OPEN MAPPING LANGUAGE

Draft Definition of a Neutral Mapping Notation

Author: Robert Worden

rpworden@me.com

Draft 01a, 22nd July 2010

Abstract: Semantic Mappings define the precise semantic relationships between physical data structures and UML class models. They define how data structures carry the information in class models. They can play many roles in standards-based interoperability, such as asserting and testing standards conformance, and supporting automated meaning-preserving translations between data structures. Mappings are widely used, but often in informal or proprietary notations. There would be many benefits in using a single open mapping notation. Adoption of a single mapping language will stimulate the market both for mappings and the tools to support them, and will enhance interoperability and quality.

This paper describes an open neutral language for expressing mappings, which is a candidate for adoption as a standard mapping language by HL7. The language is designed to be as simple as possible while having adequate expressive power. It has been evaluated by its use in a set of open-source tools, in complex healthcare applications.

Table of Contents

1. INTRODUCTION

1.1 Need for a Standard Mapping Notation in Healthcare

Large-scale healthcare applications typically use multiple overlapping models, message formats and data structures:

- National initiatives such as HITECH have adopted several overlapping standards
- Within one SDO, there may be multiple overlapping standards (e.g. HL7 V2, V3 and CDA)
- There are requirements for local data formats and Domain Specific Languages with precise semantic relations to standards (e.g. Green CDA, Micro-ITS, RIM ITS, Detailed Clinical Models, Virtual Medical Record)
- Healthcare providers need to interface IT systems with a wide range of different data models and message formats.
- The HL7 Enterprise Compliance & Conformance Framework (ECCF) has a Specification Stack with layers from OMG's Model Driven Architecture (MDA): CIM, PIM, and PSM. ECCF Conformance Statements are testable relations between artefacts in different layers. They are mappings between layers.

National programs, SDOs, IT suppliers and healthcare providers frequently need to understand and document the precise relationships between different healthcare models and data formats. Failure to do so carries risk to patient safety. The relationships between different semantic models and data formats are commonly referred to as 'mappings', and the activity of defining them is called 'mapping' between the formalisms.

While mapping is critical to the integrity of healthcare IT systems and to patient safety, there is no single widely-used notation for mapping, and there are no agreed quality criteria for mappings. Mapping is typically done behind closed doors within implementation projects, with the results buried in code. Mappings are documented informally in Excel spreadsheets, and become shelfware.

A single standard notation for mappings would have clear benefits to the healthcare IT community, and for patient safety:

- Different groups could exchange results in the notation, getting common insights and sharing problems.
- Mappings could be made available for peer review and ballot, increasing their quality and the reliability of the systems that depend on them.
- Mappings in a standard notation are machine-processable, and will create a market for tools
- Mappings support conformance testing
- Mappings expose interoperability problems early – saving costs
- Mappings are more maintainable than code

- A common mapping notation will create a market for mappings in the notation.

Mappings are a critical element of the Healthcare IT eco-system. They will continue to be so, as long as no one healthcare data standard rules the world. Yet mappings have remained a second-class citizen of the Healthcare IT skill-set and toolkit. Failures of mapping have remained hidden, and have contributed to poor system integration.

By adopting one common notation for mappings, HL7 can change this. A standard mapping notation will make mappings a shareable, peer-reviewed resource, and increase their quality.

1.2 HL7 Project to Choose and Adopt a Mapping Notation

The Implementation Technology Specifications (ITS) working group of HL7 International initiated a project in May 2010 to choose and adopt a neutral mapping notation. The wiki page for the project is at http://wiki.hl7.org/index.php?title=Neutral_Mapping.

The milestones of the project are:

a) Write requirements for the notation

b) Receive feedback on requirements and refine them as necessary

c) Written definitions exist of one or more candidate mapping notations

d) Receive feedback on written definitions of notations and refine them

e) Projects use candidate notations and provide feedback

f) Written definitions of chosen notations submitted for ballot

g) Balloting of chosen notations complete

For milestone (a), a set of requirements for the mapping notation was discussed at the HL7 Rio Working Group Meeting in May 2010, and were then posted on the wiki. Those requirements are repeated in the next section, to make this paper self-contained.

This paper is a written definition of one of the candidate mapping notations, for milestone (c). It is intended to be reviewed by HL7 members and other interested parties over August and September 2010, and then to be discussed as the HL7 Working Group Meeting in October 2010.

1.3 Resources and Supporting Material

There is a set of open source Eclipse-based mapping and transformation tools, supporting a mapping language which is the close antecedent of the language described here – but not precisely the same. These tools will be updated to support precisely this language, as modified by the review process. The tools can be downloaded from http://gforge.hl7.org/gf/project/v2v3-mapping/frs/.

The project to define and adopt a standard mapping language for HL7 has a wiki page at http://wiki.hl7.org/index.php?title=Neutral_Mapping.

Please post your comments on this language definition to the HL7 Implementation Technology Specifications (ITS) Working Group List at its@lists.hl7.org. You may also wish to add to the review comments page on the project wiki.

This document is part of a review pack for the Open Mapping Language, which contains other supporting material:

1. **Open Mapping Language Definition.doc**: This document; the definition of the language
2. **Open Mapping Language Introduction.ppt** : Main concepts of the language
3. **Business Case For Mapping.ppt**: Reasons why mapping matters, and saves costs

4. **HL7 Mapping Requirement and OMG Standards.doc**: compares OMG mapping standards (QVT Relational, MDMI and others) with HL7's requirements
5. **Getting Started With Mapping.doc**: Ten tips for starting a mapping project; the beginnings of a methodology for mapping
6. **Examples Folder**: Example sets of mappings in XML and Excel formats
7. **Schemas and Models Folder**: Contains the XML Schema for the concrete syntax of the language, and the abstract syntax in EMF Ecore and XMI.

1.4 Brief Description of the Mapping Language

The mapping language described in this paper is a tabular notation. It is intended for Structure-to-Model mapping, where the structure is typically an XML document structure, and the model is a UML class model. Tables of mappings can be held in spreadsheets, in XML with stylesheets, in relational databases, or as annotations in XML schema documents.

The mapping language is currently called 'Open Mapping Language'. It is open in the sense that its definition is open, and its tabular forms can be easily read, understood or machine-processed. Open mappings can be converted to proprietary mappings.

It is intended to be the simplest mapping notation that can do the job of mapping, including only those features which have been shown to be necessary to map structures to UML model instances in common cases. No mapping notation can capture all the ways in which people use data structures to record information; this notation captures the ways most commonly used. There are procedural escapes for other cases.

There is a range of design patterns which people may use to represent information in data structures such as XML or Relational Databases. The full mapping language has a range of constructs, to capture how these design patterns are used. The mapping language has been divided into a simple **core mapping language** – which consists of just tables with five simple columns – and an **extended mapping language**, including a further eight further columns to be used where necessary.

The core mapping language, which can go a long way in describing commonly-used XML languages, is described over 10 pages in section 3 of this report, and can be understood at one sitting.

Mappings describe how an instance of the XML represents an instance of the UML class model – which is a set of objects (instances of classes) with values for their attributes and linked by instances of the associations (links). Mappings define which nodes in the XML represent objects, attribute values, and links, and how they do so. Mappings are declarative descriptions, not a procedural recipe. They are underpinned by a procedural model of how a mapping-driven **XML Reader** can convert an instance of the XML into an instance of the class model. Therefore the mappings are executable.

Each row of the tabular form defines one mapping. A mapping states that a node of the XML (defined by its XPath from a root node) represents some part of the UML class model – which may be an object in a class, the value of some attribute of an object, or a link between objects.

The core information in a mapping row consists of an XPath defining the XML node, the class of the object represented in the UML class model, or the attribute or association represented. This core information is contained in four columns of the table. A fifth column is for free text comments on the mapping, completing the core mapping language.

Other columns in the extended mapping language contain extra information which is sometimes needed to define adequate mappings. The content of these further columns is based on experience of using a mapping and transformation toolset which supports a notation similar to this notation – which can be regarded as a stage in the evolution of this notation. All columns are described with examples which show how they are used.

The tabular form of mappings, as described in most of this document, is a concrete syntax for mappings. There is also an abstract syntax (a MOF metamodel) for the mappings, which has been defined in EMF Ecore (compatible with Essential MOF, EMOF) and is described in Appendix A.

1.5 Standards on which the Language Depends

The Open Mapping Language has the following dependences on existing standards or models:

Standard	Reference	Nature of Dependence
UML (static class models)	http://www.omg.org/technology/documents/formal/uml.htm	The mapping language depends only on the static part of UML which defines UML class models. Mappings of data structures onto a UML class model define how those data structures represent information in the class model.
Eclipse Modelling Facility (EMF) Ecore	http://www.eclipse.org/modeling/emf/	Eclipse EMF Ecore is not a necessary part of the mapping language, but is a useful adjunct to it. Eclipse EMF Ecore is a subset of static UML class diagrams which is sufficient to express all UML-dependent parts of the mapping language. So mappings can be defined with reference to the EMF Ecore metamodel, or to class models expressed in terms of that metamodel. Eclipse EMF tools provide a convenient implementation of the EMF metamodel (for instance, for holding instances of UML class models in both in-memory and persistent forms)
XML	http://www.w3.org/TR/REC-xml/	XML is the main type of data structure which is used to define mappings onto a UML class model. For the purposes of mapping, other data structures such as Relational Databases or text files are first converted to an XML form by simple transforms.
XML Schema	http://www.w3.org/XML/Schema.html	When any XML language is to be mapped onto a UML class model, the structure of that language may be defined by a set of XML schemas. When mappings are to be divided into reusable modules, it may be convenient to make each mapping module correspond to an XML Schema Complex Type.
XPath	http://www.w3.org/TR/xpath20/	A small subset of the XPath standard is used to define useful sequences of nodes in an XML document. These sequences are treated as sets of nodes which represent some aspect of a UML class model; position in the sequence of nodes is rarely used.

1.6 Status of This Draft

This draft of July 2010 is the initial draft definition of the mapping notation, for review by HL7 members and other interested parties.

2. REQUIREMENTS FOR A MAPPING NOTATION

2.1 Types of Mapping Notation

There are three types of mapping: **Model-to-Model** (M2M), **Structure-to-Model** (S2M) and **Structure-to-Structure** (S2S). 'Model' refers to a semantic model (e.g. a UML model), and 'Structure' means a physical data structure (e.g. defined by an XML schema or a relational database schema).

Here, the distinction between 'Model' and 'Structure' can be understood by reference to OMG's model-driven architecture. In this respect, a 'Model' is part of the Platform-Independent Model (PIM) and may typically be a UML class model; while 'Structure' is part of some Platform-Specific Model (PSM), and is a physical data structure such as a relational database or XML message structure.

This specification assumes that logical data models in the PIM are defined as UML class models.

- **S2S** mapping notations have been developed in many commercial tools (such as Altova's MapForce, or Microsoft's BizTalk Mapper) with the aim of transforming data instances from one structure to another, sometimes bi-directionally, or in ETL applications.
- The primary use case of **M2M** mappings, as in the OMG QVT languages, is to transform one model to another (possibly uni-directionally).
- **S2M** mappings have multiple uses: asserting and testing conformance to models, and making transformations between structures. They can also support model-to-model transforms.

Any of these types of mapping can be declarative or procedural. A **declarative** mapping language makes statements of the form 'the parts of formalism F1 and the parts of formalism F2 are related in these ways'; the order of statements does not matter. A **procedural** (or imperative) mapping language says: 'To get from F1 to F2 , do this sequence of steps'; the sequence and interactions of the statements matter. Declarative languages are generally easier to write, understand and analyse. A declarative mapping language says **what** is the relationship between two formalisms, rather than **how** you get from one to the other.

2.2 Use Cases for Mapping Notations

The following uses cases for mappings and mapping notations have been identified:

No.	Description of Use Case	Explanation and Examples
1	Define precisely how a data structure represents the information in a UML class model.	This is the core use case for S2M mappings, from which all others follow. The main types of data structure are XML, relational databases, and text files. There are four main ways to create mappings: a. Retrospectively, with no tools (e.g. making tabular mappings in a spreadsheet)

		b. Retrospectively , with tool support, such as a dedicated mapping editor c. Automatically, as the data structure is designed d. Convert proprietary mappings to open mappings Of these, (c) is the easiest. (d) is only possible in restricted cases. Having precise formal definitions, in a single mapping notation, enables the following use cases (with or without machine support).
2	Assert conformance between a Platform-Specific Model (PSM = data structure) and a Platform-Independent Model (PIM = UML Model)	Conformance statements are statements that certain required items of information in the class model of the PIM can be represented in the data structure of the PSM, i.e. have adequate mappings. Given sets of mappings for system databases, APIs or message structures, conformance can be checked automatically or manually from the mappings. Conformance failures can be listed, leading to specific remedial actions.
3	Assert conformance between an instance of a Platform-Specific Model (PSM = data structure) and a Platform-Independent Model (PIM = UML Model)	Conformance of instances is different from conformance of structures, and requires further testing. Conformance of structures says: 'this data structure has all the mappings to represent the required items of information in the PIM'. Conformance of instances says further: 'this instance is an instance of a conformant data structure, and the instance has all the date items required of it, no inconsistencies, etc.'.
4	Assert conformance between different UML models	Conformance statements between models are statements that required information items in one model also exist in the other model. They are statements that adequate mappings exist. The UML models may have been derived independently (e.g. by different SDOs), or one model may have been derived from the other (e.g. CIM and PIM) Given formal mappings in a single defined notation, conformance statements can be checked automatically or manually, and failures listed for remedial action.
5	Convert an instance of a data structure to an instance of a class model	This is the function of an **XML Reader** (see below). An XML Reader can be implemented fairly simply from the definitions of the mapping notation. Then a number of sub-use cases follow: a. To check that mappings are correct as they are developed: inspecting a class model instance, made from an XML instance by the mappings, rapidly exposes problems in the mappings b. To examine the content of an XML instance, in class model terms independent of its structure, for query purposes c. To develop application code purely in terms of a class model, so those applications can retrieve data from any data structure mapped to the class model, through an XML reader d. To compare the semantic content of XML instances having different structures, by converting them both through their mappings to a common class model form.
6	Convert an instance of a class model to an instance of a data structure	This is the function of an **XML Writer** (see below), which can also be developed from the definitions of the mappings. Sub-use cases which follow: a. To develop applications code purely in terms of a class model, so those applications can write data to any data structure mapped to the class model b. To translate data instances from one mapped data structure to another, by translating 'in' from the first data structure to the class model, then translating 'out' to the second data structure c. Using (b), to implement up to N*(N-1) different data translations from only N sets of mappings.
7	Create 'cross mappings' from one data	Mappings of a first data structure relate nodes of the structure to items in a

	structure to another	UML model. Similarly, mappings of a second data structure relate its nodes to the same class model items. The two sets of mappings can then be 'joined' through the UML model items, to create and inspect cross-mappings from the first data structure to the second. This process creates S2S mappings from S2M mappings.
7	Convert open mappings to any proprietary mapping notation	The mapping notation contains all the information required to convert an instance of XML to a UML model instance and thence to an instance of another mapped data structure. Therefore the notation contains the information used in proprietary mapping notations to do data translations, and it will be possible to auto-convert open mappings to proprietary forms (e.g. for use on proprietary integration engines). Since many proprietary mapping notations are S2S, the conversion may involve use case (6).

These use cases have informed the requirements stated below.

2.3 Requirements for a Mapping Notation for Use in HL7

These requirements have been reviewed by the HL7 ITS working group, and are taken from http://wiki.hl7.org/index.php?title=Mapping_Notation_Requirements

A notation for mappings to be adopted by HL7 should:

1. **Be Linked to a Semantic Model**: Because HL7's concern is semantic interoperability, the mapping notation should link to a semantic information model, such as a RIM-based model. So the notation should be S2M or M2M, but not S2S.
2. **Have Well-Defined Meaning**: What the mapping notation can say about the relation between two formalisms, and how it says it, should be well-defined and documented.
3. **Be Semantically Expressive**: The notation must be able to express how the meaning in one model or structure is commonly represented in the other. Sub-requirements include:
    a. Can map XML, formatted text, relational databases or UML models
    b. Can express how associations are mapped (for all cardinalities; 1:1, 1:N, or N:M)
    c. Can state where attribute value conversions are needed (e.g. different code sets)
    d. Can handle mappings of lists of items
    e. Can note where mapping is not possible, and why
4. **Be Easy to Understand and Review**: It should be possible to read the mapping notation, with or without tools, and readily understand what it says about the relation between two formalisms. Sub-requirements include:
    a. Modularity: It should be possible to divide a large mapping set into smaller 'modules' of mappings, with clear definition of the links between modules and allowing reuse of modules
    b. Cross-Mapping: when several data structures are mapped onto one semantic model, it should be easy to view the parts of each data structure mapped to the same semantic model item
5. **Be Easy to Write**: It should be possible to create mappings in the formalism easily – with or without tools.
6. **Have Static Validation Criteria**: There should be clear criteria for a valid set of mappings in the notation, suitable for checking by eye or by machine.

7. **Be Declarative rather than Procedural**: Because declarative languages are easier to write, understand and analyse, a declarative mapping language is preferred (especially if transformations can be generated from it automatically – see below). The notation should allow for a 'procedural escape' for cases where mappings are hard to express declaratively.
8. **Be Executable**: It should be possible to use the mappings without further coding to convert data instances from one model or structure to another, so as to:
    a. translate data instances in live applications;
    b. test that mappings are correct;
    c. review mappings by reviewing translations made from them;
    d. support iterative development of mappings.
9. **Have Editing Support**: A mapping editor can make it easier to develop mappings, and to view them (e.g. with automatic capture of the structures and models to be mapped)
10. **Have Automated Validation**: This can go a long way towards defining and checking quality measures for the mappings.
11. **Be open and neutral, not proprietary**: It will probably be feasible for any supplier to convert mappings from a neutral notation to their own proprietary notation – but may not be possible the other way round

After the mapping language has been described, it will be compared with these requirements in section 9.

3. CORE MAPPING LANGUAGE

3.1 Principles: What Needs to be Mapped

The mapping language described in this paper is primarily a Structure-to-Model mapping language (S2M) although it can also be used to do Model-to-Model (M2M) mapping.

The mapping language needs to describe how an instance of a data structure (such as XML, or a relational database) describes an instance of a UML class model. The UML class model instance is a set of objects in different classes, each object having values for some of its attributes, and with links (instances of associations) between the objects. Frequently the objects and links will form a single connected graph, but this is not always the case.

A UML class model instance may be stored and manipulated in memory (for instance in a Java program) or stored in some persistent form. Exactly how this is done does not matter for the purposes of this paper; the ideas of mapping apply to whatever internal or persistent forms are used for UML class model instances. But for concreteness, some ways of storing UML class model instances are discussed in an Appendix to the paper.

In what follows I shall use the case of an XML instance representing an instance of a UML class model; other structures, such as Relational Databases, can be handled with the same principles.

The purpose of the XML instance is to convey information about objects in a UML class model instance. The XML instance (or document) represents, or partially defines, an instance of the UML class model (the definition is partial because, for instance, not all attribute values may be defined by the XML). I assume that the XML instance must at least define:

- The number of **objects** of each class, in the UML model instance
- Some **attributes** of each object which have defined values
- Some defined **links** between the objects – which are instances of the associations in the class model.

So it should be possible to say, on inspecting the XML instance: "This represents an instance of the UML class model, with 12 objects in classes X, Y, and Z, with 25 defined values of attributes, and with 37 links between the objects; ...".

The mapping language defines, in essence, how the XML does this. It makes declarative statement which can be used by an XML Reader (see below) to read any XML instance (which conforms to the XML structure) to create a UML instance (which conforms to the class model).

A set of mappings is always a set of mappings between one XML structure (defined by a schema) and one UML class model (defined in UML). If a piece of XML defines things in several different class models, several different sets of mappings are needed to say how it does it. Similarly if parts of one UML class model are represented by several different XML languages (a more common case), several sets of mappings are needed, to say how they do so.

It is not possible in a mapping language to describe all the different ways in which people might use XML to represent the information in a UML class model. The designer of some perverse XML language might require the use of some highly complex algorithm to create a UML instance from an XML instance in the language. The purpose of a mapping language is to describe the commonly-used ways of representing information – as used by designers of XML languages which are not perverse, and which (by design) only require simple algorithms to create a class model instance from an XML instance.

The design of the mapping language has drawn on the design of many non-perverse XML languages; so that the mapping language can capture the design patterns used in these non-perverse languages.

