Methodius Design Document

V0.1

08 February 2006

Introduction

This document is the first step towards a design specification for Methodius. At present it is in early stages and will evolve as further areas of the design are worked out. At present it contains a rough overall architecture, some thoughts on storing and accessing the data and a list of areas still to be fleshed out.

Design Aims

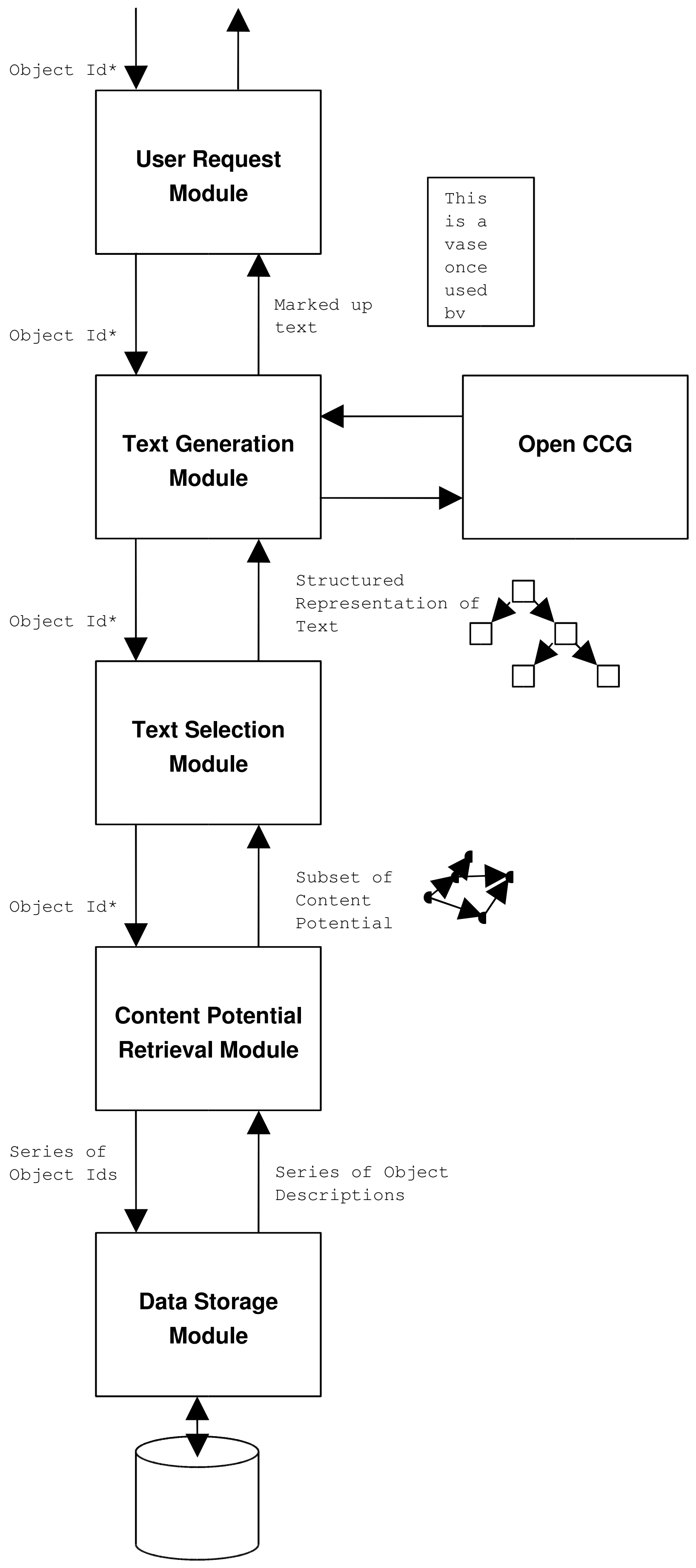
The aim of this design process is to describe an architecture for Methodius in sufficient detail that we can proceed to implementing it in the next phase of the project.

The design goals are:

Efficiency
Reliability
Scalability
Multi-user capability
Maintainability
…….

