rectangular region.

To solve this issue I created event handlers that will read the event information and determine using the equation of a line how far from the line the event occurred. This only applies to mouse events as all other events are non-graphical. If the event coordinates are within a certain tolerance from the line given by the line equation then the event continues as normal. If the event coordinates are beyond the tolerance then we find all HTML elements that intersect with those coordinates and then choose the element with the highest z-Index and trigger the appropriate event handler for that element. The z-Index is a HTML property that determines where 2-dimensional elements exist with respect to other 2-dimensional elements stacked on top of each other. The highest z-Index represents the top-most element and the lowest z-Index represents the bottom-most element. By doing this we can opt out of processing events that do not concern our line and ensure that the event message is not lost.

The problem of repositioning connectors

The second issue concerns how the connector should interact with the user in order to be repositioned on the modelling area. This was solved by creating 3 "touch-regions" that when clicked and dragged the connector repositions itself. Two of these touch-regions are small 10 pixel regions at the source and destination tips. Once a mousedown event occurs within either of these two touch-regions the drag handlers initialise and set up to reposition the connector relative to the original event coordinates and the current mouse coordinates. Dragging either of these regions only alters the logical points of the connector end and does not affect the other end.

The third touch-region is a region around the line body itself that is plus or minus 10 pixels from the line. Every time a mouse event is triggered the distance from the line is computed then compared against the event coordinates. When within the defined tolerance the touch-region is considered to be activated. This touch-region is used for dragging the entire connector and alters the logical points of both ends of the line simultaneously relative to the original mouse coordinates and the current mouse coordinates.

The problem of re-rendering the modelling structure during user-interaction

The third issue is that of how to inform connectors of a change in the graphical model that directly affects them. If a connector represents a directional relationship between two modelling objects we first need to be able to establish that relationship logically and secondly we also need to be able to redraw the connector when either modelling objects is moved.

The solution for these issues ties in with how the line end-point touch-regions function. When the mouse is released from dragging one of the line end-points all HTMLCanvasElements that intersect with the final coordinates are found and the element with the highest z-Index is chosen. The connector instance then sets the intersected object to be the appropriate connector destination or source object. The JavaScript object instance of the intersected canvas element is then retrieved from the run-time model data. Finally the connector instance is added to a list of incoming or outgoing connectors from the found JavaScript object instance.

When the object is moved the incoming and outgoing connectors are iterated and their logical points are updated relative to the new position of the source and destination objects. This allows us to successfully maintain the model structure graphically.

Modelling mechanisms implementation

The modelling mechanisms were implemented by iterating over all available mechanism code and printing it out in index.jsp within <script> tags. When an analysis button was clicked the respective entrypoint is invoked and the mechanism is executed. When a modelling element button is clicked the addRelation or addInstance functions are called which in turn invoke the appropriate graphRep entrypoint.

Problems with GWT-EXT

The original choice of technology for interface design was a Java library based on Google Web Tools combined with a JavaScript library called ExtJS. A developer can use this library to rapidly and easily create powerful JavaScript web applications that rely on RPC (remote procedure calls) and AJAX (Asynchronous JavaScript and XML) for data retrieval.

I attempted to implement my user interface with GWT-EXT for the first few months of development. I built the connector and object implementations separately with the intention to merge their code with the GWT-EXT UI code. I encountered many difficulties with getting user-defined JavaScript to work with GWT-EXT. I later discovered this was due to scoping within the compiled JavaScript output of GWT-EXT.

I attempted to debug the output to find a method to successfully add the elements to the modelling area but due to the complexity of the GWT-EXT output and the machine generated variable names I eventually abandoned this avenue of development and redesigned my application using techniques and an architecture that I was confident would succeed. I salvaged the meta-model Java classes and reused it when I implemented the re-designed code.

Problems in Internet Explorer

I encountered many problems with Internet Explorer. The first of which was the poor support for the canvas element and a broken implementation of the context component of the canvas. During my exploration for solutions to the canvas problem I discovered a work-around for this issue that was implemented by Google and is available through the GoogleExplorer project. The second problem was the long list of quirks and issues with the Internet Explorer DOM (document object model) implementation. Due to time constraints on this project and the difficulty of cross-browser development I decided that my application would not support Internet Explorer. I instead focused on compatibility with the Firefox, Opera and Safari browsers.