These XML languages all share a common approximate principle which can be called the **principle of local node representation**, and can be stated for objects, attributes and links:

- For every **object** represented in the UML instance, there is a focal **mapped node** in the XML instance
- For every **defined attribute value** represented in the UML instance, there is a focal **mapped node** in the XML instance
- For every **link between objects** represented in the UML instance, there is a focal **mapped node** in the XML instance

This principle does not imply that the relation between XML nodes and parts of the UML instance (objects, attribute values, or links) is always a 1:1 relation, although it frequently is. Exceptions to a 1:1 rule arise in different XML languages. Equally, it does not imply that one focal 'mapped node' does the whole job on its own; sometimes, even when there is one main mapped node, other nodes are also involved. Nevertheless, the approximation that there is a focal mapped node for any object, attribute value or link in the UML instance is a good starting approximation, and we then can address the exceptions as they occur, within the same framework.

The principle leads to three kinds of mapping:

- **Object Mappings**, which state in effect 'each node of this type in the XML (with 'node type' defined by an XPath from the root node) represents one object of this class in the UML model'
- **Attribute Mappings**, which state in effect 'each node of this type in the XML (with 'node type' defined by an XPath from the root node) represents the value of an attribute of one object in the UML model'
- **Association Mappings**, which state in effect 'each node of this type in the XML (with 'node type' defined by an XPath from the root node) represents a link (instance of an association) between one object of a first class in the UML model, and an object of a second class'

Each of the three kinds of mapping is a relation between a set of nodes defined by an XML node type (i.e. defined by its XPath from some root node) and a part of the class model (defined by its class, and possibly an attribute name or association name). So each mapping can be a row in a table of mappings, with columns to define the type of mapping, XPath, class, and so on. This is the core of the mapping language.

It can easily be seen that all three kinds of mapping are necessary. Without object mappings, you could not even count the objects in the UML instance; without attribute mappings, you could not even count the defined attribute values in the UML instance; and without association mappings, you could not even count the defined links between objects. So you could not define a UML model instance without all three kinds of mapping.

Association mappings will be new to some people. In some cases, association mappings are very simple – but they can sometimes turn out to be more complex, because they always link two objects together. While they are the most complex type of mapping, you cannot do without them; they are the skeleton which holds a UML class model instance together.

The mapping notation depends on the XPath standard – because XPath is the most appropriate way of picking out sequences of nodes (usually treated just as sets of nodes) in an XML instance, which all have

the same role in representing a UML class model instance. This is an empirical fact, based on the bulk of the XML languages that people have designed.

Relative XPaths between nodes, as well as absolute XPaths from the root, also play a role in the notation. To see this, consider attribute values. It is no use if the XML just tells you: "Some object has an attribute 'id' with value '134'; and another object has the same attribute 'id' with value '147'". There may be 5 objects and 12 values; you need to tie the correct attribute value to the correct object. This is done by defining the relative XPath needed to get from the node representing the object, to the node representing the attribute value.

In most cases, the required XPath is the shortest relative XPath connecting the two nodes, so the mapping language will take that shortest path as the default; but occasionally it is not, and you need to define the relative XPath explicitly. When you have to do this, the relative XPath always differs from the default shortest XPath only by having an 'apex node' which is higher in the XML tree than the shortest path; you then need only define the higher apex node.

So while the open mapping language depends on the XPath standard, you can get away with using only a small subset of the XPath standard when defining mappings. This subset is defined in an Appendix.

3.2 Mappable UML Models

There are some features of UML2 class models which we need to re-express in order to make mapping as simple as possible. This does not diminish the expressive power of UML class models, but sometimes requires that power to be expressed in slightly different ways. The two factors which need to be controlled are:

- In a UML model, an attribute may be multi-valued. For simplicity of mapping, we would like attributes to be single-valued. Any multi-valued attribute should be re-expressed as a one-to-many association to some object which has a single-valued attribute.
- In a UML model, the value type of an attribute is a Classifier, which may be either an elementary Data Type or a Class. Examples are the data type classes used in HL7 V3. In cases where the value type is a class, this should also be re-expressed as an association to objects of the class. These associations should eventually bottom out at simple single-valued attributes.

These constraints may require some re-expression of some UML class models, but do not reduce their expressive power. Any UML attribute which has maximum cardinality greater than one, or whose type is a complex type (i.e. another class, rather than an elementary date type) needs to be replaced by an association to the class which is the type, or to some new class which holds just one value of the attribute. This process needs to be continued until all UML attributes are single-valued and have simple types. This can always be done, leaving the class model just as expressive as it was before.

It would have been possible to avoid this need to re-express UML class models, at the expense of making the mapping notation more complex. If attributes are allowed to be multi-valued and have complex types, then attribute mappings would need to have all the power and complexity of association mappings, and more.

3.3 The Core Tabular Mapping Language

This section defines the core mapping language, which can capture a lot about how XML languages represent the information in class models, but does not deal with some important cases. Those cases will be addressed in section 4, on the extended mapping language.

The core tabular mapping language is a table with five columns: 'Type', 'XPath', 'Class', 'Feature', and 'Comments'. The last column is used to record any free-text comments about the mapping – particularly what cannot be mapped - and will not be discussed further.

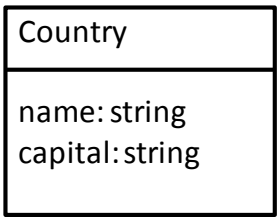
The order of rows in a table of mappings has no significance. The idea is that when viewing mappings, the rows can be sorted on any column or set of columns for convenience of viewing. For instance, sorting

by column ‘XPath’ helps to answer questions like: ‘what is this XML node or sub-tree mapped to?’; whereas sorting by column ‘Class’ helps to answer questions like: ’What is this part of the class model mapped to?’.

Similarly the order of columns has no significance – although people might adopt some standard ordering for familiarity.

The column ‘Type’ has only five possible values: ‘object’, ‘attribute’, ‘assoc’, ‘macro’ and ‘call’. The first three of these values define the three types of mapping. The values ‘macro’ and ‘call’ are used when dividing mappings into modules.

To illustrate the core mapping notation, consider a class model with just one class:



An XML language represents information in this class model. An instance of the XML looks like:

```
<?xml version=”1.0”>
<countries>
      <countryEl cName=”France” capCity=”Paris”/>
      <countryEl cName=”Germany” capCity=”Berlin”/>
<countries>
```

The table of mappings, which describes how the XML represents the class model, is as follows:

Type	**Class**	**Feature**	**XPath**	**Comments**
object	Country		/countries/countryEl	Only represents countries in Europe
attrib	Country	name	/countries/countryEl/@cName	Country name is anglicised
attrib	Country	capital	/countries/countryEl/@capCity	Capital city name is anglicised

Remarks about this mapping table:

- The first row, with Type= ‘object’, says that for every ‘countryEl’ element, there is one Country object represented. Without the following attribute mapping rows, you could say nothing else about that object – so you could do nothing more than count the objects represented by the XML.
- The next row, with Type = ‘attrib’, says that every XML attribute ‘cName’ represents the name of some country. Which country is it? The ‘shortest XPath’ default assumption in the mappings implies: it is the Country represented by the <countryEl> element with the shortest possible XPath to the ‘cName’ attribute. This means, it is the ‘countryEl’ element which owns that cName attribute (and not any other countryEl element)
- The same shortest XPath default applies to the other attribute mapping in the next row, ‘capital’. Applying the shortest path default to both attributes, means you cannot possibly interpret the XML instance as having a country ‘France’ with capital ‘Berlin’. Shortest XPaths tie attribute values to the correct objects.

- Since there are no associations in the UML model, there are no association mappings.
- The 'comments' column allows you to say anything that you cannot say elsewhere in the mapping row.

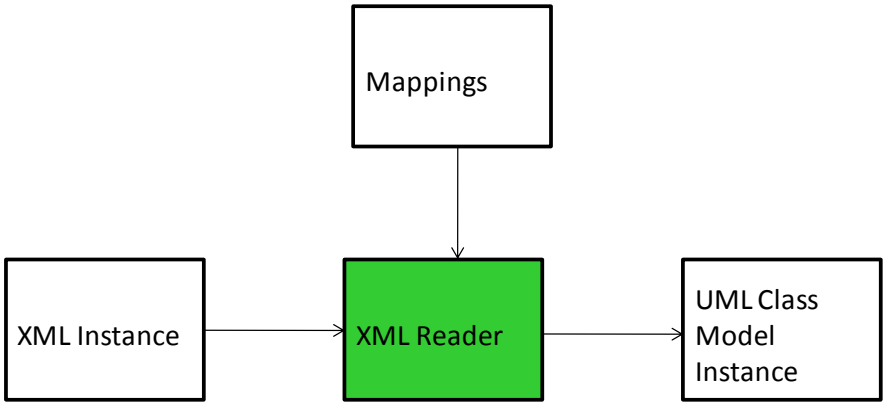
This example illustrates that there are a number of static validity conditions that the mappings must obey – for instance, all the XPaths must be valid XPaths which can lead to some nodes in the XML language, all the classes and attributes must exist in the UML model, and so on. The validity conditions will be described after the full mapping language has been described.

The three types of simple mapping row are described in sub-sections 3.5- 3.7.

3.4 XML Readers and XML Writers

The purpose of a mapping language is to state how some XML language conveys information in a UML class model. It defines the relationship between an instance of the XML language (i.e. an XML document) and an instance of the UML class model (a collection of objects, attributes, and association links between the objects).

Mappings are **executable**. This means it is possible to build software, driven entirely mappings, to extract pieces of class model information from the XML document. We will call the piece of software which does this an **XML Reader**, because it can read the class model information from a piece of XML. The operation of an XML Reader is shown in the diagram:



The XML Reader typically does not deliver the whole UML class model instance in one piece; a **selective** XML reader delivers partial information about the class model instance, on demand.

A selective XML Reader is typically called by some other piece of software which needs to have the class model information. The questions which the calling software can ask of the XML reader are of three kinds:

1. Return the set of **object tokens** for all the objects of some class represented in the XML instance (an object token is passed back to the calling application to stand for one object in a class, and the application passes it back to find out other things about the object, such as the values of its attributes)
2. Given an object token for an object in some class, find the value of some attribute of that object, which is represented in the XML instance.
3. Given an object token for an object in some class, return the set of object tokens for objects in some other class (or possibly, the same class) which are linked to the first object by links (instances of some association represented in the XML)

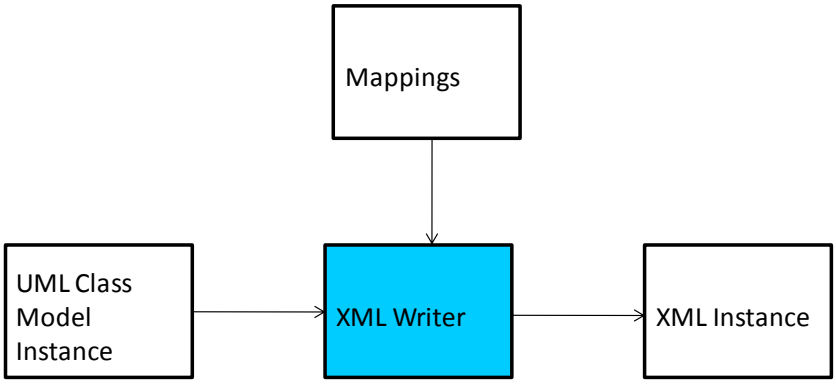
These three types of question involve respectively the three types of mapping – object mappings, attribute mappings, and association mappings. It is evident that by making a series of requests of this form, the

calling software could reconstruct the whole class model instance (all the objects, attribute values and links between objects) in some form of its own choosing.

It is usually straightforward to describe, from the definitions of the mappings, how an XML Reader can answer the questions (1), (2) and (3), using the mappings. This description gives a practical definition of what the mappings mean – i.e. of the semantics of the mapping language.

The object tokens which are passed back and forth between an XML Reader and its client application need consist of little more than the XML node which represents an object, and a definition of which class is being represented (as one node can represent objects of several classes at the same time). It is then fairly straightforward to develop a working XML reader; all you need to do is directly code the definitions of the mappings which follow, using some underlying XPath implementation to follow the XPaths in the XML document.

The software that goes in the opposite direction, from a class model instance to an XML instance, is called an **XML Writer** – because it writes a piece of XML from a class model instance. The mapping language should also be sufficient to drive an XML writer, and this consideration sometimes enters into the definition of the mapping language. The operation of an XML Writer is shown in the next diagram:



XML writers can be developed, but it is not nearly as straightforward to do so as developing an XML reader.

3.5 Simple Object Mappings

A simple object mapping is a statement that: 'Every node reached by some XPath from the root of the document represents one object of the defined class'. The uses of the five columns in simple object mappings are:

Column	**Usage**
Type	Must be 'object'
Class	Must uniquely define a class in the class model. This definition may be just the class name, if unique in the model; or if there are multiple packages, the unique designator may also need to include the package name. If the UML instance is a connected graph, it is allowed to identify the class by a path of association names from the entry class, followed by the final class name.
Feature	Empty
XPath	Must be a valid, pure descending, absolute XPath from the root node of the

	document, which may in some cases lead to one or more nodes in the XML as defined by its schema. Each XPath defines a set of nodes; this set must be capable of sometimes being non-empty. The XPath may contain predicates on nodes in the path steps, such as '(@val='F'); but for simplicity, it is recommended to use paths without predicates. Conditions can be expressed elsewhere in the notation. May use either the full form of XPath axes, such as 'child::' or 'descendant-or-child::' or 'attribute::', or the abbreviated forms '/' , '//', '@'. The short forms are preferred.
Comments	Free text

It is common in HL7 applications for the same class in the UML class model (such as the data type class 'II') to be represented separately several times in the same XML document, by sets of nodes with different XPaths. Each of these 'II' has a different meaning. In these cases, the different usages of the same class must be distinguished. The 'path of associations from the entry class' designation does this; but there are also other ways of doing so, discussed in the chapter on extended mapping notation.

Sometimes the nodes picked out by some simple XPath only represent objects of some class if they also satisfy other conditions. These other conditions could be stated as conditions on steps in the XPath. However, it is recommended in these cases to continue to use simple XPaths, and to state the conditions separately using the extended mapping notation discussed in the next chapter.

The path step '//' meaning 'descendant-or-child' is only appropriate when the XML language contains self-nesting of some elements, to indefinite depth. In these cases, the self-nesting usually represents an object in a class with an association to itself. The techniques for mapping such classes and associations are considered as part of the extended mapping language.

Note that there are cases where the same object of some class can give rise to several different nodes, all with the same XPath from the root, in the XML. In these cases, a count of the object nodes does not give a valid count of the number of objects represented. These cases are addressed by the extended mapping notation.

How does an XML Reader use object mappings? It starts from the root node of the XML document, and follows the XPath defined in the object mapping. This gives a set of nodes, each of which represents an object of the mapped class. These are called **object nodes**, and are then wrapped up as object tokens and passed back to a calling application. Without having any further mappings, for attributes and associations, that is all the XML Reader can do. All it can tell you is the size of the set of nodes; so it can tell you: "this piece of XML represents two objects of class 'Country'", and nothing more.

3.6 Simple Attribute Mappings

It is only possible to map some node type in the XML to an attribute of some class in the class model if the mapping set also has an object mapping to the same class. This follows because every attribute value represented in the XML must be a value of the attribute of some defined object; and we only consider cases where that object is represented in the same piece of XML as its attribute.

The uses of the five columns in a mapping table, for a simple attribute mapping, are shown below:

Column	**Usage**
Type	Must be 'attrib'
Class	Must uniquely define a class in the class model, using the same conventions as for

	object mappings.
Feature	The name of the mapped attribute in the class model. This attribute may be inherited from a superclass of the class in the 'Class' column.
XPath	Must be a valid, pure descending, absolute XPath from the root node of the document, which may in some cases lead to one or more nodes in the XML as defined by its schema. Described using the same subset of XPath as for object mappings
Comments	Free text

It is important to understand how an attribute mapping is used by an XML Reader, when using mappings to convert an XML instance into an instance of the UML class model. It is assumed that a represented object has already been discovered in the XML by navigating to one of the nodes that represents objects of that class, which is the object node; and the XML reader has returned an object token containing that object node. The attribute mapping is used to find out the value of some attribute as follows:

By inspecting the XPath in the object mapping, and the XPath in the attribute mapping, the XML Reader finds the shortest possible relative XPath between an object node and a node representing the attribute (which will be called an **attribute node**). The XML Reader navigates this shortest relative XPath from the object node. If this gives zero nodes, the attribute value is undefined. If it gives one node, the attribute value is the value of that node (i.e. the value of an attribute, or the text content of an element). If it gives more than one node, it is an error.

In most cases, the shortest relative XPath from an object node to an attribute node is a short XPath, and it is obvious that it can give at most one node (e.g. it is the path from an element to one of its attributes). If the relative XPath is capable in principle of giving more than one attribute node, then the simple attribute mapping is not valid (but see later for extended attribute mappings, which may have longer relative XPaths delivering more than one node, and conditions to narrow down to one node).

3.7 Simple Association Mappings

The notation only considers UML associations which link two objects of classes A and B – not the more complex associations which link three or more objects. These may be navigable in one or both directions - from an object of class A to find all the linked objects of class B, or from an object of class B to find all linked objects of class A. The association has a role name associated with each direction of navigation; or the role name is empty if the association is not navigable in that direction.

In the general case, each link (= instance of the association; possibly navigable in both directions) is represented by a node in the XML. A node representing a link is called a **link node**. The set of link nodes for some association is defined by an XPath from the root of the document.

The core mapping language can define how an XML language represents associations only under the following restrictive conditions:

- The association represented is between two different classes
- There are simple object mappings for both classes
- The cardinality of the association is 1:1 or 1:N.
- If it is 1:N, objects at the 'N' end of the association are represented by nodes nested inside the nodes representing objects at the '1' end of the association; or for 1:1 associations, both classes may be represented by the same node

In these restricted conditions, the node representing the 'inner' object always represents the link (the instance of the association) between the two objects. Link nodes and inner object nodes are identical, implying that there are exactly the same number of links as there are inner objects.

Consider an association between an ‘outer’ class A (represented some outer set of nodes) and an ‘inner’ class B (the nodes representing objects of class B are always nested inside the nodes representing objects of class A). So A is at the ‘1’ end of the association, and B is at the ‘N’ end; there may be several B objects for every A object. Assume the role name used to navigate from class A to class B is ‘toInner’, and the role name used to navigate from B to A is ‘toOuter’. The mapping table then contains two association mappings, one for each role name, as follows:

Type	**Class**	**Feature**	**XPath**	**Comments**
assoc	A	toInner.B	/AElement/BElement	Used to navigate from an A object to all linked B objects
assoc	B	toOuter.A	/AElement/BElement	Used to navigate from a B object to the one linked A object

In this example, objects of class A are represented by nodes with XPath ‘/AElement’, and objects of class B are represented by nodes nested inside those, with the longer XPath ‘/AElement/BElement’. Both association mappings are on the inner of the two elements; so both association mappings must always have the same XPath. The uses of the columns for simple association mappings are as follows:

Column	**Usage**
Type	Must be ‘assoc’
Class	Uniquely defines one of the two linked classes (outer or inner), using the same conventions as for object mappings.
Feature	The role name to get from the class defined in the ‘Class’ column to the other class (inner or outer), followed by ‘.’ and then the name of the other class. If there is no role name (the association is not navigable in this direction), the role name for the other end of the association is used, preceded by a ‘-‘.
XPath	For both the association mappings, must be the absolute XPath to the node representing the inner of the two classes (i.e. the longer of the two object mapping XPaths)
Comments	Free text

When the XML Reader has an object node representing an object of class A (the outer class) and is asked to retrieve all the linked objects of class B, following the role name ‘toInner’, it finds the shortest relative path from the nodes representing A to the nodes representing B. This is a pure descending path, and it follows that path from the object node to find object nodes representing objects of class B, and returns tokens for those objects.

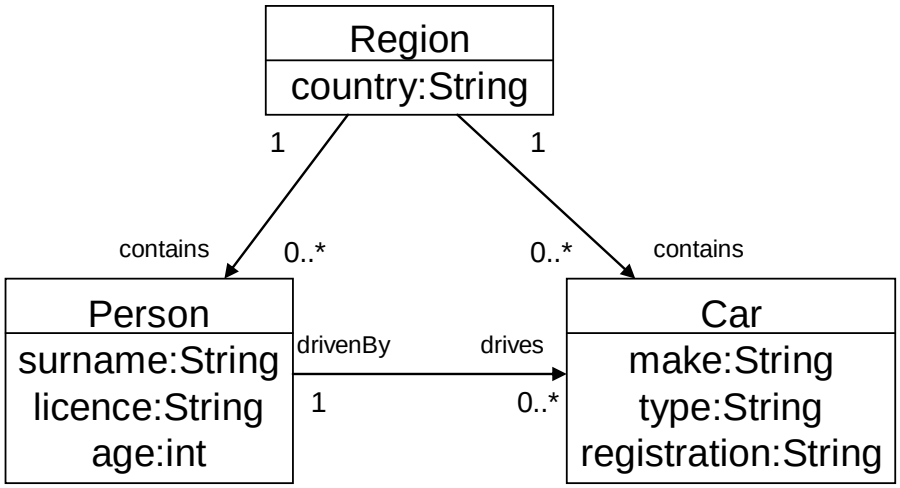
When the XML Reader has an object node representing an object of class B (the inner class) and is asked to retrieve the one linked object of class A, following the role name ‘toOuter’, it finds the shortest relative path from the nodes representing B to the nodes representing A. This is a pure ascending path, and it follows that path to find the one node representing an object of class A, and retrieves a token for that object.