Overall Architecture

The overall architecture will be a hierarchical pipeline architecture. Hierarchical in the sense that each module only needs to know about the one below it. Pipeline in the sense that it's pretty much linear. At present it looks like each module will receive a request for information about an object, pass that request to the module below, receive the information it needs to work on from that module and then enhance that data in some way. First a diagram representing the architecture and then comments on it


Object Id*
User Request Module
This is a vase once used bv
Object Id*
Marked up text
Text Generation Module
Open CCG
Object Id*
Structured Representation of Text
Text Selection Module
Object Id*
Subset of Content Potential
Content Potential Retrieval Module
Series of Object Ids
Series of Object Descriptions
Data Storage Module

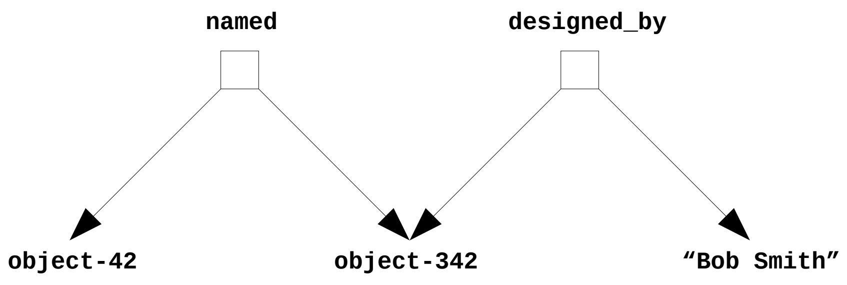
Domain Model

The domain model consists of two parts: the domain data and the domain semantics. The domain data relates to how the data is stored and retrieved. The domain semantics relates to how the domain data is interpreted.

Data Data and the Content Potential

The system works on top of a graph-based representation of the data called the content potential. The content potential is based on and Entity-Relationship model and contains objects, facts about objects which are two place predicates and relations between facts (and ….). In Ilex and M-Piro, this was generated as a whole from the underlying data at start-up time. It would be more efficient to generate the content potential as part of a system setup phase which is performed once and then to persist the content potential itself. Furthermore, rather than holding the whole of the content potential in memory, we want to be able to work on a subset of the content potential localised around some object of interest.

For example, if we have an object designed by a person, this might be represented as



If this was a single level hierarchy, we could store this in a simple relational way such as

```
Object 42
Designed_by: "bob smith"
Year_of_manufacture: 1956
……
```

But in this case, because we want to be able to say things about the facts themselves and because we want to be able to go in both directions through the relationship i.e. from the object to the designer and from the designer to the object, this is not sufficient. In order to be able to say things about facts, they need to be entities in their own right which can take part in further relationships with other facts. In order to be able to move both ways through the relationship we need to preserve the graph structuring in way that will allow this and, finally, if we're not going to load the whole content potential in one go, we need a way to load part of the graph and stop. The following describes one possible way to do this.

Proposed Data Storage Design

The data will be stored in a tabular format in some database system (to be defined but perhaps Sleepycat's Java implementation of Berkeley DB). Each entity in the data will be uniquely named. This unique name can be generated as part of the content potential creation. Objects will contain a list of facts in string form. Facts will contain the id's for the objects they relate, the identifier of the predicate they represent and possibly other relationships (generalisations, comparisons etc)

```
Object-42

Class: pendant-cross
Fact-1032
Fact-86
Fact-408
```

```
Fact-1032

Pred: designed_by
Arg1: object-42
Arg2: object-324
```

```
Object-324

Class: person
Fact-378
Fact-854
```

These can be stored in string form, exactly as represented here. When we want to load Object-42, we can load it from the database as a single object containing four strings. When we want to expand its context we can load Fact-1032 and replace the string in object-42 with a reference to the newly loaded fact. Likewise within the fact the Arg1 can be replaced with an object reference. This process can be viewed as

Load Object-42

```
Object-42

Class: pendant-cross
Fact-1032
Fact-86
Fact-408
```