Chapter 5: Testing

Manual testing

Bounds checks and validation

Validation and bound checking was performed upon the completion of the prototype. I created a spreadsheet “test case” document that lists all the fields that need checking. You can find the full test case sheet in appendix B.

In order to configure the system for the tests I had to configure a new modelling class that I named “TestCase”.

Automated testing

Client-side

It is very difficult to perform automated testing with JavaScript however I did employ lint testing with a tool called JavaScript Lint (JavaScript Lint 2007). JavaScript Lint is a CLI (Command Line Interface) tool that checks a JavaScript source code file for suspicious language usage and prints a list of possible problems for the developer to look into. This helped me refine my code and encourages confidence in the source code.

How to set up JavaScript Lint

You can download JavaScript Lint (JSL) from http://javascriptlint.com/download.htm . Once you have unzipped the download package you can see that there are only a few files. The first step is to configure JSL by opening the jsl.default.conf file. The default configuration file is well commented and it is straight forward to customise the Lint tool for my environment.

Run to execute JavaScript Lint

Once the configuration is complete we execute the application using the following command:

```
JavaScript Lint> jsl conf jsl.default.conf
```

The Lint tool will then parse each file identified in the configuration and if any concerning issues are found the tool prints out the filename, line number and description of the issue.

The results of the Lint testing

After applying JSL to the JavaScript code I received some warnings. After altering the source code in response to the warnings and re-testing with JSL I found no problems. The JSL tool advised me to turn on the “option explicit” setting whereby JSL would check that all my variables are properly declared. Doing so would increase the measure of confidence that can be placed in the JavaScript code. I turned on “option explicit” in all my JavaScript files using JSL directives (which can be read about in the documentation on the JSL website) I found a few further issues which I resolved. After applying the changes indicated by the results of the testing, JSL reported that there were no known errors or suspicious code practices remaining in the JavaScript component of the application.

Server-side

The server-side testing is automated through usage of the JUnit framework. JUnit is a popular testing framework that can be set up as an eclipse plug-in. A developer then just has to click run and all configured testing methods are invoked.

Test case classes are written by the developer upon completion of a class definition. Each testing method of the test case class has an assertion clause that confirms that the current application state is as expected.

I have implemented JUnit test cases for the Java classes for my classes. In addition I have to implement some classes to simulate functions of the tomcat environment. These are all included in appendix C.

Chapter 6: Evaluation

Comparison between set goals and achieved goals

When I began this project, my objects were to develop a modelling tool that:

1. is capable of logically and graphically modelling a concept
2. can be used with multiple modelling languages
3. can be customised such that new modelling languages can be implemented or existing modelling languages may have features added or removed as per the requirements of the modeller
4. can be used to describe, analyse and optimise the modelled concept
5. is easy to be used as a collaborative tool between multiple departments of an organisation
6. is easy to install, update and maintain throughout the organisation
7. is platform independent and can thusly be used by multiple departments of an organisation even if not all departments use the same operating system

I will address each goal individually below, giving an assessment of how complete the goal is.

Goal 1: To be capable of logically and graphically modelling a concept

This goal was achieved. A user may position modelling objects, create relations between those objects and describe each element of the concept graphically. The run-time model data supports the requirement of being able to retain the model logic and allows us to save, load and analyse the model.

Goal 2: To be usable with multiple modelling languages

This goal was achieved. Multiple modelling languages can be defined in a separate library XML file and loaded/unloaded into my prototype application via the web.xml configuration file. The drawback is that in order for a new library to be loaded the web server must undergo a restart for the changes to take effect.

Goal 3: For each modelling language to be customisable

This goal was achieved. Each library file is an open unencrypted XML file and so can be edited by anybody. This means that somebody with knowledge about how the libraries are defined may customise a base language into a customised language for a specific purpose and environment. However, like goal 2, changes only take effect after the web server has been restarted. This presents a minor inconvenience in that the language developer must restart the web server each time they wish to test their changes to the language.

Goal 4: To aid in analysis and optimisation of the modelled concept

This goal was achieved to an extent. There is an analysis component that works very well. There is no built-in mechanism for automatic optimisation of models because the rules for optimisation would rarely radically between modelling languages. However it would be possible to write an analysis mechanism that automatically alters the model to an optimised state.