So using a simple association mapping to retrieve all linked objects is simply a matter of navigating up or down the XML tree structure. This navigation may involve zero, one or more steps. Going upwards, it can only deliver one object node; going downwards, it may deliver any number of object nodes.

3.8 Example of Simple Mappings

This example uses a more complex class model than the previous example to illustrate object mappings, attribute mappings and association mappings being used together. It is taken from the 'Guided Tour' of the mapping tools available on the HL7 GForge at http://gforge.hl7.org/gf/project/v2v3-mapping/frs/ .

The UML class model is shown below:



The model is about people and the cars they drive. Every person and every car is contained in a region (which for simplicity is only identified by the country it is in), and every car has just one driver. Both cars and drivers have three simple attributes. We assume that drivers are uniquely identified by their driving licence numbers – the attribute 'licence'. Similarly cars are uniquely identified by their registration number.

There is a simple nested XML language which conveys information about instances of this class model. An example XML instance in this language is:

```
<?xml version="1.0" encoding="UTF-8"?>
<drivers country = "England" >
      <person name = "Black" license = "L12778" age = "45" >
            <drivesCar make = "Ford" type = "saloon" reg = "KLL788" />
            <drivesCar make = "Ferrari" type = "sports" reg = "PEW445" />
      </person>
      <person name = "Jones" license = "L4314" age = "28" >
            <drivesCar make = "Alfa Romeo" type = "sports" reg = "VBG998" />
      </person>
</drivers>
```

It should be evident from this example which elements and attributes convey which types of information. The association between people and cars (person.drives.car) is represented by nesting of the <drivesCar> element inside the <person> element. This simple representation of associations can be captured in the core mapping language, as described above.

The full set of mappings is shown in the table below (without any 'Comments' column):

Type	Class	Feature	XPath
attrib	car	make	/drivers/person/drivesCar/@make
attrib	car	type	/drivers/person/drivesCar/@type
attrib	car	registration	/drivers/person/drivesCar/@reg
attrib	person	surname	/drivers/person/@name
attrib	person	licence	/drivers/person/@license
attrib	person	age	/drivers/person/@age
attrib	region	country	/drivers/@country
object	region		/drivers
object	person		/drivers/person
object	car		/drivers/person/drivesCar
assoc	region	containsPerson.person	/drivers/person
assoc	person	pContainedBy.region	/drivers/person
assoc	car	drivenBy.person	/drivers/person/drivesCar
assoc	person	drives.car	/drivers/person/drivesCar
assoc	car	cContainedBy.region	/drivers/person/drivesCar
assoc	region	containsCar.car	/drivers/person/drivesCar

Note that there is one mapping row for each attribute, one row for each class, and two rows for each association.

The table illustrates that one node type in the XML (i.e. one XPath) can represent several different parts of the class model at the same time. For instance, nodes with path '/drivers/person/drivesCar' represent one class and two associations.

3.9 Could the Core Mapping Language Be Simpler?

The five columns of the core mapping language can define fully how certain restricted XML languages represent information in a class model. In these cases, the mappings give sufficient information to drive an XML Reader – that is, software which can take an instance of the XML and derive from it the class model instance, or answer specific questions about the class model instance.

It is hard to see how any structure-to-model mapping language could be any simpler than this, because:

1. The three different types of mapping (object, attribute, and association) are all necessary, to define how those three types of information in the class model are represented. The column 'Type' is needed to distinguish between the three types of mapping.
2. For all types of mapping, the notation must define what class of object is involved; the 'Class' column is necessary to do this.
3. There has to be a way of defining the sets of XML nodes which all represent some type of information. XPath is a standard and widely-understood way to define sets of nodes. So the 'XPath' column is a good way to define node sets.
4. For attribute and association mappings, it is necessary to define what attribute or association is represented. The 'Feature' column is needed to do this.
5. The 'Comments' column is needed to record comments and problems in the mappings.

In having these columns, the core mapping notation is little more than a formalisation of the informal spreadsheet notations that are often used to capture mappings. But it is a formalisation that works, for

building XML Readers. And it can be extended, as in the next section, to address the great majority of mappings we will need to describe.

Perhaps the only surprising feature about the core mapping notation is the need to have **two** association mappings with the same XPath for each mapped association – one for each end of the association. The need for two mappings per association will become clear when we consider how the extended notation captures different ways of representing associations.

4. EXTENDED MAPPING LANGUAGE

This section describes the extensions to the mapping language which are needed to capture the majority of ways in which XML documents and other data structures represent static class model information.

The extensions to the mapping language described below have all been found necessary to tackle practical mapping problems in healthcare. These extensions have been test-driven, by using the mapping tools available on the HL7 GForge site to make mappings, which have been tested by reading and writing XML instances. These mapping constructs are the result of the practical evolution of a mapping language. They are all needed; but suggestions for how they might be better packaged and presented will be welcome.

Extra columns for the tabular mapping language will be introduced as they are needed. The language and the uses of each column will be summarised in section 6 .

4.1 Mapping to Relational Databases and Text Files

4.1.1 Relational Databases

Mappings to a relational database define how an instance of the database represents a (typically large) instance of a UML class model.

In order to make mappings to a relational database, using a mapping language oriented to XML, you need to imagine that the whole database can be converted to a large XML document with the following structure:

- The root element has tag name ‘database’
- Immediately beneath the root element is one element for each table in the database. The name of each table element is the table name.
- Immediately beneath each table element are zero or more elements with tag name ‘record’. There is one such element for each record in the table.
- Immediately beneath each ‘record’ element, there is one element for each column of the table. The text content of this element is the value of the column for the record, in a text representation.

A small example of this relational database-style XML is shown below:

```
<database>
	<PEOPLE>
		<record><NAME>fred</NAME> <AGE>34</AGE> <D_ID>10</D_ID> </record>
		<record><NAME>joe</NAME> <AGE>55</AGE> <D_ID>11</D_ID> </record>
	</PEOPLE>
	<DEPT>
		<record><DNAME>accounts</DNAME> <DEPT_ID>10</DEPT_ID> </record>
		<record><DNAME>marketing</DNAME> <DEPT_ID>11</DEPT_ID> </record>
```

```
      </DEPT>
</database>
```

The mappings to the relational database are made as mappings to this XML structure, using XPaths such as '/database/PEOPLE/record/NAME'.

Then an XML Reader can be used to read information from the database and convert it to class model form. When doing so, it would usually be very inefficient to convert a whole database to one huge of XML document representing every record in every table. Instead, selective retrievals are made from the database. The results of those retrievals are expressed in the same XML form and are then passed to the XML Reader.

4.1.2 **Text Files**

It is necessary to convert any pure text data representation, such as HL7 Version 2, to an XML form so that its meaning can be defined by mappings. There may be some standard way of converting a text file to an XML form (such as the V2.xml standard for HL7 Version 2), or you may need to devise your own conversion from text to XML. This should be designed to be reversible (so that an XML writer could be used to convert a class model instance to an XML instance, and then to a text file instance). It should also be fairly fine-grained (i.e. with many XML elements each containing a small amount of text, rather than a few XML elements each containing a large amount of text), so as to make it easy to map.

Generally, the transformations needed to convert some non-XML form such as text to a mappable XML form (or even to convert some XML which is tricky to map into a more easily mappable XML) are called **wrapper transforms**. They are described in section 4.7.

4.2 Mappings and Inheritance

XML documents may represent information in UML models which have inheritance (subclasses and superclasses), including multiple inheritance. There may therefore be cases in which some node type in an XML document represents objects in one class or any of its subclasses; or represents an attribute of objects in a class and its subclasses, and so on.

The general approach of the mapping language is to be conservative about inheritance, requiring the mappings for a class and its subclasses to be made explicitly for each subclass if that is required. This conservative approach may require more explicit mappings to be made, but it gives finer control to the mappings. It allows the mappings to express whatever the designer of an XML language intended.

For instance, consider an XML language which represents two classes A and B, both subclasses of C. Classes A and B both inherit an attribute P from class C, and the XML language represents this attribute P. The designer of the language may want to represent attribute P in the same way for objects of both classes A and B; or he may want to represent attribute P differently for the two classes. The conservative approach, where attribute mappings for P need to be expressed separately for classes A and B, allows the mappings to express the difference.

This leads to the rule: If there are only object mappings to classes A and B, then all attribute mappings can only have classes A or B in their 'Class' column; they may not mention a common superclass C, unless that class is also explicitly mapped.

Similarly, all association mappings can only have classes A or B in their 'Class' column, and can only have classes A or B after the role name in their 'Feature' column.

When an XML Reader has an object node of class A, and is looking to find the values of the attributes of that object, or to find objects by navigating associations from that object, it will only use mappings which have the class A in their 'Class' column; it will not use mappings which have some superclass of A in their class column.

However, it must be remembered that each XML language may convey only partial information about a class model. So when some objects belong to a class A, there may be some XML language L_1 which represents object as being in that class A, and some other XML language L_2 which only represents the same objects as being in some superclass C. For instance, L_1 may be about cars and motor-cycles, whereas L_2 is only about vehicles. An object in some class can always be treated as an object in its superclass; cars are always vehicles. Then, using the mappings, it will be possible to translate from L_1 to L_2, but not in the reverse direction. For any vehicle in L_2, you do not know if it is a car or a motor-cycle, or neither of these; so you cannot represent it in L_1.

This means that when an XML Reader is asked to retrieve all objects of some class, either directly from its object mappings, or by using association mappings to navigate an association from some other object, it will look at all the mappings for that class **and all its subclasses** – because instances of any subclass are also instances of the class.

Generally it is recommended that the same set of mappings should not contain any mappings for a class and for some of its subclasses – because the result is likely to be confusing. But sometimes it may be necessary.

4.3 Extended Object Mappings

4.3.1 Conditional Mapping

Consider the following fragment of XML:

```
<fleet>
      <vehicle type=”car” cReg = “kpy733”/>
      <vehicle type=”car” cReg = “ju4332”/>
      <vehicle type=”van” vReg = “kog995”/>
<fleet>
```

If this XML were representing a class model which had only the class ‘Vehicle’, then we could express how it does so with a simple object mapping; all nodes with the XPath ‘/fleet/vehicle’ from the root node represent a Vehicle. However, if the class model also contained the classes ‘Car’ and ‘Van’ we would not be able to capture it with a simple mapping, because some nodes with the root XPath ‘/fleet/vehicle’ do not represent a Car; they represent a Van.

The nodes then have to obey some extra condition before we can be sure that they represent a Car or a Van. This extra condition can be evaluated by following some relative XPath from the candidate object node, finding the value of that node (the value of an attribute, or the text content of an element) and comparing it with some fixed value. The test used for comparison is usually equality, but it need not be.

We do this by introducing an extra column ‘Condition’ in the mapping table. The left-hand-side of the condition is the relative XPath to follow from the candidate object node, in square brackets; and the right-hand side is the constant to compare it with, in single quotes if it is a String (or not, if it is an integer).

Using the new column, the mappings for the XML above would be expressed:

Type	Class	Feature	XPath	Condition
object	Car		fleet/vehicle	(@type) = ‘car’
object	Van		fleet/vehicle	(@type) = ‘van’

Note:

- This kind of condition is known as a ‘value condition’ because it compares the value at some node against a constant value. There is another kind of condition, called a ‘cross-condition’ which compares the values on two different nodes; that will be introduced later. Cross-conditions are not used in object mappings.
- There can be more than one value condition on an object mapping; if so, the different conditions are separated by semicolons. The node only represents an object of the class only if it passes all the conditions; a logical AND is taken (if a logical OR is needed, two or more mappings will do the job)
- The relative XPath on the left-hand-side of the condition must be guaranteed to deliver at most one node in its node set – because a unique node value is needed to test the condition. Otherwise it is an error in the mapping.
- If the relative XPath delivers no nodes, then the value used in the condition is the empty string, denoted by ‘’. This would, for instance, allow you to map a ‘vehicle’ node with no ‘type’ attribute to the superclass ‘Vehicle’.
- The tests used to compare the two sides of the condition can be ‘=’, ‘!=’ (for strings or numbers), ‘<’, ‘>’, ‘=<’ or ‘>=’ (for numbers), and ‘contains’ or ‘containsAsWord’ for strings. The last test looks for a substring, separated from the rest of the string by separators ‘ ‘, ‘,’,’-‘, ‘_’ or ‘.’, which matches the constant string on the right-hand side.
- The relative XPath on the left-hand side may have one ascending step (with axis ‘parent::’ or ‘ancestor::’) preceding its descending steps.

These value conditions have an interesting effect on an XML Writer. If an object mapping has a value condition with an equality ‘=’ comparison, then when an instance of the XML is written from an instance of the object model, the nodes necessary to satisfy the condition are automatically inserted in the XML instance. For instance, in the example above, to represent one car object in the XML, an XML Writer would not only create a <vehicle> element, but it would also give it an attribute type=”car”.

4.3.2 **Multiple Mappings to the Same Class**

It can happen that several different node types (characterised by different XPaths) all represent objects of the same class, but that these objects play different roles in the class model instance – for instance by being associated to different objects of some other class, or by having different attributes represented.

In these cases it is necessary to distinguish between the different object mappings to the same class. This distinction needs to be made not only in the object mappings themselves, but also in attribute mappings and association mappings, so that one can always know (for instance) that ‘this attribute mapping denotes an attribute of the object defined by that object mapping’. Every attribute mapping must be tied uniquely to one object mapping; and every association mapping must be tied uniquely to two object mappings, one for each end of the association.

To do this, the idea of **subsets** is used. A subset is an identifier for a set of objects within a class – all having the same object mapping. The subset is identified by a string (typically short) which must to be unique within the class and within the mapping set, and which appears in brackets after the class name, in an object mapping, attribute mapping or association mapping. If there is no bracket following the class name, the subset is taken to be the empty string ‘’, which is distinct from any other subset.

An XML Reader uses subsets as follows:

- When asked to find the value of some attribute of an object, the XML Reader looks for an attribute mapping with the same subset as the object mapping or association mapping which was used to retrieve the object.
- When asked to follow an association from an object, the XML reader looks for an association mapping with the same subset as the object mapping or association mapping which was used to retrieve the object.

Subsets can be regarded as if they were subclasses which are visible in the XML, but not in the class model.

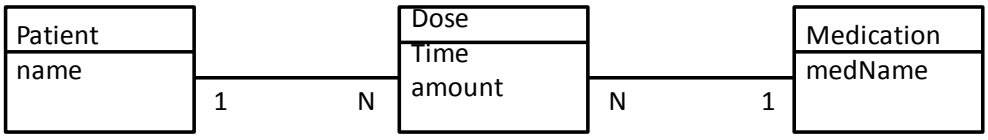
The table in the next sub-section shows an example of subsets. Here, objects of the class ‘Car’ are represented by two separate sets of nodes, with different XPaths, so these two object mappings are distinguished by a subset – which has values ‘’ (no bracket) and ‘s1’. These subsets are then reused in the two attribute mappings for the attribute ‘make’, to tie each attribute mapping to its correct object mapping.

It was mentioned earlier that when there are several distinct mappings to the same class, the different occurrences of the same class can sometimes be distinguished by a path of association role names from some entry class to the whole mapping set. It is often convenient to show these association paths in the ‘Class’ column of object mappings for clarity. However, even when two occurrences of the same class are distinguished by having different association paths, it is necessary also to distinguish them by a subset – so that the full association path does not need to be used in all attribute and association mappings.

4.3.3 **Non-Unique Mappings**

When tree-like XML structures are used to represent UML class models whose associations are not a tree of 1:N associations – particularly where there are N:M associations – the XML tree may contain nodes representing objects, such that one object in the class model instance corresponds to several nodes in the XML tree. Then, the count of objects and the count of XML nodes are not equal; there may be more nodes than objects. This may happen when all the XML nodes have the same XPath, used in just one object mapping. Then the object mapping is said to be **non-unique**.

As an example, consider the following class model:



This represents a situation in which one patient has many doses of the same medication, or of different medications (association names and data types have been omitted for simplicity). In XML, this may be represented by a tree in which the top node represents the patient, and child nodes represent both doses and the medication being taken. A simplified sample of the XML may look like:

```
<patient name=”Fred”>
      <dose date=’040610’ qty=’2’ unit=’pill’ med=’aspirin’ />
      <dose date=’140610’ qty=’1’ unit=’pill’ med=’aspirin’ />
      <dose date=’140610’ qty=’30’ unit=’mg’ med=’insulin’ />
</patient>
```

Many child ‘dose’ nodes may all represent the same medication; or they may represent doses of different medications. One may want to answer the question ‘how many different medications is the patient taking’?

If any object mapping is non-unique, it is indicated by a non-empty value in another new column, ‘Key’. This column denotes what combination of attributes, represented in the XML, constitutes a unique identifier (or key) for every object. If, for instance the medication name attribute ‘medName’ constitutes a unique identifier for a medication, the Key column simply contains ‘medName’. If a combination of more than one attribute is needed to uniquely identify an object, the attribute names are separated by semicolons. All key attributes must have mappings and be represented for the class (and subset) in the XML.

The object mappings for the XML example shown above, and the one required attribute mapping, are as follows:

Type	Class	Feature	XPath	Key
object	Patient		/patient	
object	Dose		/patient/dose	
object	Medication		/patient/dose	medName
attrib	Medication	medName	/patient/dose/@med	

(Association mappings have been left out of this example)

Note that the same nodes with XPath '/patient/dose' represent objects of two classes, Dose and Medication. Using these mappings, an XML Reader can return all the object tokens for class 'Medication', including several tokens that all represent the same object; but it can also look at the key attribute 'medName' to work out that several of the tokens refer to the same medication 'aspirin'. Thus it can identify or count the distinct objects of class 'Medication' represented.

When one object is represented non-uniquely by more than one XML node, it is in principle possible for some attributes of the object to be represented close to one of its object nodes, and some other attribute to be represented close to some other object node for the same object. The XML Reader might in principle need to examine all the object nodes for one object to find out the value of some attribute – or different object nodes might even give inconsistent values for the same attribute of the same object. Similarly associations may be represented in complex or error-prone ways. This is similar to a non-normalised relational database being capable of inconsistencies – and should be rare in well-designed XML languages.

4.3.4 **Inclusion Filters**

In most cases where XML represents objects in some class, it is obvious that the XML does not represent **all** objects in the class; it only represents selected objects, which obey certain filters. For instance, a patient record will only contain medication records for those medications being taken by the patient.

This affects the operation of an XML writer, because the XML writer needs to know just which objects to write from a class model instance to an XML instance. A class model instance may contain large numbers of objects, many of which are not to be written to some XML instance. Therefore there needs to be some way of expressing this in the mappings, so that XML writers can operate properly.

It also effects the operation of an XML Reader, as described in section 4.3.5 below.

Inclusion filters for objects are described in another new column, 'Filter'. The filters can be of three main types, in that:

1. Some attribute of the object must have a certain value; this is denoted by "attribute = 'value'". The attribute must be represented in the XML. Value inequalities are allowed. For instance, a data structure might represent all patients with age greater than 30 years.
2. Some attribute of another 'lower' object, reached from the object by some chain of associations, must have some value. The chain of associations must lead to only one object (to give a unique value to test).In this case, there must be a chain of association and object mappings leading to a unique object which has the filter on its attribute; and the filter is expressed in that object mapping.
3. The object must be reachable by some association, from some other 'higher' object which is also included in the XML instance. This is denoted by "role FROM class", where 'class' is the class name (and subset) of the other object which must also be in the XML instance, and 'role' is the role name of some association, to get from the other object to this object. The association must be represented in the XML.

If there is more than one inclusion filter, they appear in the 'Filter' column separated by ';' and their logical AND is applied.

The third kind of inclusion filter is quite common in XML languages. Objects represented by child nodes can only be included in an XML instance, if the object represented by the parent node is also included. The association from the parent object to the child object may typically be a composition.

Extending the previous example to add the required inclusion filters and association mappings gives the result:

Type	Class	Feature	XPath	Filter	Key
object	Patient		/patient		
object	Dose		/patient/dose	takesDose FROM Patient	
object	Medication		/patient/dose	ofMed FROM Dose	medName
attrib	Medication	medName	/patient/dose/@med		
assoc	Patient	takesDose.Dose	/patient/dose		
assoc	Dose	takenBy.Patient	/patient/dose		
assoc	Dose	ofMed.Medication	/patient/dose		
assoc	Medication	inDose.Dose	/patient/dose		

4.3.5 **Filters May Imply Fixed Attribute Values**

If an object mapping has an equality filter on some attribute – so the XML only represents objects with a defined value of the attribute – then it follows that the value of the attribute is fixed and known for every object of the class represented in the XML. So equality filters on attributes imply a kind of fixed–value attribute mapping, wrapped up in the object mapping.