Load Fact-1032

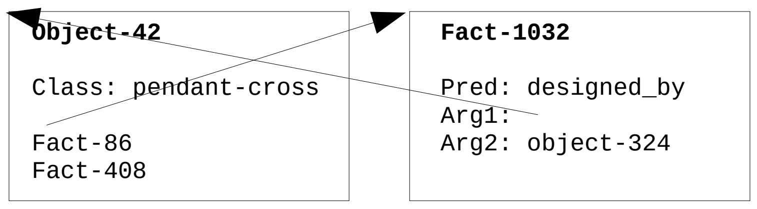
```
Object-42

Class: pendant-cross
Fact-1032
Fact-86
Fact-408
```

```
Fact-1032

Pred: designed_by
Arg1: object-42
Arg2: object-324
```

Replace strings with object references

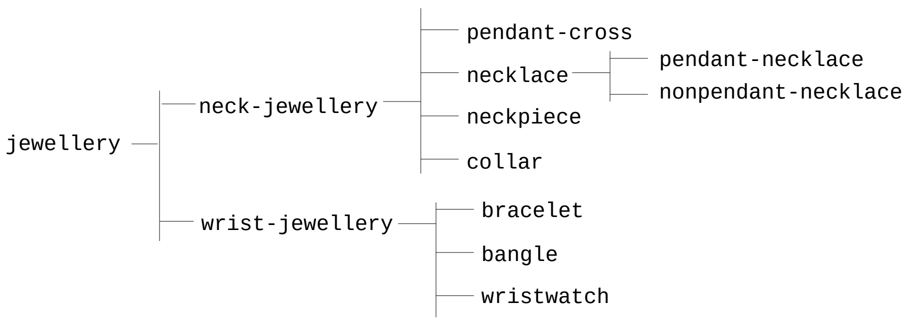


This gives us a way to store the content potential in a fairly simple way from the point of view of storing the data in a database but also a fairly natural way from the logical view of the content potential. It gives us the bi-directionality we want: from the object to the fact and from the fact to the object. It gives us a way to incrementally load the content potential from the database. Finally, the overheads in terms of disk storage are pretty small.

This scheme is also extensible to other relationships such a generalisations, defeasible rules etc. (still to be worked through ….)

Domain Semantics

The domain semantics define the types of objects in the domain data, the predicates which can link those objects and the hierarchy of objects. The objects are related in a hierarchy, for example,



Object Definition

Each object belongs to one class in this hierarchy. Each class in the hierarchy has a set of facts which are (can be?) stored about it. For example, the jewellery class may have facts of type

```
designed_by
 made_for
   date
  style
  place
```

which it can have. The class neck-jewellery may then also have these facts but in addition may have a circumference. Then a pendant-cross may have all the facts of neck-jewellery but also a style-of-cross fact.

So we need to be able to specify the facts which each class of object can have and the hierarchy between them. In essence the hierarchy is a single inheritance hierarchy.
(Do we want compulsory and optional attributes?)

The hierarchy and the attributes types can be defined by a class specification e.g.

```
class jewellery {
        designed_by
        made_for
        date
        style
```

```
                place
        }
```

specifies that an object of type jewellery can have attributes (or facts or predicates) of type designed_by, made_for etc

```
        class neck-jewellery is jewellery {
                circumference
        }
```

specifies that neck-jewellery can have any of the attributes of jewellery (e.g. designed_by) but can also have a circumference attribute.

```
        class pendant-cross is neck-jewellery {
                style-of-cross
        }
```

specifies that an object of type pendant-cross can have any of the attributes of jewellery or neck-jewellery or an additional attribute style-of-cross.

Predicate Definition

Predicates have two special attributes (arg1 and arg2) and the types of the objects these can reference are specified e.g.

```
        Predicate designed_by {
                arg1: jewellery
                arg2: person
        }
```

In addition to this, any object of this predicate type can also include any of the abstract relations e.g. generalisations, comparisons, defeasible rules etc etc.