Goal 5: To be used as a collaborative tool

This goal was achieved to a limited extent. There is a load model and save model mechanism that works well. This can be used to allow members of other departments to work on the same model.

However there is no synchronisation mechanism for simultaneous model access between two different users. Nor is there any mechanism to determine if another user has the model open for editing (i.e. placing locks on models). There is also no mechanism for two users to work on the same model in real-time like some other collaborative tools.

Goal 6: To be simple to set up

Evaluation of this goal requires a look at both the software installation and the development of new modelling languages.

Installation of the system is done in two parts. The first part is to install an Apache Tomcat server on a computer system. The second part is to install the application. When this application is deployed it comes in a self-extracting Tomcat webapp archive called a *.WAR file*. The administrator must copy the archive file "*WebBasedModelling.war*" into the *webapps* directory of the Tomcat installation and then you start the Tomcat webserver. The war file is automatically deployed. To configure a particular meta-model the administrator needs to edit the web.xml file that can be found at the path "*<Tomcat Directory>/webapps/WebBasedModelling/WEB-INF/web.xml*". The administrator should edit the initialisation parameters for the index_jsp servlet and change the default path to the path of the meta-model they wish to set the server to run. An example of the configuration of the index_jsp servlet is given below:

```xml
<servlet>
	<servlet-name>index_jsp</servlet-name>
	<jsp-file>/index.jsp</jsp-file>
	<init-param>
		<param-name>modellinglibrary</param-name>
		<param-value>
			<!-- Give the path to the meta-model here -->
		</param-value>
	</init-param>
</servlet>
```

The second issue of meta-model development I believe was done well. The XML meta-model file is easy to read and to understand. However the modelling mechanisms (analyses and notation sections) could be difficult to understand without proper development documentation on the available JavaScript API. There is also the issue of namespacing with the entrypoints. A developer who is unaware of all the variable names and the names of existing functions may accidentally name their entrypoint the same name as global variable or function that will cause the application problems once loaded.

In conclusion the installation of the application itself is straight forward and should be easy for any administrator familiar with configuring applications. However there are some existing issues with meta-model development that would need to be better addressed to fully satisfy this goal.

Goal 7: Platform independence

To evaluate this goal I must look at both client-side platform independence and server-side independence.

The server side is entirely implemented using Java. Further the Tomcat server itself was implemented using Java and has minimal platform dependence issues. There are minor platform dependant issues with each Tomcat package (such as the Windows package implements a Windows service and Windows specific code) which is why Apache offer several platform packages. As a result of my implementation and the implementation of the Tomcat web server, the server-side component of my system can be deployed on any platform that supports a JVM (Java Virtual Machine).

The client-side code was implemented using HTML and JavaScript and no other plug-ins. The majority of modern browsers support HTML and JavaScript. My application should be viewed as a desktop browser application rather than an application designed to run on embedded devices and their browsers. This means that an even greater proportion of browsers implement the required technologies to run my application. However the HTMLCanvasElement is a new addition to HTML and so support for it is limited. I focused on testing my system in Opera, Firefox and Internet explorer and found that Internet Explorer had too many quirks and issues (See Chapter 5: Implementation for a description of these issues) that needed to be overcome.

I dropped Internet Explorer from the list of browsers my project can support due to time constraints. As a result the client-side code for this project has poor platform independence. The two remaining supported browsers are both available for other platforms where as Internet Explorer is only available for Apple Mac OSX and Microsoft Windows. This means that my system can still be used on many operating systems that Internet Explorer doesn’t support (e.g. Linux).

Implementation improvements

- More sophisticated syntax validation perhaps similar to IDS Scheers BPMN tool ARIS (IDS Scheer 2008)
- For better efficiency connectors should store the JavaScript objects as the source or destination object rather than a HTMLCanvasElement object to avoid having to iterate over all object class instances in order to find the JavaScript object that represents a source or destination
- The XML meta-model files could be replaced by JSON meta-model files which could simplify the process of constructing and retrieving the meta-model
- Allow the object class definitions to determine where the destination & source logical points of connectors draw to. This would allow for modelling languages such as circuit diagrams to define precisely which connector node on the graphRep a modelling “wire” should appear to be connected to
- The run-time model could have been implemented better by taking a more object orientated and structured approach allowing for faster retrieval of run-time model data

Chapter 7: Conclusions

Summary

This was an ambitious project given that the allocated development time was initially just 10 working days. I have achieved much of what I set out to achieve but some of the project goals were not fully met and some features not implemented.