An example is given below:

```
<fleet>
       <carAuto make="Nissan"/>
       <carAuto make="Ford"/>
       <carManual make="Vauxhall"/>
</fleet>
```

Type	Class	Feature	XPath	Filter
object	Car		/fleet/carAuto	transmission='automatic'
object	Car(s1)		/fleet/carManual	transmission='manual'
attrib	Car	make	/fleet/carAuto/@make	
attrib	Car(s1)	make	/fleet/carManual/@make	

Here, the attribute 'transmission' of the class 'Car' is given two different filter values, for different nodes representing cars. These filters imply fixed values of the attributes concerned, which will be picked up by an XML Reader asked for values of those attributes. It is also given variable values of the attribute 'make'.

The (s1) after the class name is an example of a subset, described in the previous sub-section.

Object mappings can specify fixed values for inherited attributes.

4.4 Extended Attribute Mappings

4.4.1 Conditional Attribute Mappings

Attribute mappings can depend in two different types of condition. Consider the following fragment of XML:

```
<fleet>
       <car>
              <pVal attName=”make” value=”Fiat” />
              <pVal attName =”carReg” value=”PEW373K” />
              <pVal attName =”gears” value=”6” />
       <car>
       <car>
              <pVal attName =”make” value=”Cadillac” />
              <pVal attName =”carReg” value=”JYK442” />
              <pVal attName =”capacity” value=”446” />
       <car>
<fleet>
```

This is an open-ended style of XML language, which can represent any attribute of a ‘Car’ object, just by using a <pVal> element with the appropriate value of the attribute ‘`attName`’ to pick out which attribute is represented. It is not possible to tell from the XML Schema of this language what attributes it is capable of representing. Using just the XPath from the root to any ‘value’ attribute, you cannot tell what attribute it is the value of.

To define what attribute is represented by each ‘value’ attribute, we again use the ‘Condition’ column in the mapping table, to express a value condition on an attribute mapping. This is done exactly as it was for conditional object mappings. The left-hand side of the condition is a relative XPath from the mapped node, and the right-hand side is some constant value.

Using the ‘Condition’ column, the mappings for the XML above would be expressed:

Type	Class	Feature	XPath	Condition
object	Car		fleet/car	
attrib	Car	make	fleet/car/pVal/@value	(../@attName) = ‘make’
attrib	Car	registration	fleet/car/pVal/@value	(../@attName) = ‘carReg’
attrib	Car	gears	fleet/car/pVal/@value	(../@attName) = ‘gears’
attrib	Car	capacity	fleet/car/pVal/@value	(../@attName) = ‘capacity’

All the different attribute mappings have the same XPath – leading to the attribute ‘value’ – but they are distinguished by their mapping conditions.

From each ‘value’ attribute, the XPath on the left-hand side of the condition has an initial step ‘..’ which is an abbreviated form of ‘parent::node()’, picking out the node which owns the attribute. It says: ‘go up to the parent node; come down to the attribute ‘attName’ and test its value. Depending on the value of the attribute ‘attName’, each attribute mapping represents a different attribute of the class ‘Car’. Which car it represents that attribute for, is determined by the closest <car> node representing a car, by the shortest path default, as for any other attribute mapping

Note the following points about attribute mapping conditions (just as for object mapping conditions):

- There can be more than one value condition on an attribute mapping; if so, the different conditions are separated by semicolons. The node only represents the stated attribute of the class only if it passes all the conditions; a logical AND is taken.

- The relative XPath on the left-hand-side of the condition must be guaranteed to deliver at most one node in its node set – because a unique node is needed to test the condition. Otherwise it is an error in the mapping.
- If the relative XPath delivers no nodes, then the value used in the condition is the empty string, denoted by ‘’. This is rarely used for attribute mappings.
- The tests used to compare the two sides of the condition can be ‘=’ ,’!=’ (for strings or numbers), ‘<’, ‘>’, ‘=<’ or ‘>=’ (for numbers), and ‘contains’ or ‘containsAsWord’ for strings.
- The relative XPath on the left-hand side may have an ascending step before its descending steps.
- An attribute mapping with an ‘=’ condition will lead an XML Writer to insert the relevant node into the XML instance, when writing the attribute. For instance, in the example above, the XML writer will insert the XML attribute attName=’make’.

There is a second kind of condition which can be used for attribute mappings, known as a **cross-condition**. This is illustrated by the following example of XML:

```
<fleet>
       <car reg = value=”PEW373K” />
       <car reg = value=”JYK442” />
       <props>
              <prop regVal= “PEW373K” make=”Ford” capacity = “2000”/>
              <prop regVal= “JYK442” make=”Renault” capacity=”1600”/>
       <props>
<fleet>
```

In this example, some attributes of a car are held close to the <car> node which represents the car; but others are held remotely, in a ‘dustbin’ of attributes for any car under the <props> node. In these cases, how can we ensure that some attribute value is tied to the correct instance of ‘Car’ ?

The attribute values are tied to the correct instance of ‘Car’ by the value of ‘regVal’ (for the element containing the attribute values) matching the value of ‘reg’ (for the element representing the car). So we need a condition which links a value close to the attribute node (representing the value of some attribute) to a value close to an object node (representing the object), to tie the value to the correct object. In both cases, ‘close to’ is defined by some (typically short) XPath. These types of conditions, which relate the values close to two different nodes, are called **cross-conditions**.

The form of a cross-condition for an attribute mapping is typically

(XPath 1, from attribute node) = (XPath 2, from object node)

(while in principle, comparisons other than ‘=’ can be used, they are in practice rarely used). So the mappings used in the example above are:

Type	**Class**	**Feature**	**XPath**	**Condition**
object	Car		/fleet/car	
attrib	Car	registration	/fleet/car/@reg	
attrib	Car	make	/fleet/props/prop/@make	(../@regVal) = (@reg)
attrib	Car	capacity	/fleet/props/prop/@capacity	(../@regVal) = (@reg)

We can follow through how an XML Reader would use the attribute mapping for ‘make’ to find the make of the car with registration ‘PEW373K’. The steps are:

1. Start from the <car> node representing that car (the object node).

2. To be ready to evaluate the mapping condition, follow the XPath on the right-hand side of that condition '@reg' from the object node <car> and find the value of the node. That value is 'PEW373K'.
3. Find the shortest possible XPath from object nodes representing objects 'Car' to the attribute nodes representing the attribute 'make'. This XPath goes up to the root node <fleet> and down the path 'fleet/props/prop/@make'.
4. Follow that XPath from the object node. This gives two 'make' nodes.
5. For each of the two 'make' nodes, test the cross condition. This means: follow the left-hand side XPath '../regVal' from the attribute node 'make' and find the value. Compare that value with the right-hand side value 'PEW373K' found in step (2). Only if the values are equal, keep the 'make' node. This leaves only one 'make' attribute node.
6. Find the value of that one 'make' node – which is 'Fiat'. That is the value of the attribute.

In summary, after following some XPath from the object node to the set of candidate attribute nodes, the cross-condition is applied to reduce the set of candidate nodes down to one node, whose value is the attribute.

For large XML documents, this 'brute force' implementation can be very inefficient. It repeatedly follows a long XPath to find a large node set, and filters down this node set by serial search though it. However, this inefficient implementation is not the only way to build an XML Reader. It is also possible to 'pre-index' nodes according to the values they provide for cross-conditions, and then efficiently pick up only those nodes which are going to pass the condition. This is just like using an index in a database.

Cross-conditions, like value conditions, are used by an XML Writer. When the cross-condition is an equality, the XML writer will write both nodes needed to satisfy the condition, and will populate them with equal values. If neither node is mapped to any attribute (which fixes the value), this equal value for the two nodes will be an arbitrary string, made up by the XML Writer.

Cross-conditions obey the same general constraints as value conditions, and some extra constraints. In particular:

- One attribute mapping may have several cross-conditions, or even a mixture of cross-conditions and value conditions. They all appear in the 'Conditions' column separated by ';' and are evaluated using a logical 'AND'.
- The XPaths on both sides of a cross-condition must be guaranteed to lead to at most one node – to give a unique value to test - otherwise the condition is not valid.
- The tests 'contains' and 'containsAsWord' can sometimes be useful in cross-conditions.

4.4.2 **Subsets in Attribute Mappings**

The same XML language may represent objects of the same class in several different places, for several different purposes – which results in several object mappings to the same class. When this happens, the different object mappings are distinguished by a 'subset', which is a unique string written in brackets after the class name in the 'Class' column. A typical example is 'Person(s1)', 'Person(s2)' and so on.

In this case, every attribute mapping for an attribute of the class must identify exactly which of the different object mappings it relates to. Therefore in every attribute mapping, the class name in the 'Class' column must have a subset identifier, if necessary.

The default subset identifier when none is given is the empty string ''.

4.4.3 **Single Attribute Value Conversions**

XML languages and relational databases frequently define the values of attributes in the class model not directly, but through other values that need to be converted into the values used in the class model (when reading into an instance of the class model) or in the reverse direction (when writing from an instance of the class model). These are attribute value conversions, and need to be specified in the mappings.

There are two kinds of attribute value conversions – 'in' conversions to go from the XML to the class model, and 'out' conversions to go back again. They are specified in two separate columns for an attribute mapping, 'Convert_In', and 'Convert_Out'.

Many attribute conversions are complex in nature and need to be done with procedural code, for instance in Java or XSLT. Furthermore, there may be no convenient declarative way to specify what the conversion does; so effectively, the conversion is specified only by reference to a piece of procedural code. In that sense, some attribute value conversions are examples of the procedural escape mechanisms discussed later in this paper; but they are described here as a part of extended attribute mappings.

There is an obvious requirement for consistency between the in-conversions and the output-conversions – that in combination, they should make a self-consistent round trip. Converting in and then converting out should get you back to the value you started from. However, when attribute value conversions are made and specified by procedural code, there is no way to test this round-trip consistency other than by running or examining the code; nothing more is said about it here.

Both in-conversions and out-conversions can be further divided into two types: **single** attribute conversions and **multiple** attribute conversions. A single attribute conversion is one in which the value of a single node in the XML is converted to a single attribute in the class model, and vice versa. A multiple attribute conversion is anything else – where two or more values from the XML must be combined to make one value in the class model, or vice versa. An out-conversion can be single if and only if the corresponding in-conversion (in the same attribute mapping) is single.

Multiple attribute conversions are discussed in the next sub-section.

If a single attribute in-conversion is specified and implemented only as a Java method, then that method must have only one argument of type String – which will be set to the string value of the attribute node – and delivers a result of type String, which then may need to be converted to the data type of the attribute in the UML class model. Similarly, Java methods for single out-conversions must take one String argument and deliver a String result.

In the 'Convert_In' or 'Convert_Out' column, the language name 'Java' is followed by the package name, class name and method name of the Java method, separated by '.'. This can be followed by '--' and any comment.

Single attribute conversions can be declared in other languages such as XSLT, using an appropriate unique designator for the template or method. If a conversion is provided in more than one language, then the declarations for different languages can follow one another, separated by ';'. Any comment must come after the declarations of the conversion in different languages.

It may not be necessary to define the conversion procedurally, if it can be defined in a table of value pairs. The attribute mapping may then refer to the conversion table by the keyword 'table' followed by some unique designator for the table, such as the name of the file it is stored in. When attribute value conversions are defined in a table, the form of the table is as in the following example:

Structure Value	**Model Value**	**Details**
1	M	-- This means 'male'
2	M	-- This means 'female'
U	3	-- This needs more precise definition of the mapping

In this example, the ‘Details’ column has only been used to make textual comments on the mappings; but in future the Details column may contain structured elements before any ‘--‘ comment, to address various common mapping issues. In particular, when two or more distinct values in the ‘Structure Value’ column all map to one value in the ‘Model Value’ column, the conversion cannot be run unambiguously in the ‘out’ direction; so for these cases, there could be some structured value in the ‘Details’ column saying how to resolve the ambiguity. The form of such structured values in ‘Details’ is not yet defined.

Because there are separate columns for ‘Convert_In’ and ‘Convert_Out’, it is possible in principle to use one table to convert inwards, and another table to convert outwards. Generally one would not expect people to do this, as it removes any guarantee of round-trip consistency.

Some examples of attribute mappings with single conversions are given in the following table:

Type	Class	Feature	XPath	Convert_In	Convert_Out
attrib	Car	type	fleet/car/@typeCode	table typeCon.xml	table typeCon.xml
attrib	Car	colour	fleet/car/@colCode	-- still to be worked out	
attrib	Car	capacity	fleet/car/@capacity	Java cPack.Conv.ccTocubicInch	Java cPack.Conv.cubicInchToCC – round trip tested

In this example, the Java package is ‘cPack’, the class is ‘Conv’ and the two methods are ‘ccToCubicInch’ and ‘cubicInchTocc’. They both take single String arguments (which are string representations of real numbers) and return Strings.

4.4.4 Multiple Attribute Conversions

A multiple attribute conversion occurs whenever it is necessary to combine the values from several nodes in the XML to find the value of one attribute in the UML, or when it is necessary to combine values of several attributes in the UML to find the value of one node in the XML. In these cases, if the conversion is defined in some procedural language such as Java, the Java method for the in-conversion must have more than one argument, or the Java method for the out-conversion will have more than one argument.

Multiple attribute conversions are distinguished from single attribute conversions by the explicit listing of the arguments of the conversion methods, in brackets after the method names. As for single attribute conversions, all arguments of the conversion methods are strings, and the results of conversion methods are strings. There is currently no means in the language to specify multiple attribute conversions in table form. There is no means to pass a variable number of arguments to a conversion method, for instance as an array.

In general, multiple attribute conversions have greater potential for being irreversible than single attribute conversions. For instance, if the class model stores a person’s name in several parts – first name, surname, and so on- whereas an XML language stores the same name as a single string, the out-conversion may be done by a simple concatenation of strings, whereas the in-conversion may be more problematic – as it is hard to know where to split the full name string, and which name part is which. However, as noted above, even single attribute conversions have the potential to be irreversible, so no new issues of irreversibility are really raised by multiple conversions.

In-conversions may require as arguments the values of some XML nodes which have not been mapped to any attribute. In these cases, the value is got by an attribute mapping to a ‘pseudo-attribute’ which is not in the class model. Pseudo-attributes are given names which have a final ‘p’ in brackets.

The following example shows where the XML stores a given name and a surname, whereas the class model stores only a full name:

Type	Class	Feature	XPath	Convert_In	Convert_Out
attrib	Person	fName(P)	/Person/@first		
attrib	Person	sName(P)	/Person/@surname		
attrib	Person	fullName		Java cPack.conv.getFullName(fName,sName)	-- not yet possible to split names

The in-conversion method has two arguments, which are taken from the two ‘pseudo-attribute’ mappings above. It has no XPath itself.

The reverse case where the UML class model stores the first name and surname separately, and the XML data structure concatenates them into a single name, is shown in the next table:

Type	Class	Feature	XPath	Convert_In	Convert_Out
attrib	Person	fullName(P)	/Person/@fullName	-- no target attribute	Java cPack.conv.getFullName(firstName,surName)

In this case, there is only one pseudo-attribute mapped in the XML. Its value is calculated from two attributes in the UML – firstName and surname – which are not mapped because there is no XML node to map them to. They are assumed to be attributes of the same class, ‘Person’, but may be inherited.

If there were a Java method which could ‘unpack’ a surname from a full name, it could be used in a second attribute mapping in the second table.

4.4.5 **Longer Relative XPaths**

To tie the value of an attribute to the correct object, the default rule is always: “Follow the shortest possible XPath from the object node to the attribute node”. This rule nearly always works for attribute nodes. However, there are rare cases where the attribute node is more remote from the object node; so following the shortest possible XPath would not reach it.

In those cases, it is necessary to specify some longer relative XPath from the object node to the attribute node. In practice, all that needs to be specified is the ‘highest’ node, closest the root of the document, which this XPath needs to pass through. This node will be called the **apex** of the XPath, and it is specified in a new column ‘Apex’, which defines the apex node by its XPath from the root. If it is unambiguous, this apex XPath can be of the form ‘//NodeName’, which picks out as apex any node of name ‘NodeName’ in the document; otherwise a fully specified XPath from the root may be needed to define it unambiguously.

The specified apex node must be closer to the root than the apex node of the default shortest XPath. The effect of this is to enlarge the set of attribute nodes reached by the path, rather than to diminish it.

Longer relative XPaths than the default are rarely used for attribute mappings, but are used more frequently for association mappings. An example use of the ‘Apex’ column will be given later for an association mapping.

4.4.6 **Alternate Attribute Mappings**

It sometimes happens that the same attribute of objects of the same class is represented more than once in the XML. This can be an unfortunate design feature of the XML language, because it means that different versions of the same attribute value, for the same object, may be inconsistent with each other. Nevertheless it occasionally happens. When it does, it can happen in one of two ways. For an attribute which is represented just twice, these ways are:

1. The program which wrote the XML is forced to write out both representations of the attribute value; then an XML Reader can choose either version to find the value of the attribute.
2. The program which wrote the XML may have been able to choose which representation of the attribute value to write out, but need not write both. Then an XML Reader is forced to look at both representations, to see if either of them gives it a value.

These two possibilities are defined by a new column ‘Alternates’, whose values can be ‘all’ (for case (1), where all the representations must be present) or ‘some’ (for case (2), where only some of the representations must be present). Usually, the ‘Alternates’ column is not needed, and is empty.

4.5 Extended Association Mappings

Extended association mappings are perhaps the most novel and unfamiliar feature of the mapping language. There is a single notation for association mappings, which covers both the most general cases and simpler cases; so it is worth understanding properly what the most general association mappings do.

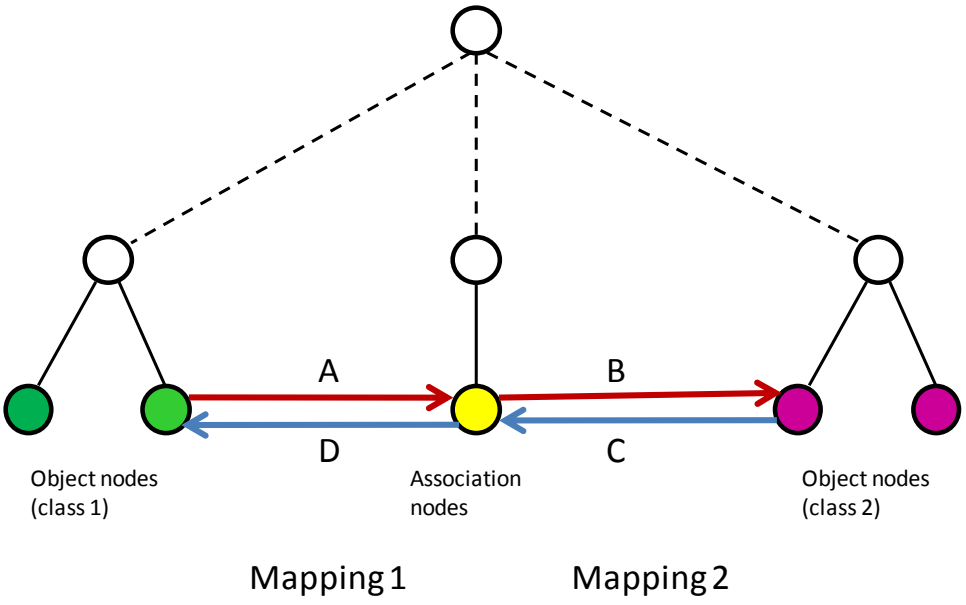
I shall describe the principles of how associations are mapped in the general case, then illustrate how these principles work out in an example of the most general case, then describe how they work in special cases.

4.5.1 Conditional Association Mappings

The general principles of mapping associations are:

- Every link (every instance of an association) is represented by a node, which is called the **association node**. The set of association nodes is defined by its XPath from the root of the document.
- There are always two association mappings for every association – one for each end of the association. These describe respectively (a) how to traverse in either direction between object nodes at one end of the association and the association nodes; and (b) how to traverse in either direction between object nodes at the other end of the association and the association nodes. Two mappings describe four possible traverses.
- There may be mapping conditions (which can be either value conditions or cross-conditions) on each of the two mappings for an association
- To **traverse an association** means to start from an object node representing an object at one end of the association, and to find all object nodes representing the objects at the other end, which are linked to the start object by the association. To do this, the XML Reader needs first to go to the association nodes, then on to the nodes representing the objects at the other end of the association.
- To do this, an XML Reader goes through the following steps:
    - From the association mapping involving the start object (which will be called the first association mapping), find the relative XPath between the start object node and the association nodes. (this is usually the default shortest path, but need not be if the ‘Apex’ column is used)
    - Follow this XPath to find a set of association nodes. Use the mapping conditions of the first association mapping to restrict this set.
    - From the association mapping involving the object at the other end of the association (which will be called the second association mapping), find the relative XPath between the association nodes and the object nodes at the other end.
    - From each association node, follow this XPath to find a set of object nodes for the target objects. Use the mapping conditions of the second association mapping to restrict this set.
    - Merge the sets of object nodes found from each association node. This merged set is the result.

In this way, the same pair of association mappings can be used in two ways, to traverse from either end of the association to the other end. Associations are intrinsically navigable in both directions. This general case is shown in the diagram:



This shows an association between class1 (object nodes are green) and class 2 (object nodes are purple). The association nodes are yellow. Starting from a green node of class 1, to navigate the association and find all linked nodes of class 2, you need to traverse the path A to find the relevant association nodes, then path B to find all the correct purple nodes. Path A comes from the first association mapping (labelled as Mapping 1), and path B comes from the second (Mapping 2).

Similarly to start from a purple node of class 2, and get all the linked objects of class 1, you need to traverse path C to find the right association nodes, then traverse path D to find all the correct object nodes of class 1. Again, both association mappings are involved.

In many practical cases, the association node coincides with the object nodes at either or both ends of the association. Then one or both of the two XPaths to traverse is the trivial 'stay here' path. In the diagram above, either paths A and D collapse to a point, or paths B and C collapse to a point.

Usually if any XPath is a long XPath, which could give a large number of nodes, cross-conditions in the association mappings are used to restrict the resulting node sets. (cross-conditions can be evaluated efficiently by pre-indexing of nodes).

The process of traversing an association is first illustrated by an example of the general case – where the association nodes are distinct from the object nodes at either end.

Relational databases frequently represent associations by pairing a 'foreign key' in one table with a 'primary key' in another. In this way they are able to represent all kinds of associations, even those which are many-to-many. When a relational database extract is converted to XML form as in section 4.1, this gives rise to an XML representing the association by the use of a cross-condition between the nodes containing the two keys, which can be stated in the 'Conditions' column of the appropriate mappings. The same approach of matching keys can of course be used in XML languages which are not derived from relational databases, and is fairly widely used. It can be captured in the same way in the Conditions column of a mapping.

We consider a database representing people and cars, in which there is a many-to-many association between cars and people. Each person can drive zero, one, or more cars, and each car can be driven by zero, one, or more people. This database needs to have an extra table, called here DRIVES, to represent the many-to-many association. The tables of the database are illustrated below:

PEOPLE

NAME	LICNO
John Smith	DV101
John Jones	DV404
Albert Faulkes	DV956

CARS

MAKE	REG
Volvo	SSK 231E
Ford	KLM 933H
Saab	UVV240

DRIVES

P_LICENCE	C_REG
DV404	SSK 231E
DV956	UVV240

In this database, there are only two links – that is, only two instances of the association. Mr. Jones drives the Volvo, and Mr. Faulkes drives the Saab. We would expect the database schema to define two constraints of **referential integrity**: that every value in the column P_LICENCE must match some value in the column LICENCE, and that every value in the column C_REG must match some value in the column REG.

The standard form of the XML extract of this whole database is shown below:

```
<database>
	<PEOPLE>
		<record><NAME>John Smith</NAME><LICNO>DV101</LICNO></record>
		<record><NAME>John Jones</NAME><LICNO>DV404</LICNO></record>
		<record><NAME>Albert Faulkes</NAME><LICNO>DV956</LICNO></record>
	</PEOPLE>
	<CARS>
		<record><MAKE>Volvo</NAME><REG>SSK 231E</REG></record>
		<record><MAKE>Ford</NAME><REG>KLM 933H</REG></record>
		<record><MAKE>Saab</NAME><REG>UVV240</REG></record>
	</CARS>
	<DRIVES>
		<record><P_LICENCE>DV404</P_LICENCE><C_REG>SSK 231E</C_REG></record>
		<record><P_LICENCE>DV956</P_LICENCE><C_REG>UVV240</C_REG></record>
	</DRIVES>
<database>
```

In this XML, the <record> tags have been colour-coded green, yellow or purple to match the node colouring of the diagram above.

The complete set of mappings which defines how the XML represents the objects and the association (but not the attributes) is shown in the table below:

Type	Class	Feature	XPath	Condition
object	Person		/database/PEOPLE/record	
object	Car		/database/CARS/record	
assoc	Person	drives.Car	/database/DRIVES/record	(PLICENCE) = (LICNO)
assoc	Car	drivenBy.Person	/database/DRIVES/record	(C_REG)=(REG)

How the association mappings work in this example:

- The two object mappings are simple mappings, using only the core mapping notation.

- Each of the two association mappings is identified by the class at one end (in the 'Class' column), and the role name to get to the other end, followed by the class name at the other end (in the 'Feature' column).
- Both association mappings have the same XPath in the 'XPath' column, because they both refer to the same set of association nodes. This is always the case for association mappings.
- The cross-condition in the first association mapping, (PLICENCE) = (LICNO), has a relative XPath from the association node on its left-hand side, and a relative XPath from the object node on its right-hand side. Cross-conditions in association mappings are always written in this way.
- In this example, all relative XPaths in the cross-conditions are simple descending XPaths, starting from a <record> node and ending on one of its child nodes. In all cases, the XPath can only deliver at most one node, to give a unique value to test.

We can follow through the process of traversing the association Person.drives.Car from one of the 'Person' object nodes, say the node representing John Jones. This is one of the nodes with XPath 'database/PEOPLE/record' from the root node. The process involves two phases:

A. Go from the starting object node to find each qualifying association node (using the first association mapping). This is path A in the diagram above.

B. Go from each qualifying association node to find all qualifying object nodes of the target class (using the second association mapping). This is path B in the diagram above.

The steps in phase (A) are:

1. Anticipating evaluating the cross-condition in the first association mapping (the association mapping with Class = Person), the XML Reader follows the XPath 'LICNO' on the right-hand side of this cross-condition from the starting <record> node, and saves the value 'DV404' of the one node it finds. This is the key value that identifies the person John Jones.
2. The Reader finds the shortest (default) XPath from the <record> object nodes of class 'Person' to the association nodes. This path goes right up to the <database> root node, and down again via the <DRIVES> node.
3. The Reader follows this XPath from the object node for 'John Jones' to retrieve a node set, which contains all the <record> nodes beneath the <DRIVES> node.
4. The Reader then filters this node set, using the cross-condition of the first association mapping. For each <record> node under the <DRIVES> node, it follows the XPath 'PLICENCE' to find a single child node. It compares the value of this node with the value 'DV404' which was saved at step (1).
5. Only one of the association nodes passes this test, and goes through to be used in Phase B.

The steps in phase (B) are:

6. Anticipating evaluating the cross-condition in the second association mapping (the association mapping with Class = CAR), the XML Reader follows the XPath 'C_REG' on the left-hand side of this cross-condition from the one starting <record> association node, and saves the value 'SSK 231E' of the one node it finds.
7. The XML Reader finds the shortest (default) XPath from the <record> association nodes to the object nodes for class 'Car'. This path goes right up to the <database> root node, and down again via the <CARS> node.
8. The XML Reader follows this XPath, from the one association node which resulted from step (5), to retrieve a node set which contains all the <record> nodes beneath the <CARS> node.
9. The Reader then filters this node set, using the cross-condition of the second association mapping. For each <record> node under the <CARS> node, it follows the XPath 'REG' to find the single child node. It compares the value of this node with the value 'SSK 231E' which was saved at step (6).

10. Only one of the object nodes passes this test. This one object node is the result of following the association from the object node for the person 'John Jones'. It represents the one car that he drives.

You will notice the close similarity between the steps 1-5 of Phase A, and steps 6-10 of phase B. The association could have been navigated in the reverse direction (from a car to the people who drive it) by an identical series of steps in the reverse direction, *mutatis mutandis* .

For both phases (A) and (B), the steps describe what an XML Reader could do in principle to get the right result. That is to follow a long XPath, retrieving a node set which may be large, and then to filter that node set using the cross conditions. In practice the same effect can be achieved more efficiently, by pre-indexing the nodes. This is just like using relational database indexes for efficient retrieval of associated records.

This has been an example of the most complex type of association mapping, where the association nodes are distinct from the object nodes of both end classes. For most association mappings, this is not the case. Usually the association nodes coincide with one or both of the object nodes. Then one of the two phases (A) or (B) becomes unnecessary, or trivial.

In the case of a 1:N association, the association node coincides with the object node at the 'N' end of the association. The number of links from one object at the '1' end is equal to the number of objects at the 'N' end. There is no need to navigate from the association nodes to the object nodes at the 'N' end, because they are the same node. An example of this will be given in the next sub-section.

In the case of a 1:1 association, the association node coincides with the object nodes of both end classes, which then coincide with each other. There are then identical numbers of objects in the two classes (as there must be, if they are in a 1:1 relation to one another) because they are represented by the same nodes. In this case, both of the navigation phases (A) and (B) are trivial.

4.5.2 **Longer Relative XPaths**

Just as for attribute mappings, it can happen that the relative XPaths between an object node and an association node needs to be longer than the shortest relative path, which the mappings assume as the default. In these cases the longer XPath can be defined simply by defining an apex node, in the 'Apex' column, which is nearer to the root of the document than the apex node of the shortest path.

This frequently occurs when there is an association between objects of some class and objects of the same class, and so the shortest possible path turns out to be too short, the simple 'stay at this node' path. Consider the following flat XML language, which represents people, each of whom has a staff number, and a manager identified by his or her staff number:

```
<staffMembers>
       <member name="Jones" staffId = "23" managerID="" />
       <member name="Smith" staffId = "40" managerID="23" />
       <member name="Walsh" staffId = "50" managerID="23" />
       <member name="Button" staffId = "65" managerID="50" />
</staffMembers>
```

Note that a manager can have a manager. This XML can represent the manager-subordinate relation to any depth, even though it is a flat XML language. How it does so is captured in the following mappings:

Type	Class	Feature	XPath	Condition	Apex
object	Person		/staffMembers/member		
assoc	Person	bossOf.Person	/staffMembers/member	(@managerId) = (@staffId)	/ staffMembers
assoc	Person	hasBoss.Person	/staffMembers/member		

The association has multiplicity 1:N; one person can be manager of many people, but can have only one manager. Therefore the association node is the same as the 'Person' object node at the subordinate end. This is captured in the second association mapping. Because this association mapping has the same XPath as the object mapping, and has no 'Apex' value, the default shortest relative XPath between these nodes is the 'stay here' path; the object node and the association node are the same.

If the first association mapping had no 'Apex' value, the same would apply to it; because it has the same XPath as the 'Person' object mapping, the default shortest path would apply, and this is the trivial 'stay on this node' path. The result would be that every person is his own manager – not the result we want.

However, because this mapping has an apex node 'staffMembers', the relative XPath from the <member> association node to the object node (or in the reverse direction) must go up to the top 'staffMembers' node; so from any association node, the XPath can reach any other <member> node, representing any person. Without the mapping condition to narrow down this large set of object nodes, it would imply that any person has all the other people (including himself) as manager – again, not the result we want.

The mapping condition compares the 'managerId' attribute on the association node with the 'staffId' attribute on the object node representing a person. Because the 'staffId' attribute is assumed to be a unique identifier of people, this means that any person can have at most one manager.

This describes how the association can be followed in one direction from a person to find the one person who is his manager. The same process can be used in reverse, to follow the same association in the reverse direction – finding all people who have one person as manager. It can be seen that in both cases, what happens is a special case of the Phase A/Phase B process described in the previous sub-section – but where one of the phases A or B is trivial.

Again, the description tells logically how an XML Reader could follow the association, by creating large node sets and then filtering them with a mapping condition. As in the previous examples, in practice one would expect to use a more efficient process, using the mapping conditions to pre-index nodes.

4.5.3 **Subsets in Association Mappings**

The same XML language may represent objects of the same class in several different places, for several different purposes – which results in several object mappings to the same class. When this happens, the different object mappings are distinguished by a 'subset', which is a unique string written in brackets after the class name in the 'Class' column. A typical example is 'Person(s1)', 'Person(s2)' and so on.

In this case, every association mapping for an association of the class must identify exactly which of the different object mappings it relates to, for both ends of the association. Therefore in every association mapping:

- The class name in the 'Class' column must have a subset identifier, if necessary.
- The class name following the association role name in the 'Feature' column must have a subset identifier, if necessary.

As usual, the default subset identifier, if none is given, is the empty string ''.

4.5.4 **Mapping Lists**

The mapping notation so far has said nothing about the order of items in lists. Because we require attributes in UML class models to be single-valued, the issue of ordering does not arise for attributes, but only arises for associations. If some association is ordered, we might typically require the order of child nodes of some XML node to follow the same order as the order of target objects of the association.

To express this constraint and others like it, we add two artificial attributes 'el_position' and 'child_position' to the XML definition of any Element, and an attribute 'order' to the UML definition of a link. We can then add mapping conditions relating these, in the Condition column of an association

mapping, to define how ordering information is propagated between the XML instance and the class model instance.

Consider the following example of a person's name:

```
<name>
       <name_prefix>Ms</name_prefix>
       <name_given>Sally</name_given>
       <name_given>Ann</name_given>
       <name_family>DuBois</name_family>
</name>
```

The definition of the XML says that amongst the child nodes of <name> , the single <name_prefix> element comes first, followed by any number of < name_given> elements (whose order is significant), and then a <name_family> element.

In the UML class model, there are two classes, 'Name' and 'NamePart'. There are three associations from Name to NamePart, with role names 'prefix', 'given', and 'surName'. The 'given' association is one-to-many, and is ordered. So there can be several instances of 'NamePart' related to a single instance of 'Name' by the 'given' association, and they are ordered.

The mappings which capture this are shown below:

Type	**Class**	**Feature**	**XPath**	**Condition**
object	Name		/name	
object	NamePart(pref)		/name/name_prefix	
object	NamePart(give)		/name/name_given	
object	NamePart(sur)		/name/name_family	
attrib	NamePart(pref)	value	/name/name_prefix	
attrib	NamePart(give)	value	/name/name_given	
attrib	NamePart(sur)	value	/name/name_family	
assoc	Name	prefix.NamePart(pref)	/name/name_prefix	(@child_position) = (1,order)
assoc	NamePart(pref)	partOf.Name	/name/name_prefix	
assoc	Name	given.NamePart(give)	/name/name_given	(@child_position) = (2,order)
assoc	NamePart(give)	partOf.Name	/name/name_given	
assoc	Name	family.NamePart(sur)	/name/name_family	(@child_position) = (3,order)
assoc	NamePart(sur)	partOf.Name	/name/name_family	

Note how the class 'NamePart' has been mapped in three different subsets (called 'pref', 'give', and 'sur') to distinguish the different types of name part (because they are not separate classes in the class model).

The object mappings and attribute mappings are then fairly straightforward. We need to understand how an XML Reader and an XML Writer will use the special mapping conditions in the association mappings.

The **XML Reader** adds an extran attribute 'child_position' to the elements <name_prefix>, <name_given>, and <name_family> which are child nodes of <name>, in the XML instance which is to be read. It gives values to this attribute according to the position in the list of child nodes (1, 2, 3, 4 in the example above). Then, in navigating the associations 'prefix' and 'family', which have maximum multiplicity 1, it does nothing with the values of these attributes. However, in navigating the ordered 1:N association 'given', it uses the attribute values to determine the order of the links in the class model instance. It reads the values of the attribute (which are 2 and 3), and orders the links in ascending order of these values.

In going from a class model instance to an XML instance, the **XML Writer** does a similar process. It writes an XML with an extran attribute 'child_position' on the elements <name_prefix>, <name_given>, and <name_family> which are child nodes of <name>. This extran attribute has values which are not simple numbers, but are number pairs as in the right-hand sides of the mapping conditions in the table.

For the four child nodes, they are (1,1), (2,1), (2,2) and (3,1). It then orders the child nodes in ascending order of these number pairs, and finally removes the attributes. This achieves the right order for the different node names, and the right order for nodes of one node name.

There are probably more complex ordering constraints between XML instances and class model instances, which cannot be captured in this mapping notation. Those may require the use of procedural escapes from the mapping notation, as described in section 4.7.

4.5.5 Alternate Association Mappings

Just as for attributes, it sometimes happens that the same association is represented more than once in the XML. This can be an unfortunate design feature of the XML language, because it means that different versions of the same association may be inconsistent with each other. Nevertheless it occasionally happens. When it does so, it can happen in one of two ways. For an association which is represented just twice, these are:

1. The XML Writer is forced to write out both representations of the link; then the XML Reader can choose either version to follow the link.
2. The XML Writer may choose which representation of the link to write out, but need not write both. Then the XML Reader is forced to look at both representations, to see if either of them can be followed.

These two possibilities are defined by the column 'Alternates', whose values can be 'all' (for case (1), where all the representations must be present) or 'some' (for case (2), where only some of the representations need be present). Usually, the 'Alternates' column is not needed, and is empty.

4.6 Mapping Modules

Large sets of mappings, for complex XML languages or complex class models, may have hundreds or thousands of mappings. Therefore the tabular mapping notation can result in tables with hundreds or thousands of rows. Even though the tables can be sorted and searched with spreadsheet tools, very large tables are inconvenient, and there need to be mechanisms for dividing tables of mappings into smaller pieces, or modules. There are two distinct uses for the modules:

- To divide mappings into convenient-sized tables for reading and review
- To make common groups of mappings available for reuse at several different places in a set of mappings.

For instance, some common XML structure may be used repeatedly in a document to represent the parts of an address – repeatedly representing the same parts of the class model. In these cases, one does not want to have to re-define the same patterns of mappings over and over again.

The requirement for reuse makes these modules of mappings very much like methods, functions or subroutines in a programming language, and there are several parallels in how it is done. There can be strong 'information hiding' between the module and the mappings that 'call' it.

It is useful to define two distinct types of mapping module, with different levels of reusability. These are called **macro modules** and **callable modules**. Both refer to separate tables of mappings which can be 'imported' for use in other tables of mappings. In the analogy with programming languages, a macro module is like a macro, which can be substituted within the body of some other set of mappings; whereas a callable module is like a callable method, which is passed an argument and returns a result.

4.6.1 **Macro Modules**

A macro module is a table of mappings. It is invoked from a row in some other table of mappings which calls it. The effects of a macro module can be entirely described in terms of in-line expansion of tables of mappings, adding the rows of the macro module table to the rows of a mapping table that calls it.

There are rows in the calling table which have 'macro' in the 'Type' column, and an identifier of the called mapping table in a new column 'Module', and an XPath in the 'XPath' column. No other columns are used in a calling row. The identifier can be a file name or file path relative to the file containing the calling mapping table. For each calling row, the effect of calling the macro module is as follows:

- The calling row, with Type = 'macro' is removed.
- One row is added to the calling table for every row of the called table.
- For each added row, the XPath in the calling row is pre-pended to the XPath in the called row. This means that for every XPath in the called table, the Element name of the first step is replaced by the inner element name of the XPath in the calling row. For instance, pre-pending '/A/B/C' to '/X/Y/Z' gives '/A/B/C/Y/Z'; so the name 'X' does not appear in the expanded result.
- For every object mapping in the called mapping table, if there is a mapping in the calling mapping table (with all other macros expanded) to the same class, then any clash in the subset names between the two mappings is removed, by altering the subset name in the mappings of the called mapping table. This may involve altering subset names in object, attribute and association mappings together, to keep attribute and association mappings in step with object mappings.

The expanded mapping table can then be used by an XML Reader or XML Writer, just as if there had been no macro expansion. The extended mapping table after macro expansion must obey all the validity constraints of any mapping table. For instance, the joined XPaths must all describe nodes which can exist in the XML document structure.

To avoid the confusion of defining mappings to the same node in two different mapping tables, there are the following constraints:

- No rows in the calling mapping table must have an XPath which is an extension of an XPath in a calling row. All such extended XPaths can arise only from the called macro table.
- Only one macro mapping table can be called from an XPath.

Macro mapping modules can be used simply to divide a large mapping table into pieces, where the pieces are defined by sub-trees of the whole XML tree. This gives a convenient shortening of XPaths within a macro mapping table.

One macro mapping module can be called from several different places in a mapping table, and one macro mapping module may call another. However, a macro mapping module may not call itself recursively, either directly or indirectly, as this would give an infinite number of rows after expansion.