It was also hoped that this project would be able to go a step further and implement a meta^2-model and allow the graphical definition of a new meta-model. I did some experiments relating to using this project as a platform for the development of meta-models. I found that it was possible to define a “meta^2-model” meta-model and use that meta-model and the current prototype to develop further meta-models that could be successfully loaded and executed by my software application. This was accomplished by defining a modelling mechanism to generate the XML meta-model from the run-time model. A copy of this “meta^2-model” meta-model is included in appendix F.

Further work

This project has a lot of potential ways within which it can be improved. Some of these are UI features. Some of these are additions to the meta-language. Others are the expansion of the framework APIs. Below I summarise improvements that can be made to this project in those three categories.

User interface features

- Connector bend points to allow the customisation of the path connectors take over the modelling area
- User management to allow different user accounts, model permissions, and to load a specific meta-model for the current user
- A date and time “picker” to allow users to graphically enter a date
- A mechanism to allow attributes to have a specific order in the properties window
- A meta-model toolkit that users can use to create a meta-model through their browsers and save that meta-model
- An undo feature

Meta-language improvements

- The introduction of true modelling libraries that contain multiple modelling language definitions (model types)
- The introduction of attributes for the model itself
- Relational cardinality in order to specify the lower and upper limits of how many relations of a certain type may be incoming or outgoing relations of a particular modelling class
- Meta-model encryption for proprietary meta-models
- Class and relation inheritance to allow specialisation of base classes

Framework improvements

- Expansion of Analysis API
- A more generalised ObjectClass and Connector JavaScript classes to make it easier for language developers to build new classes and relations with little knowledge of the underlying code base.
- Namespacing in modelling mechanisms was not addressed in this project and work should be done to create reliable namespacing mechanisms

References

Apache Software Foundation (2009) Apache Tomcat [online] available from <http://tomcat.apache.org/> [15 January 2009]

BOC Information Technologies Consulting AG (c. 2008) ADONIS Business Process Management Toolkit [online] available from <http://www.boc-group.com/index.jsp?file=WP_582571cc1ed802de.46e381.f59775478f.-7f17> [02 April 2009]

BPTrends (2006) Process Modeling in the 21st Century [online] available from < http://www.bptrends.com/publicationfiles/05-06-ART-ProcessModeling21stCent-Recker1.pdf> [08 April 2009]

Business Process Technology Group, University of Potsdam (2009) The Oryx Editor [online] available from <http://bpt.hpi.uni-potsdam.de/Oryx/> [02 April 2009]

Google Inc. (2009) ExplorerCanvas [online] available from <http://excanvas.sourceforge.net/> [15 April 2009]

Harold, E. R. (2002) Processing XML with Java.
Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

IDS Scheer (2008) Adopting BPMN with ARIS [online] available from <http://www.ids-scheer.com/set/6473/EBPM%20-%20Stein%20-%20Hagen%20-%20BPMN%20with%20ARIS%20-%20AEP%20en.pdf> [15 April 2009]

JavaScript Lint (2007) JavaScript [online] available from < http://javascriptlint.com/> [04 April 2009]

Megginson Technologies Ltd. (2005) SAX [online] available from <http://www.saxproject.org/> [02 April 2009]

StarUML (2005) StarUML – The Open Source UML/MDA Platform [online] available from <http://staruml.sourceforge.net/en/> [02 April 2009]

Technische Universität Berlin (1998) Dimensions and Dichotomy in Metamodeling [online] available from <http://user.cs.tu-berlin.de/~geislerr/doc/GKP98.ps> [15 April 2009]

Toivo Lainevool (2004) XML Design Patterns – Collection Element [online] available from <http://www.xmlpatterns.com/CollectionElementMain.shtml> [02 April 2009]

Toivo Lainevool (2004) XML Design Patterns – Container Element [online] available from <http://www.xmlpatterns.com/ContainerElementMain.shtml> [02 April 2009]

University of Vienna (2002) Metamodelling platforms [online] available from <http://www.dke.univie.ac.at/mmp/FullVersion_MMP_DexaECWeb2002.pdf> [02 April 2009]