4.6.2 **Callable Mapping Modules**

A callable mapping module is more restrictive than a macro module, and has a much greater degree of information hiding. Every callable mapping module has a **parameter class** in the class model. The effects of the call can be described in terms of how an XML Reader uses the callable module, and how an XML Writer uses it. In both cases we describe what the XML Reader or Writer passes to the callable module, what it does, what result it returns, and how the result is used in the 'calling' mappings:

- The **Un-Selective** operation of an **XML Reader** is as follows: the callable module takes as argument an XML node (with its sub-tree); and it returns an 'object sub-tree', which is represented by the XML sub-tree through the mappings in the module. This is a fragment of an instance of the class model, rooted at an entry object of the parameter class and containing its attributes, other objects linked to it directly or indirectly, and their attributes. The module reads

the XML sub-tree to create the object sub-tree. This object sub-tree is stitched into the object sub-tree being made for the calling mappings.

- The **Selective** operation of an **XML Reader** is as follows: the callable module takes as arguments (a) an XML node (with its sub-tree); (b) an object token for an object of the parameter class (or a subclass of it), and (c) the name of some association to be navigated or attribute to be retrieved; and it returns either an attribute value for that object, or object tokens for objects got by navigating some association from that object. (It may on later calls return attribute values for those objects, navigate further associations, and so on)
- The operation of an **XML Writer** is as follows: the callable module takes as its argument an XML Document (to define namespaces) and an object sub-tree, rooted at an object of some class. It returns the XML sub-tree representing the object sub-tree, created through the mappings in the module. This XML sub-tree is stitched into the XML tree being made for the calling mappings.

The use of a callable mapping module is restricted to cases where:

a) All mappings within the module are mappings to nodes within a sub-tree of the XML

b) The fragment of the class model being represented is has a single 'entry object', and any other objects represented within the module are linked only to that object, directly or indirectly; there are no links to objects outside this subset.

c) No attribute or link is represented within the XML sub-tree unless the corresponding objects are also represented in the sub-tree.

d) Mappings within the module do not depend in any way on nodes outside the XML sub-tree (e.g. for mapping conditions)

So the use of callable modules tends to be restricted to the use of 'tree-like' class models, such as an HL7 RMIM – even though the restriction (b) does not require the class model to be tree-like within the part represented in the module. They typically cannot be used for mappings to relational databases, where relationships across tables are represented by prime key/foreign key pairs (i.e. cross-conditions in mappings) which span the XML sub-trees for two tables.

A callable mapping module has two formal parameters passed to it: the XML node or sub-tree, and an object token for an object of its parameter class. Parameters which match these formal parameters are defined in the mapping row which defines the call. This row has the following cell values

- Type = 'call'
- Class = the class of the object token to be passed to the module. This may be a subclass of the class which is a formal parameter of the module.
- XPath = the XPath of the node being passed to the module
- Module = the identifier of the module – a file name or file path from the calling module.

No other columns except the 'Comments' column are used.

The mappings in a mapping module include an object mapping to the parameter class on its root node, and the mappings are validated as if there were such a mapping, although that mapping is not actually used to read XML; some mapping from the calling set of mappings is used to pass an object token to the module. For an object token to be passed to the module, there must be an object mapping of that class in the calling mapping set, on the calling node or on some ancestor of that node – so that when the module is called, there is a unique object token of that class 'in hand' to pass to the module as argument. The actual object token may come from a mapping on an ancestor node, rather than the root node of the mapping module.

When the module is called, the effect is similar to an inline expansion of a macro mapping module – except that the object mapping to the parameter class on the root node is not included in the expansion, because there is already an object mapping in the calling module. The expansion process includes the

concatenation of the XPaths – which takes the name of the joining node from the calling mapping set, not the called mapping set.

Some of the same restrictions as for macro mapping modules apply to callable mapping modules:

- The effect of in-line expansion of a callable module must be to create a valid set of mappings – for instance, all concatenated XPaths must be valid and be able to reach some node in the XML structure.
- The calling mapping set should have no mappings whose XPath includes the XPath at which the call in made and is longer than it
- Only one callable mapping table or macro mapping table can be called from any XPath.

One callable mapping module can be called from several places in one calling set of mappings, and callable modules can call each other. Unlike macro mapping modules, callable mapping modules can call themselves and each other recursively. This is a way to capture the mappings to a recursive XML structure, which represents an association between a class and itself, to indefinite depth.

In cases without recursion, an XML Reader or Writer should be able to handle a callable module by inline expansion of its mapping rows – just as if it were a macro mapping module.

Because callable mapping modules relate to reusable sub-trees of an XML document, it is often convenient to associate a mapping module with an **XML Schema Complex Type**, which defines the structure of a reusable sub-tree.

In both callable and macro mapping modules, XPaths in the 'XPath' column are written as absolute XPaths, with an initial '/'. This is so even though they are not actually absolute XPaths in any document; they are always concatenated after some XPath from the calling mappings.

4.7 Procedural Escape Mechanisms

The mapping language has three procedural escape mechanisms, for cases where mapping with the declarative features of the language is awkward or impossible:

1. Attribute value conversions in procedural code
2. Callable Procedural Mapping Modules
3. Wrapper classes.

Mechanism (1) has been described in the sub-section on extended attribute mappings. Mechanisms (2) and (3) are described below.

4.7.1 Callable Procedural Mapping Modules

A callable procedural mapping module has the same external behaviour as any callable mapping module, as described in section 4.6 above; but it is implemented in code, rather than in mappings. The un-selective XML Reader behaviour (building a whole object sub-tree) can be built up using calls of the selective XML Reader behaviour. So there need to be defined interfaces for the selective XML Reader behaviour, and for the XML Writer behaviour. Either or both of these may be provided, depending on requirements. These interfaces are described below in abstract, language-independent terms.

The selective XML Reader behaviour involves three interfaces: get all objects, get an attribute of an object, and navigate an association from object. So there are four interfaces which may be provided, defined below:

Interface	Arguments	Returned result
Reader: get all objects	• XML Node	Set of object tokens, for all objects in the class or sub-classes represented in the XML subtree
Reader: get attribute value for object	• XML node • Object token • Attribute name	Value of attribute for object
Reader: follow association	• XML node • Object token for start object • Association role name • Target class name	Set of object tokens, for all objects in the target class or sub-classes represented in the XML subtree and reached from the start object by the association
Writer: create XML sub-tree	• Entry object to object sub-tree • XML Document (to define namespaces)	XML sub-tree in the Document

Coded implementations of mappings can be provided for XML sub-trees, and object-sub-trees, which would be awkward or difficult to express by declarative mappings; and these coded implementations can be called from within any mapping set. Calls in the reverse direction, from a coded implementation into a mapping set, can be made using the same interfaces.

The table defines interfaces at an abstract level. Concrete interfaces depend both on language and implementation framework. For instance, within Java the precise interfaces of 'object token' and 'XML Node' can be as defined in the open source mapping tools on the HL7 GForge. Within the Eclipse Modelling Framework (EMF), object sub-trees might be represented in Eclipse Ecore. The precise representation of a set needs to be defined, and so on.

4.7.2 **Wrapper Transforms**

Some XML languages represent class model information in ways that are awkward to express in the mapping language – or would require special extensions to the mapping language, creating a danger of a host of messy *ad hoc* extensions.

For instance, some XML language may represent a link between two objects by describing the position of one object in the XML document, using some long string value of an attribute; pulling this string apart may be inconvenient or impossible using mapping language constructs in a cross-condition. But it may be easy to do in a procedural language such as Java.

Similarly, some XML node may contain a long text field, which can be parsed to represent several objects of some class. The definition of the mapping language contains no such parsing functionality.

To avoid having to define a series of messy extensions to the mapping language to address such cases, it is possible to apply a **wrapper transform** to the XML before reading it through the mappings. This wrapper transform should do as little as possible, making local changes to the XML (e.g. within certain attributes) to make it easier to read through mappings.

The wrapper transform is applied to the whole XML document before reading it through mappings. It can be defined as a wrapper function or class or template as appropriate in any language, such as Java or XSLT. The definition of the wrapper transform, if there is one, is part of the information pertaining to the whole mapping set, described in the next sub-section.

If the mappings are to be used to write XML as well as to read it, the wrapper transform should be made reversible, and implementations should be provided in both directions. These two implementations should round-trip consistently.

The wrapper transform applied before reading an XML document through mappings is called an in-transform; the wrapper transform applied after writing an XML document through mappings is called an out-transform. They should form a self-consistent round-trip.

If the wrapper transform alters the schema of the XML, mapping tools will probably require an XML schema for the modified language.

To make mappings to non-XML structures (such as HL7 V2) a wrapper transform can be used to convert the non-XML form to an easily mappable XML form.

4.8 Information Pertaining to a Whole Mapping Set or Module

There are a few small items of information which pertain to a whole mapping set or module. These are:

- The XML schema defining the structure being mapped
- The class model being mapped to (in some persistent form such as a UML2 model or EMF Ecore model)
- Whether this is a top-level mapping set, or macro mapping module, or callable mapping module
- If it is a callable mapping module, its parameter class
- It is convenient to define all the namespaces of the XML (just in case there is no schema to do so)

For some of these items, it is a constraint that they have the same value for a calling mapping set and the called (callable or macro) mapping set. For instance, they must be mappings of the same XML structure (e.g. the same schema) to the same class model.

4.9 Persistence and Exchange Formats

There are several possible data formats for persistence, exchange and sharing of mappings, and there should be simple tools to convert between these formats. Possibly not all will be required, and it will take experience to show which are most used. The following are candidates:

4.9.1 Tabular XML

This is a simple XML language with one XML document per mapping table. Each document has a header followed by an XML definition of a single table, as in this example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<MappingSet
        classModel="lra_1.ecore"
        structure=" ADT_A03.xsd"
        callable="true"
        module="true"
        entryClass="CR_RolePerson">
    <Mappings>
        <row>
                <Type>object</Type>
                <Class>CR_RolePerson</Class>
                <Feature/>
                <XPath>/PID</XPath>
                <Comments/>
                <Condition/>
                <Key/>
                <Filter/>
                <Apex/>
                <Convert_In/>
                <Convert_Out/>
                <Module/>
                <Alternates/>
```

```
</row>
<row>.....
```

The XML Schema for this language, with some example mapping tables in that form, are in a folder in the Open Mapping Language Review Pack.

Some drawbacks of this persistent form are:

- It is not straightforward to edit it.
- When viewing the table, it is not easy to sort the rows into some order they were not supplied in.
- Similarly, it is not very easy to hide or rearrange columns.

Also under consideration is a change to this XML tabular form. For some columns, such as the 'Condition' column, the text in one cell may describe several conditions (separated by ';'), each with its own structure. Rather than cram these all into text in one XML element, it would probably be better to have nested XML elements which represent the different conditions explicitly; similarly for other columns with composite content.

4.9.2 **Comma-Separated Value Files**

Text files with comma-separated values (csv files) can be easily read into a spreadsheet, which is convenient for browsing or editing the mappings. Rows can be sorted on any column; columns can be re-ordered or hidden; and any cell can be easily edited.

Mappings can be stored in csv files with one column for every column of the mapping language, with names 'Type' and so on. There is one column of the csv file for every mapping column, even though it might not be used in any mapping. The header information of a set of mapping held in a csv file (described in the previous sub-section 4.8) is actually held in a small table at the start of the file.

Examples of mappings in csv form, saved as Excel worksheets, are in a folder in the Open Mapping Language Review Pack.

4.9.3 **Relational Databases**

Because mappings are tabular, they can in principle be stored in a relational database. If it turns out that there are important uses for this, this format will be defined in more detail.

4.9.4 **Annotations in an XML Schema**

In a table of mappings, one XPath can have several different mappings, each defined by a row in the table. However, an XPath from the root of the document (as is used in the 'XPath' column of a mapping table) can also be defined as the path to an element or attribute.

Therefore it would be possible to store sets of mapping rows as annotations to the 'element' or 'attribute' elements in a schema that they apply to. Every column of the mappings, except the XPath column, would be stored this way.

It is not currently known what would be the applications of this way of storing mappings, and there is the added question of what happens when an XML Schema complex type extends or constrains some other complex type. Both may have mapping annotations, so how are they to be combined? For these reasons, the storage of mappings as annotations to an XML Schema has not yet been explored further.

5. MODEL-TO-MODEL MAPPING

The primary use of the mapping notation is for structure-to-model mapping, where the data structure is an XML structure (or something that can be converted into it) and the model is a UML class model.

The same mapping notation can be used unaltered for Model-to-Model mapping. When used in this way, the mappings can be used to translate an instance of one UML class model to an instance of another, in either direction.

Using the notation for model-to-model mapping is a straightforward application of its use for structure-to-model mapping. This is so, because nearly all formalisms and tools designed to handle UML class models and their instances have defined means to make persistent XML forms of those instances. For example:

- An HL7 RMIM is a UML class model. The HL7 XML ITS defines an XML persistent form for an instance of any RMIM.
- The XMI standard defines an XML persistent form for instances of a UML model or metamodel, which is used by many UML tools to import and export instances
- The Eclipse Modelling Facility (EMF) defines a persistent XML form for instances of Ecore class models, which is a variant of XMI.

Usually the principles of these persistent XML forms are fairly straightforward and regular.

To map one UML model onto another, you use the XML persistent form of one of the models as the definition of an XML structure; then you map that structure onto the other model, using the structure-to-model mapping constructs.

If Model A has been used as a definition of a structure, Structure A, which is then mapped onto model B, then you can convert an instance of model A to an instance of model B by making the persistent form of instance A (i.e. an instance of structure A), then using an XML Reader and the mappings to convert it to an instance of model B. To go in the reverse direction, you use the same mappings with an XML Writer – going from an instance of model B to an instance of Structure A and then converting it to an instance of model A.

In this process, there is some asymmetry between the two models. You can make model-to-model mappings in two different ways – either by treating model A as a structure, or by treating model B as a structure; and there will be no very simple relation between the two sets of mappings. However, they should both give the same results when translating from one model to another.

The persistent forms of some UML representations (such as EMF Ecore) have some features which can be awkward to map. For instance, the representation of non-containment associations in the EMF Ecore persistent form is awkward to map, because it uses a particular string notation for the location of a node in the XML, and this notation is hard to use in mapping conditions. To deal with this problem, it is best to use a simple wrapper transform to get rid of any awkward features – effectively re-defining the persistent form to make it easier to map. The mapping tools available from the HL7 GForge have such a wrapper transform included.

The persistent forms of UML models depend on which associations in the model have been declared to be composition (or in EMF terminology, containment) relations. Composition relations are usually represented simply as nested XML elements. See the Appendix for a discussion of composition relations.

6. SUMMARY OF THE MAPPING LANGUAGE

This section gives a summary of all the table columns used in the mapping notation, briefly describing their usage. This is done first for the columns of the core mapping language, then for the extended language.

6.1 Core Mapping Columns

Column	Usage	Sections
Type	This column denotes the type of mapping. There are three main values, for the three main types of mapping: • object : denotes an object mapping, which states that the node represents an object in some class • attrib: denotes an attribute mapping, which states that the node represents the value of some attribute • assoc: denotes an association mapping, which states that the node represents a link (instance of an association). Association mappings always occur in pairs for the two role names and target classes Two other values are used for mapping modules: 'macro' to use a macro mapping module, and 'call' to denote the use of a callable mapping module	3.5, 3.6, 3.7, 4.6
Class	Contains a unique designation of the class being mapped. This may be the class name, and may be preceded by the package name if necessary to disambiguate it. Or it may be the name of an entry class into the mapping set, followed by a trail of association names to a final class name. If there is more than one object mapping to the same class, the different object mappings are distinguished by a subset indicator. This is a string in brackets after the class name. It must be unique within the class and the mapping set or module. Subset indicators must be used even when association trails are used. Any attribute or association mapping must be linked to some object mapping, by having the same subset indicator	3.5, 3.6, 3.7
Feature	Empty for object mappings. For attribute mappings, it is the name of the attribute, which may be inherited. For association mappings, it is the role name used to get from an object of the class in the Class column, to an object of some other mapped class. The second class is defined by a unique class name (and subset, if necessary) after the role name and '.'. The association may be inherited.	3.6, 3.7

XPath	A descending absolute XPath from the root of the document, or from the root node of this mapping module. Usually only contains 'child::' and 'attribute::' steps, in abbreviated form, but may also contain 'descendant-or-child::' (//) steps. Should generally not contain conditions on the nodes; these are held in the 'Conditions' column.	3.5, 3.6, 3.7
Comments	Any text comment or description of the mapping. May be used to note problems, where mappings cannot be made.	

6.2 Extended Mapping Columns

The column 'Mapping Type' in the table below describes which of the three types of mappings – object, attribute , or association – the named mapping column is used for. Association mappings denoted by the Type value 'assoc'

Column	**Mapping Type**	**Usage**	**Sections**
Condition	all	For all types of mappings, this column may contain a number of value conditions, separated by ';'. The usual form of a value condition is (XPath) = 'constant'; but comparators other than '=' may be used. 'XPath' is a relative XPath from the mapped node, and 'constant' is a string constant. For attribute and association mappings only, it may also contain cross-conditions. The usual form of a cross-condition is (XPath1) = (XPath2); but comparators other than '=' may be used. 'XPath1' is a relative XPath from the mapped attribute or association node, and 'XPath2' is an XPath from an object node, denoting the object owning the feature (attribute or association). The relative XPaths are allowed to contain predicates, where necessary to ensure that their resulting node sets can only contain one node, to give a unique value to test.	4.3.1, 4.4.1, 4.5.1, 4.5.5
Key	object	Denotes that there may be several nodes with the same XPath, all of which represent the same object (instance). The value of the column is a series of attribute names, or chains of associations followed by an attribute name, for attributes of the mapped class or of linked classes which define a unique key for the object. All these attributes and associations must be mapped. If two different nodes both represent the same instance, it can then be detected in that the values of all attributes in the key are equal for the two nodes – assessed by using the attribute and association mappings to get the attribute values.	4.3.4
Filter	object	Denotes that the only objects of the class represented in the XML are those that obey the filter conditions. Filter conditions are expressed in terms of attributes and associations in the class model. Equality filters on attributes of the class carry an implied attribute mapping – that the attribute has that value for all objects represented by the object mapping that has the filter.	4.3.5

Apex	attribute, assoc	Denotes that the relative XPath, used to get from the object node to the association or attribute node, is not the default shortest XPath between those nodes, but has some higher apex node, as defined in the column.	4.4.2, 4.5.2
Convert_In	attribute	Defines the function or table used to convert ‘in’ from the value of the XML node (possibly together with values of other attributes of the same object represented in the XML), to get the value of the attribute in the UML class model.	4.4.3, 4.4.4
Convert_out	attribute	Defines the function or table used to convert ‘out’ from the value of the attribute in the UML model instance (possibly together with values of other attributes of the same object), to get the value of the XML node.	4.4.3, 4.4.4
Module	call, macro	Defines which mapping module is to be called or macro-expanded at this row of the mapping table	4.6
Alternates	attribute, assoc	Used when there are several alternate representations of the same attribute or the same link in the XML. Denotes whether they are all expected to be present in the XML, or whether only some of them may be present.	4.4.5, 4.5.3

7. VALIDATION CRITERIA FOR MAPPINGS

The validation constraints applicable to the contents of mapping columns are described here.

Column	Validation Constraints
Type	• Must be one of 'object', 'attrib', 'assoc', 'macro' or 'call' • 'assoc' mappings must come in pairs for the two ends of the association, both having the same XPath
Class	• Must be a class in the class model, possibly followed by a subset indicator in brackets. If no brackets, the subset is ''. • Each subset can appear in only one object mapping for a class • For attribute and association mappings, there must be an object mapping with the same class and subset
Feature	• Empty for object mappings, or rows with Type = 'call' or 'macro' • For attribute mappings, the feature must be the name of an attribute in the class model (may be inherited) • For attribute mappings, the named attribute must be single-valued and have a simple type • For association mappings, the feature must be the role name of an association from the class (may be inherited), followed by '.' and then the target class name and subset. • If there is no role name (that end of the association is not navigable) the role name must be the role name of the other end of the association, in brackets and preceded by '-'. This ensures the two ends can be unambiguously matched together. • For association end mappings, the target class must be mapped
XPath	• Must be a valid absolute descending XPath from the root of the document or the mapping module subtree, which can sometimes lead to nodes • For attribute mappings, the relative path from the object node to the attribute node must lead to at most one node, unless there are cross-conditions • For association mappings, the relative path from the association node to the object node must lead to at most one node, if the maximum cardinality of that end of the association is 1 and there are no cross-conditions. • Minimum cardinalities of attributes and associations should match the minimum number of nodes given by the relative XPaths • Maximum cardinalities of associations should match the maximum number of nodes given by the relative XPaths, after any conditions are applied
Comments	• Any text
Conditions	• Value conditions are (XPath) {relation} 'value' • {relation} can be =, <, >, =<, >=, contains, containsAsWord • XPath is a relative XPath from the mapped node; following XPath from the mapped node must lead to at most one node • Cross-conditions are allowed only in attribute and association mappings • Cross-conditions are (XPath1) {relation} (XPath2)

	• Following XPath1 from the mapped attribute or association node must lead to at most one node • Following XPath2 from the object node must lead to at most one node
Key	• Must be a list of attribute names, or role names of associations followed by attribute names • Attributes are attributes of the mapped class, maybe inherited • The associations must lead to a unique object • Attributes and the objects they lead to must be mapped
Filter	• Form is attribute='value', or several of these separated by ';', or 'role name FROM class' • 'attribute' is an attribute of the mapped class
Apex	• Must be a descending XPath from the root of the document or root node of the mapping module • Defined node must be an ancestor of the apex node of the shortest relative XPath between the object node and the attribute or association node
Convert_in	• Must uniquely define the conversion methods or tables • If conversion methods have more than one argument, arguments must all be mapped attributes of the class
Convert_out	• Must uniquely define the conversion methods or tables • If conversion methods have more than one argument, they must all be attributes of the class
Alternates	• Must be 'all' or 'some'

8. APPLICATIONS AND TOOLS

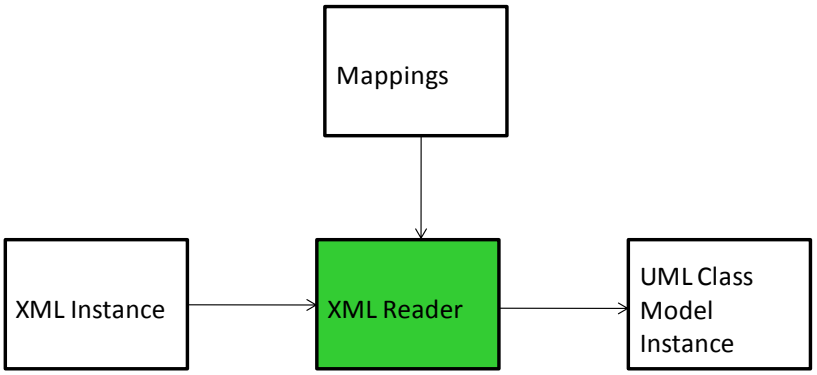
Having a precisely defined mapping language enables different people to exchange, review and use mappings in the language. It also enables machine support of the language; so that when using mappings in these ways, there can be common tool support.

This section gives brief descriptions of some of the applications and tools which are made possible by the existence of the language. For many of these tools, the development of a tool would be straightforward in principle, because the definition of what the tool does follows directly from this definition of the mapping language.

For some of these applications, the mapping tools available at http://gforge.hl7.org/gf/project/v2v3-mapping/frs/ provide an open-source implementation of the functionality; so those tools demonstrate that the functionality can be implemented, based on the mapping language in this paper.

8.1 Reading Data Instances

This is the functionality of an **XML Reader**, which has been used through this document to define what the mappings mean. Its function is to take an instance of the mapped data structure (e.g. XML or relational database extract) and to create from it an instance of the class model, or provide information extracted from that instance. For convenience, the diagram from section 3 is repeated here:



There are two types of XML Reader:

- An **Un-Selective XML Reader**, which takes an instance of the data structure and converts it to an entire instance of the class model, in some language-dependent representation of class model instances such as EMF Ecore, which is built on Java.
- A **Selective XML Reader**, which takes an instance of the data structure and answers specific questions about the resulting class model instance – without constructing the whole of it. These questions are generally of three forms:

- Retrieve all objects of some class in the instance
- Retrieve the value of some attribute of an object in the instance
- From some object in the instance, navigate some association and retrieve all the objects at the other end of it.

If you have a selective XML reader, then an un-selective XML Reader can be implemented by making repeated calls to it, to navigate the graph or graphs in the class model instance.

To implement each of the three main functions of the selective reader, you have to use the appropriate type of mapping (object mapping, attribute mapping or association mapping), and do what the definition of the mapping tells you to do, as defined in this specification. So there are no very challenging design decisions in implementing an XML Reader, and the requirements are well-defined.

Because the meanings of mappings depend on the XPath standard, any implementation of an XML Reader will be built on top of an implementation of XPath. Given an XPath implementation, developing most of the XML Reader functionality on top of it is straightforward.

Once you have an XML Reader, several other useful applications can be built directly on top of it. These are:

1. Testing Data Translations
2. Model-based query
3. Model-based application development

These are described in the following sub-sections.

A further important application, running data translations, is enabled when you have an XML Reader and an XML Writer.

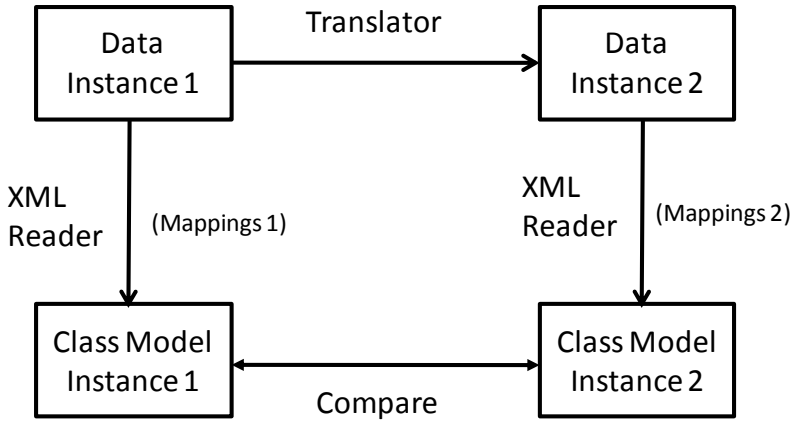
8.2 Testing Data Translations

Suppose you have an integration engine or translation engine, which translates data instances from one XML language to another. You need to test whether the translations are semantically accurate – i.e. whether they translate all data items correctly, preserving their precise meanings as in the class model.

If you have the mappings of both data structures onto the class model, and an XML Reader, this can be done fairly directly, with little other software development required. For instance, in the Eclipse Modelling Framework (EMF), supposing the translator you wish to test is from language 1 to language 2. You need to carry out the following steps:

1. Use the mappings of language 1 with the XML Reader, to convert the input data instance 1 (of language 1) to an instance of the class model, in EMF Ecore. This is class model instance 1.
2. Run the translation from data instance 1 in the translation engine. The result of this is data instance 2.
3. Use the mappings of language 2, with the XML Reader, to convert data instance 2 to another instance of the class model, in EMF Ecore. This is class model instance 2.
4. Class model instance 2 should be a precise subset of class model Instance 1. Use EMF tools such as the EMF compare tool to check this, and see what is missing from class model instance 2.
5. All information present in class model instance 2 should exactly match information from class model instance 1.
6. Any information missing from EMF instance 2 should arise only from missing mappings for language 2. (Missing mappings for language 1 are not detected in this test).

This is shown in the diagram below.



With further development effort you can build a powerful translation testing tool along these lines, to automate all the steps and report on discrepancies.

8.3 Model-Based Query Tool

Consider a simple object-based query language, which can express queries in terms of a UML class model. For instance, it would support the following query statement:

```
select Person.name Person.drives.registration where Person.drives.make='Fiat'
```

Here, 'name' is an attribute of class Person. 'drives' is an association from Person to Car, and 'registration' and 'make' are attributes of class Car. The query tells you the names of all people who drive Fiats, and the registration numbers of their cars.

The answer to this query is a small table whose columns denote attribute values from a class model instance, such as Person.name.

One way to answer the query (without any optimisation) uses the following steps:

1. Retrieve all objects in the class Person.
2. Get the value of the attribute 'name' of each Person.
3. For each Person, navigate the 'drives' association to find all Cars he drives.
4. Get the values of the attributes 'make' and 'registration' for each Car
5. Retain only those records where the make is 'Fiat', and write them out.

The steps (1) – (4) are the basic functions of an XML Reader. For any data source which is mapped onto the class model, you can use an XML Reader to answer the query, and write out the answer in terms of the class model. You can even answer the query from two or more data sources at once, in different XML languages mapped to the same class model, and compare the answers side by side in class model terms.

A benefit of this query tool is that queries can be written, and the answers understood, by someone who knows nothing about the data structures. The user needs only understand the class model that the structures are mapped to.

This query tool could for instance be used to make the comparisons in the previous use case, of testing data translations.

8.4 Model-Based Application Development

Just as the user of a model-based query tool needs to know nothing about the data structures that are mapped onto the model, so a developer for model-based read-only applications need know only about the class model, not about the data structures.

If such a read-only application proceeds by statements which get all objects in a class, find their attribute values, and navigate associations to other classes, then all of those statements use XML Reader functionality; they are just calls to the XML Reader API. Then, by using the XML Reader together with the mappings for some data structure, the same application can be used to read and process data from any data structure which has been mapped to the class model.

This style of application development can significantly reduce costs, by removing all the need to write code to handle the data structures. It also makes the applications much more future-proof, by ensuring that any new data structure can be used by the application, as long as it can be mapped onto the class model.

If the application is not read-only, it will require more than XML Reader functionality; it will also need to use an XML Writer.

8.5 Validating Mappings

The previous section set down a number of validation rules that mappings must obey. These rules can easily be used to build a validator tool, which tests all mappings against the rules, and reports on discrepancies.

The validator tool needs access to two important pieces of information:

- The XML Schema for the data structure being mapped.
- The UML class model it is being mapped to

The mappings need to be a valid bridge between these two pieces of data. To check that, the validator needs access to these at the same time as the mappings.

8.6 Creating and Editing Mappings

Mappings are tabular, with five core mapping columns and nine other extended columns which are used less frequently. The rules for the contents of each column have been defined in this specification. So it is always possible to create and edit mappings with a spreadsheet tool, possibly in conjunction with a validation tool to spot errors.

It may also be useful to have better tool-based support for creating mappings. A mapping editor tool, like a validation tool, needs to have access to the XML schema for the structure and the UML class model. It can then help the user select valid XPaths in the structure, select valid classes and features in the class model, and so on.

There is a further big advantage of creating mappings using a mapping editor which is integrated with an XML Reader in the same toolset. When making mappings, it is always useful to have some example instances of the mapped XML to hand. As you make the mappings, you use the XML Reader to turn those example instances into class model instances, and inspect them. This quickly exposes any problems in the mappings, where they do not have the effect you intended. A rapid, iterative **map-and-test cycle** is a very effective way to develop mappings.

Similarly, a static validator for the mappings can be integrated with the editor. This further helps detect any mistakes in mapping as soon as they are made.

8.7 DSL Creation Tools

These are tools for creating a simple, special-purpose XML Language (a Domain-Specific Language, or DSL), and creating its mappings onto a class model automatically at the same time. This is a particularly easy way to create valid mappings. As the tool creates the mappings automatically as the XML language is defined, the user does not even need to understand the mapping language in order to make valid mappings.

Suppose that you have some fairly complex class model which has been specialised to a domain. Instances of the class model contain the information needed for applications in the domain, but contain a lot else besides, because of the generality of the model.

Suppose also that the main associations in the model form a tree rooted at some entry class, and that there is a defined Implementation Technology Specification (ITS) to turn an instance of the model into an XML instance. This is the case for an HL7 V3 RMIM or CDA. What is wrong with just using the XML serialisation of this model as the specialised XML language?

What is wrong is that the XML is too deep and complex, because it matches the complex class model, node for node. If the domain-specific application needs to convey, say, 30 items of information, the deep XML will convey much more than that; those 30 items will be scattered about a broad and deep XML tree structure. What is needed is a much shallower XML, which conveys just those 30 items of information and nothing else.

A **Re-shaping Tool** creates this shallow XML, and creates its mappings onto the full class model at the same time. The tool first displays the full tree structure of the class model. The user selects the attributes he needs to include in the DSL, and renames them to give business names that are meaningful in the application. These attributes are still deep in an XML tree structure, joined to the entry node by long chains of associations. The user can 'collapse' these associations to make the XML shallower, or he can give them meaningful business names. So the nodes of the XML are selected, renamed and flattened until it has the form required for the application, and is much simpler than the class model.

Throughout all this process, the tool automatically keeps track of the mappings between the modified XML and the unmodified class model. Therefore the semantics of the simplified XML are precisely defined in terms of the full class model. Via these mappings, instances of the simple XML can be automatically transformed in either direction to or from instances of the full class model, or of any other XML that has been mapped to the class model.

A re-shaping tool can be downloaded from the HL7 GForge site.

8.8 Comparing Mappings

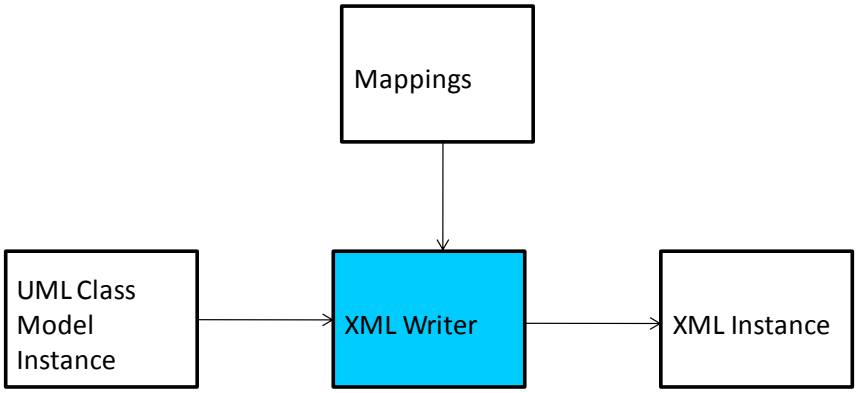
Given a precise definition of the mappings, it is possible to build tools to compare different sets of mappings – for instance comparing different versions of a set of mappings, to spot changes.

A different sort of comparison tool is a **cross-mapping comparison tool**. This takes two sets of mappings, of different data structures 1 and 2 onto the same class model, and identifies where some part of data structure 1 is mapped to the same part of the class model as some part of data structure 2. It identifies parts of the two data structures which are equivalent, in terms of the class model information they represent.

One way to regard this functionality is that it derives structure-to-structure mappings, from data structure 1 to data structure 2, from two sets of structure-to-model mappings. Structure-to-structure mappings are commonly used in commercial mapping and translation tools.

8.9 Writing and Translating Data Instances

The functionality of an XML Writer is summarised in the diagram:



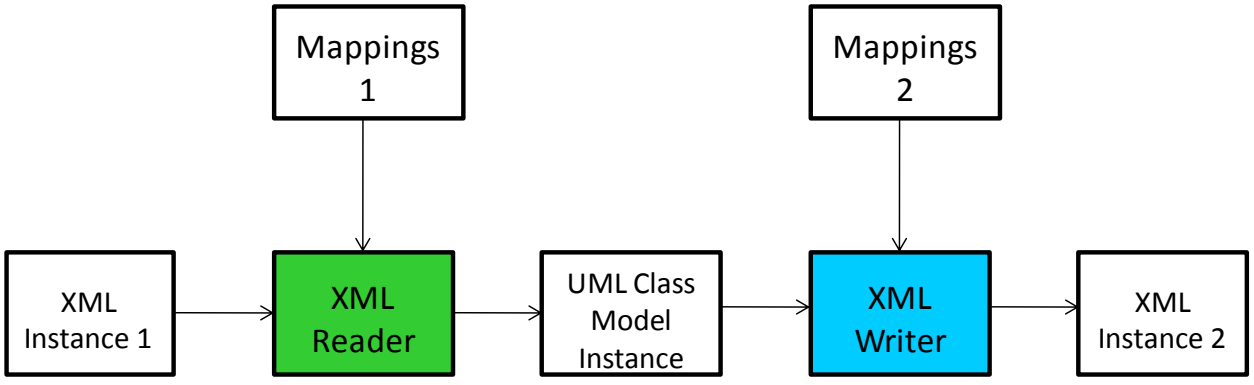
Its function is to convert an instance of the class model into an instance of the XML data structure.

Both selective and un-selective XML Readers have been discussed. One might imagine a selective XML Writer, with APIs to write only selected parts of an XML structure, but it does not seem very useful; so we only consider the unselective writer function, to write a whole XML instance from a whole class model instance.

It is not obvious how to develop an XML Writer direct from the definitions of the mappings, and it turns out to be much less straightforward than developing an XML Reader. Nevertheless it has been done. The implementation uses a compiler to convert mappings into a set of XML writing instructions, which are then used by a fairly simple run-time XML writer engine.

An XML Writer can be used to support model-based development of applications which are read-write, and not read-only. Such an application can be used to read any mapped data structure, and to write any other mapped data structure – which may or may not be the same as the structure being read. So one can develop data structure-independent and future-proof read-write applications in this way.

An XML Reader and an XML Writer can be chained together in an obvious way, to translate data instances from one data structure to another:



If the XML Reader is selective, this translation can proceed by 'lazy evaluation' – only creating parts of the UML class model instance as they are needed.

This method of data translation has several benefits over other methods:

- The translation is done automatically from mappings, and requires no hand-coding – reducing costs and errors.
- From N different sets of mappings (of N different data structures onto one UML model) it is possible to make N(N-1) different translations; a further cost-saving

- When this method is used to make serial translations such as A=>B=>A, there is a guarantee of round-trip consistency, in that information may be lost (if there are missing mappings) but it will not be changed by the round trip.

8.10 Bridges to Other Mapping Languages

The Open Mapping Language has been designed to have sufficient information in it to fully define the semantic relation between any XML language and a UML class model. Thus it has enough information to fully support the functions of an XML Reader or an XML Writer – to transform in either direction between XML instances and class model instances, or between XML instances in different languages.

The completeness of the language for these purposes has been tested for complex examples such as HL7 V2, V3, CDA, ASTM CCR, NHS LRA and relational databases. If further examples reveal any important gaps in the language, these can be remedied.

Therefore the language is expected to be near-complete as a semantic mapping language. As such, a set of mappings will contain all the information required to transform it to any other proprietary mapping format – as for instance used in proprietary mapping and transformation tools. It should be possible for the owners of those proprietary formats to develop software to read open mappings, and write mappings in their own formats. Therefore suppliers should be able to run their own translation engines from the open mappings.

Many proprietrary mapping languages are structure-to-structure mapping languages, and focus on mapping attribute values from one structure to another. When converting open mappings (structure-to-model mappings) to these proprietary mappings, this does not create a problem. If there are open mappings of two structures onto a common class model, these can easily be converted to structure-to-structure mappings between the two structures. Or if there is an XML Implementation Technology Specification (ITS) for the class model, mappings from data structures to the ITS structure can be created.

For most suppliers, conversion from open mappings to proprietary mappings will be the most important use case – because SDOs and healthcare procurement authorities will provide open mappings to define their requirements, and suppliers will want to run or test translations from those requirements. However, it will also be possible in some cases to convert mappings in the opposite direction, from proprietary to open.

It should also be possible to inter-convert between this mapping language and open source declarative model-to-model mapping languages, such as OMG's QVT Relational. This is currently being investigated.

9. COMPARISON OF THE LANGUAGE WITH REQUIREMENTS

The following table comments on how the language meets the requirements already defined for an HL7 standard mapping language. A fuller assessment depends on the further use of the language in projects.

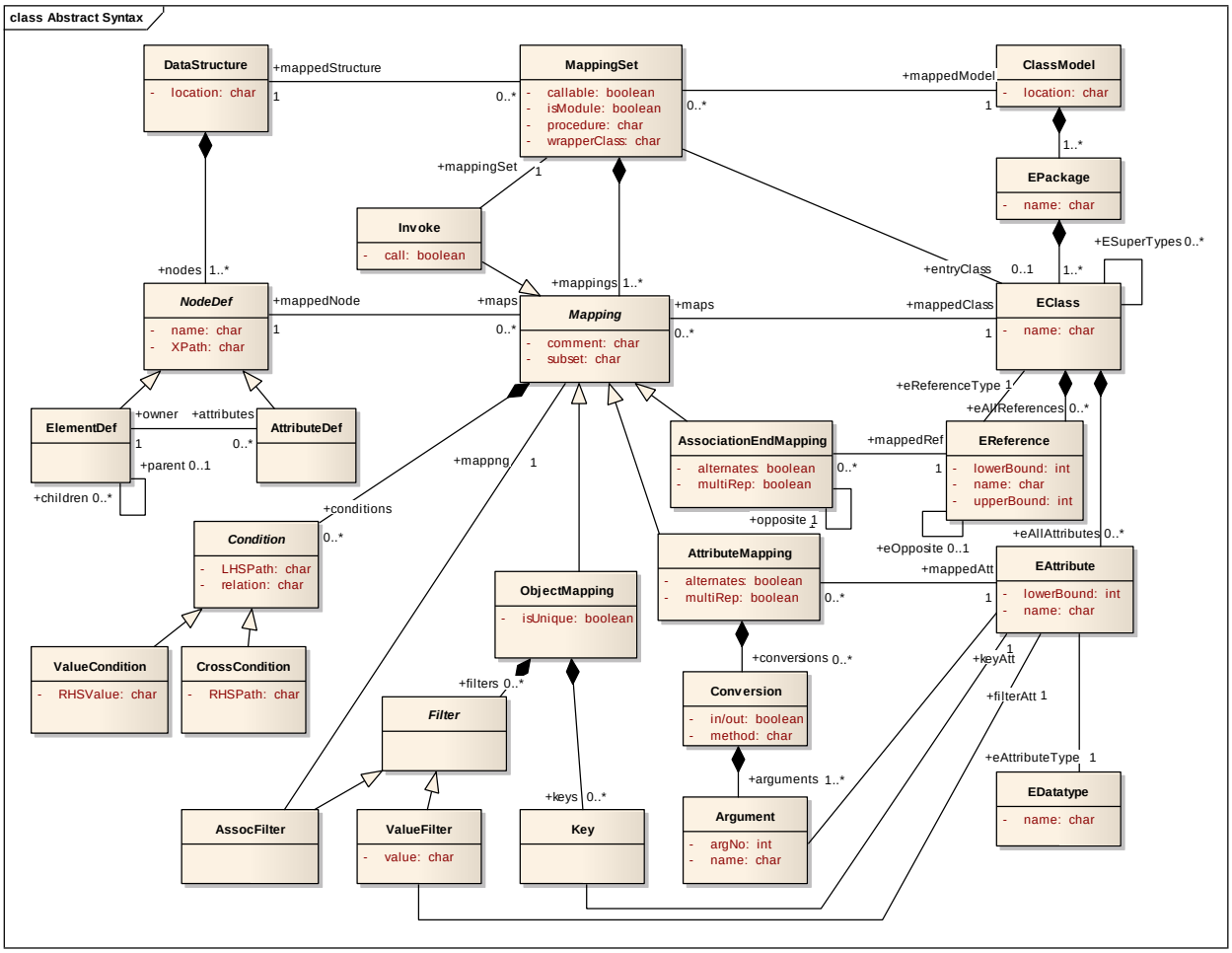
Reqt. no	Requirement	Comments on how the language meets the requirement
1	The notation should be linked to a Semantic Model	The language is a structure-to-model mapping notation, which maps data structures onto a UML class model. It can also be used for model-to-model mapping.
2	The notation should be semantically expressive. Sub-requirements of the 'semantically expressive' requirement include:	The language has been designed to capture all the common design patterns used to express class model information in XML or relational databases. Its capability to express these patterns has been tested in complex examples including HL7 V2, V3, ASTM CCR and relational databases. No major gaps in capability have been found, and there are procedural escape mechanisms to deal with specific gaps.
2a	Can express how associations are mapped (1:1, 1:N, N:M)	The language can map all these kinds of associations. Examples are given in the paper, section 4.5.
2b	Can express where value conversions are needed (e.g. different code sets)	Attribute value conversions are handled in the Convert_in and Convert_out columns for attribute mappings, described in section 4.4
2c	Can express mappings of lists of items	This has been addressed in section 4.5 under extended association mappings, and examples are given.
2d	It should be possible to note where mappings cannot be made, and why	The free-text Comments column can be used for this purpose. Conventions may be developed to identify different kinds of comment, so they can be searched for.
2e	The language must be able to map XML, formatted text, or relational database structures onto a semantic model	Mapping XML is the primary use case for which the language is intended. Relational databases are handled by mapping a simple flat XML derived from the database. Text files such as V2 bar-hat form are handled by first applying a wrapper transform from text to XML.
3	The notation must have well-defined meaning.	The meanings of the different mapping constructs have been described in the paper and illustrated with examples. The main test of a mapping notation is that mappings in the notation should support a working XML Reader (to convert XML instances to class model instances) and XML Writer (to convert in the other direction). The mapping tools on the HL7 GForge can do this for complex XML languages and class

		models.
4	The language should be easy to understand and review. Sub-requirements include:	The language uses a familiar tabular notation, similar to the spreadsheets which are often used for informal 'home-brew' mappings. Tables can be sorted on any column for convenience of viewing. The five columns used in the core mapping language are familiar to most people and have obvious meanings. The language requires three types of mappings – object, attribute and association mappings. The need for all three is not familiar to some people, but can be easily understood. The language uses the XPath notation widely, so familiarity with XPath is a help. The biggest intellectual hurdle is probably to understand extended association mappings, for instance when mapping many-to-many associations. This can be explained in a few minutes; see section 4.5.
4a	It must be possible to divide a large mapping set into convenient 'modules' of mappings, with clear definition of the relations between modules and provision for reuse of modules	Two kinds of mapping modules are supported: macro modules (a kind of in-line expansion of table rows) and callable modules (with a defined interface to the calling mappings, and information hiding) . See section 4.6.
4b	When more than one data structure is mapped onto the same semantic model, it should be convenient to view the parts of each data structure mapped to the same semantic item (cross-mappings between data structures)	Cross-mapping between data structures can be done by a simple join of rows in the mapping tables, joining on the class model columns (Type, Class, Feature and Filter)
5	The language should be easy to write.	Mappings can be created in any spreadsheet tool. The contents of each cell are not complex, but mappings must be precise. See requirement (9) below for specialised editing support, to make it easier to make mappings. There are specialised tools to create the mappings of an XML structure automatically, while designing the XML structure. These support a rapid map-test-map cycle, which is the best way to make precise working mappings. The main problems encountered in making mappings are semantic problems and ambiguities from the data structure, the class model, or a poor fit between them. Technical problems of mapping are small compared to these.
6	Mappings should have static validation criteria	Validation criteria exist and have been summarised in section 6.
7	Mappings should be declarative rather than procedural. But there should be procedural escape mechanisms.	The style of the mapping language is declarative and two-way, saying which nodes represent which types of information in the class model, rather than defining procedures to go from one to the other. Three procedural escape mechanisms are provided: • Procedural conversions between node values and attributes in the class model • Callable mapping modules, implemented as procedures • 'Wrapper transforms' on the data structure to make it

		easier to map, which can be implemented as procedures See sections 4.4 and 4.7.
8	The language should be executable.	The requirements for an XML Reader or XML Writer follow directly from the definitions of the mappings. These are the software to execute the mappings. Implementations exist for both an XML Reader and an XML Writer, based on the mapping language. Mappings can be used to execute translations in both directions between the data structures and the class model. The reader and writer can be chained to make translations between different data structures.
9	The language should have editing support	Specialised editors can be developed, using the definition of the language to make it easier to make mappings, validating and testing them incrementally. One such editor exists.
10	The language should have automated validation	Validators can be easily written to check the validation criteria, flagging errors and warnings. One automatic validator exists.
11	The language should be open and neutral, not proprietary.	The language defined in this specification is entirely open, and can be used by anyone. There are open-source tools to support it. It is possible to convert form the open mapping language to any proprietary mapping format, to use the mappings in proprietary translation engines. Limited conversions can be made in the other direction.

10. APPENDIX A: ABSTRACT SYNTAX FOR MAPPINGS

The abstract syntax of the mapping language is expressed as a MOF metamodel. The metamodel below has been developed in Enterprise Architect and converted to EMF Ecore, which has the same expressive power as Essential MOF (EMOF).



The class diagram is divided into three areas:

- The **left-hand side** of the diagram describes the **data structure being mapped**. While the mapped structure is an XML structure, this part of the mapping model is not the XML Schema metamodel, as used in the EMF XSD tools. A simpler model of node trees is sufficient for mapping, and the XSD metamodel would only add unnecessary complexity.

- The **right-hand side** of the diagram describes the **class model being mapped to**. This part of the model is the EMF Ecore metamodel; only the parts relevant to mappings have been shown. The Ecore metamodel is sufficient for mapping; having a fuller metamodel such as UML2.1 would probably only add complexity.
- The **centre** of the diagram describes the **mappings**, which are the bridge between the structure and the class model.

Outline descriptions of the classes, their attributes and associations follow in the tables below:

Data Structure Classes (left–hand side of diagram):

Class	Description	Attributes and Associations
DataStructure	This denotes the data structure being mapped – typically in an XML structure, or XML derived from a relational database	• **location**: the file location of the definition of the data structure, e.g. XML Schema • **nodes**: all Nodes (XML elements and attributes) in the data structure definition
NodeDef	The abstract superclass for definitions of XML elements and attributes. It describes a node type (uniquely defined by its XPath), not an individual node instance in an XML instance	• **name**: the name of the node • **XPath**: descending absolute XPath from the root of the document or mapping set to the node.
ElementDef	Definition of an XML element node	• **parent**: the parent element of this element • **children**: the child elements of this element • **attributes**: the XML attributes of this element
AttributeDef	Definition of an XML attribute	• **owner**: the element that owns this attribute

Ecore class model classes are more fully described elsewhere (e.g. see the help files for Eclipse EMF). The parts of the Ecore model most relevant for mapping (on the right-hand side of the diagram) are:

Class	Description	Attributes and Associations
ClassModel (not part of Ecore)	The UML/Ecore class model being mapped to	• **location**: the location of the file defining the class model, typically as EMF Ecore
EPackage	Ecore class for a UML package	• **name**: the package name • **eClassifiers**: all classes and data types in the package
EClass	Ecore class for a UML class	• **name**: the class name • **eAllReferences**: all references (ends of associations) owned by this class or its superclasses • **eAllAttributes**: all attributes of this class or its superclasses
EReference	Ecore class for one end of an association	• **name:** the role name to get from the owning (source) class to the target class • **lowerBound:** minimum number of instances of the target class at the end • **upperBound:** maximum number of instances of the target class at the end • **eType:** the target class

		• **eOpposite:** the EReference which is the other end of the same association as this EReference
EAttribute	Ecore class for a UML attribute. For mapping, we require the attribute to be single-valued (upperBound = 1) and have a simple type; otherwise it needs to be converted to an association (EReference)	• **name:** attribute name • **lowerBound:** if zero, the attribute is optional • **eAttributeType:** the data type of the attribute
EDatatype	Ecore class for a data type	• **name:** name of the data type

Mapping classes (centre of diagram):

Class	Description	Attributes and Associations
MappingSet	A whole mapping set or mapping module	• **isModule**: true if this is a mapping module, false if it is a main mapping set • **callable**: true if this is a callable mapping module, false if it is a macro mapping module • **procedure**: if non-empty, this mapping set or module is implemented in code, and the string identifies the class that implements the XMLReader and XMLWriter interfaces • **wrapperClass**: if non-empty, there is a wrapper transformation on the XML before mappings are applied, and the string identifies the class that implements the Wrapper interface • **mappedStructure**: the data structure being mapped • **mappedClassModel**: the class model being mapped to • **entryClass**: if the class model instances always have a single entry object of some class, the entry class. • **mappings**: all the mappings in this mapping set
Mapping	Abstract superclass of all mappings; with four subclasses: ObjectMapping, AttributeMapping, AssociationMapping, and Invoke	• **subset**: when there is more than one object mapping to the same class, a string which identifies them uniquely • **comment:** free text comments • **mappedNode:** the XML node involved in this mapping • **mappedClass:** the EClass of the object being mapped, or the object owning the EAttribute or EReference being mapped • **conditions:** any Conditions the node must satisfy. The node must satisfy the logical AND of all conditions, to represent what the mapping says it represents.
ObjectMapping	An object mapping, saying that some nodes with a given XPath represent objects of a class.	• **isUnique**: if true, any object, attribute or association is represented only by one node with the specified XPath. If false, the XPath may lead to several node instances, all of which represent the same object, attribute or link. • **keys**: only relevant if isUnique = false. Keys are attributes of the object (or of objects uniquely

		associated to it) which uniquely identify the object. • **filters**: Filters in the class model which the object must satisfy to be included in the class model instance
AttributeMapping	An attribute mapping, saying that nodes of a given XPath represent the values of an attribute, of an object of the class.	• **multiRep**: true of there are multiple possible representations of the same attribute, for the same class and subset, with different XPaths • **Alternates**: only used if multiRep = true. If true, only some of the alternate representations of the attribute may be populated. If false, they must all be populated. • **mappedAtt:** the EAttribute which the node maps to • **conversions:** attribute value conversions used with this mapping
AssociationEndMapping	An Association mapping, saying that nodes with some XPath represent links between objects of two classes. Association mappings must exist in pairs with the same XPath, for the two ends of an association – even if both ends are not navigable.	• **multiRep**: true of there are multiple possible representations of the same association, for the same class and subset, with different XPaths • **Alternates**: only used if multiRep = true. If true, only some of the alternate representations of the association may be populated. If false, they must all be populated. • **mappedRef:** the EReference (association end) that the mapped node represents • **opposite:** the association mapping for the opposite end of the association. Must always exist, even if the EReference has no EOpposite (i.e. if the association is navigable in only one direction)
Condition	A condition which nodes must satisfy in order to represent what the mapping says they represent. Abstract superclass for ValueCondition and CrossCondition. Conditions have a left-hand side (LHS), a relation such as '=', and a right-hand side (RHS)	• **LHSPath:** relative XPath from the mapped node to a node whose value gives the LHS of the condition. • **relation:** the relation used to compare the two sides of the condition. Will usually be '=', but other relations are sometimes used.
ValueCondition	Condition in which the RHS is a constant	• **RHSValue:** constant which is the right-hand side of the condition
CrossCondition	Only used for attribute mappings and association mappings, in which there is also a mapped object at an object node. The right-hand side of the condition is got by following a relative XPath from the object node	• **RHSPath:** relative XPath from the object node to a node whose value gives the RHS of the condition
Filter	Abstract superclass of the class model filters the object must obey to be represented in the XML. A logical AND of all filters is taken.	
ValueFilter	Filter requiring that the value of some attribute of the object must have some relation to a fixed value	• **filterAtt:** the attribute whose value is to be tested • **relation:** the relation used to test the attribute value against a fixed value • **value:** the fixed value

AssocFilter	Filter requiring the object to have some association to another object represented in the data structure	• **mapping:** the association mapping representing the association the object is required to have
Key	One of the key values used to uniquely identify an object, if it is not uniquely represented in the data structure (if the same object is represented by multiple nodes with the same XPath)	• **keyAtt:** the attribute which is part of the key
Conversion	A method or lookup table used to convert attribute values between the data structure and the class model, in either direction. Different alternate conversions may be supplied, e.g. in different languages – but should all have the same effect. For an in-conversion, the result is put in the mapped attribute by the XML Reader. For an out-conversion, the result is put in the mapped node by the XML Writer.	• **in/out:** if true, this is an 'in' conversion from values in the data structure, to a value in the class model; if false, it is an 'out' conversion from values in the class model to a value in the data structure • **method:** specification of the method (e.g. Java, xslt, or lookup table) used to make the conversion
Argument	One of the arguments of the attribute conversion. For an in-conversion, this must be a mapped attribute. For an out-conversion, it must be an attribute in the class model.	• **argNo:** argument position, 1..N • **mappedAtt:** the attribute whose value is the argument.

11. APPENDIX B: SUBSET OF XPATH NECESSARY FOR MAPPINGS

Where XPaths are used in mappings, it has not yet been decided whether or not to allow unrestricted use of any XPath expression in these places. My current preference is to make it fairly restricted, to keep the language simple, and to keep implementations of XML Writers and XML Readers simple.

Independent of this decision, we can define a small subset of the XPath standard, which is all that is **necessary** to define mappings. That subset is described here.

XPath expressions are used in three places in the mappings:

1. In the 'XPath' column of any mapping row (including those that invoke mapping modules)
2. In the 'Apex' column, to define the apex node of some relative XPath from one node to another, where that XPath is longer than the default shortest path.
3. In the 'Condition' column, as the left-hand side of any condition, or as the right-hand side of any cross-condition.

For uses (1) and (2), all that needs to be expressed is a purely descending absolute XPath from the root node, either of the XML document, or of the sub-tree mapped in a mapping module, leading to a node with a definite node name. The only axes that are needed for XPath steps are 'child::' and 'descendant-or-self::'. The only node tests that are needed are defined node names.

Therefore the usual form for this XPath is simply a list of node names from the root down to the selected node name, as in '/A/B/C' or '/A/B/@d'. The initial '/' denotes that it is an absolute path. No axis descriptor is needed for the default 'child::' axis, and '@' is used for the 'attribute::' axis.

It is also possible to use the '//' abbreviation for the 'descendant-or-child::' axis, at any step of the path after the first node. So the XPath or Apex column may contain expressions like '/A//D' or '/node()//D' to pick out any element of name 'D' in the whole XML tree.

Case (3), the relative XPaths used in the 'Condition' column, may be slightly more complex. These are generally short paths, as they need to pick out at most one node in the result, to give a unique value in the condition. However, they may need an ascending part, as well as a descending part. All the required XPaths can be expressed by using an optional single ascending step, followed by a number of abbreviated descending 'child::' or 'attribute::' steps.

For the single ascending step, one may either use the 'ancestor::' axis followed by a node name, or the '..' abbreviation for 'parent::node()'. Thus some examples of the relative XPaths used in conditions are:

- ancestor::A/B/@name
- ../D/@id
- D
- @id
- D/@id

Also for case (3), it is sometimes necessary to include predicates in the relative XPaths, to ensure that they always deliver only one node. This is necessary to give a unique value to test in the condition.

It is possible to write all the XPaths needed in any mappings , using just these constructs.

12. APPENDIX C: REPRESENTATIONS OF UML CLASS MODELS AND INSTANCES

A number of techniques are used to represent UML class models and their instances, both in executing programs and in persistent storage. Those techniques are not a part of the mapping language definition, and the mapping language can be used in conjunction with any of them.

However, for many issues of using the mapping language, including especially tool support, implementations of UML become important. This appendix discusses some issues which arise, using the Eclipse Modelling Facility (EMF) and its UML metamodel Ecore, to illustrate them.

12.1 Reflective and Hard-Coded Implementations

There are two main approaches to representing instances of UML class models in executing programs: reflective and hard-coded.

Reflective programs may not have to be recompiled to work with a different class model, because they effectively take the class model as data. Hard-coded programs do need to be re-compiled, because changes.

For instance, using Eclipse EMF to write an application involving the classes Person and Car, one may write reflective code using the EMF class EObject – which can represent an object of any class in the model – and EClass, which represents its class. Alternatively (non-reflectively) one can generate and extend code which has classes like 'Car', 'Person' in it, with EMF facilities such as persistence added to the usual Java facilities.

When working with mappings, the API of an XML Reader or XML Writer is naturally reflective. For instance, to ask for the value of some attribute, as represented in an XML instance, the natural form of call to the XML Reader is something like 'getAttribute("Person", "firstName")', which is reflective, rather than 'Person.getFirstName()' which is not.

To develop non-reflective mapped applications, one needs to embed the reflective calls to an XML Reader or Writer in non-reflective code. One could use code generation techniques to do so.

12.2 Persistence and Composition Associations

In the practical use of mappings and mapping tools, persistent forms of UML class model instances become important. For instance, a good way to test a set of mappings is to use the mappings with an XML Reader and an example data instance to create a class model instance, and then to inspect it, to see that the mapped objects and attributes appear as expected. So you need a persistent form of UML class model instance which you are familiar with and can easily inspect.

Eclipse EMF defines a persistent form for UML class model instances, and can even auto-generate an editor to inspect such instances. However, you may not always want to generate such a model-specific editor, as it is a bit of a performance to update it every time for model changes.

The form of a class model instance in EMF (whether seen through a dedicated editor or plain text editor) depends crucially on which associations in the class model are defined in EMF to be **containment**

relations (which is the EMF terminology for an association which is a **composition**). If the association between two classes A and B is a containment or composition, then instances of B can only exist inside instances of A – either in an executing program or in the persistent XML form. EMF likes to store its instances in a containment tree, with one root object and everything else contained inside it, directly or indirectly.

The mapping notation does not directly support the idea of containment, because although composition/containment is part of the UML metamodel, it is not strictly a part of a logical UML class model, but is more a physical choice for the storage of instances of the class model. In stead of containment, the mapping notation has the idea of inclusion filters, including the filter that an object of class B may not exist unless associated to some object of class A.

For instance, a relational database may represent an instance of some class model, where that instance does not have a single 'entry' object which everything else is connected to by associations which can be thought of as containment relations; so it would be awkward to try to represent the whole database as a single EMF Ecore instance. To do so, you may need to introduce some artificial 'top' object in a class such as 'database', which can contain all the others.

For the practical persistent storage and inspection of class model instances, it may be necessary to define the class model in a formalism such as EMF Ecore, and this will involve deciding which associations in the class model are to be containment relations. That decision centrally affects what the instances will look like.

12.3 Navigability

UML implementations such as EMF Ecore include the idea that some associations may be navigable in only one direction. This idea is part of the UML metamodel. However, from the point of view of mappings and mapped data structures, every association is navigable in both directions. An XML Reader will allow you to do that. Even when the class model gives you not role name to navigate in one direction, the default role name '' can be used to do so.