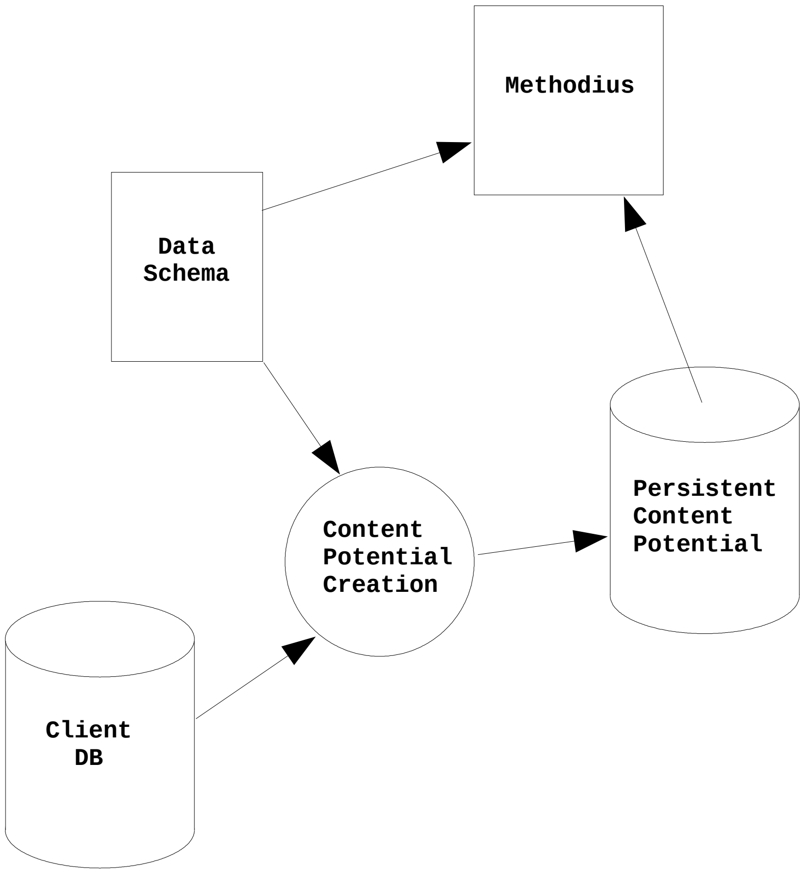
Linguistic data?
what do expressions look like – in predicates and in objects?

Storing Semantic Information

The semantic information will be stored in a root schema file. This in turn may reference other files in which the object definitions, predicates and linguistic information is stored. This file will be loaded at start-up time and parsed into a suitable internal form to allow for data integrity checking and language generation.

Creating the Data Store

The data store in methodius is essentially the content potential made persistent. This means that it is not the data in its original form but is put into a particular form (described above) and also augmented. This persistent content potential will be created from the data provided by client databases and the semantic description.



The process of content potential creation will be, at least to some extent, a one-off process for each client domain. However, we can expect that a number of tools will emerge which will provide significant functionality in this area. These are not defined here.

Forward Comparisons

One significant addition to the Persistent Content Potential is the addition of forward comparisons. being forward comparisons, these are not dependent on the path taken by the user (unlike backward comparisons). These can therefore be added statically to the persistent store when it is created.

Content Potential Retrieval

The content potential retrieval will be passed an object identifier which is the focal object of the description to be generated. This object will be retrieved from the Persistent Content Potential along with all objects within a fixed distance of the focal object. This fixed distance will be configurable but, at present, we expect it to be very small e.g. 3. So the result of the content potential module is a graph of objects made up of entities, predicates and relations.

Text Selection

The Text Selection Module is given a focal object identifier and a number of facts wanted. The number of facts is expected to be small (5-10). Algorithm – Comparisons, forward and back.

Text Realization

Text Output

Input and Return Parameters

The input to each module will be an object id. We're pretty sure about that. It seems likely that there may be other input at different levels e.g. user profile data. On the return path the parameters will be gradually evolving towards text. The exact formats are still to be decided.

Comparisons

There are two types of comparisons: forward and backward. Backward comparisons are with object already seen. This will be implemented using a queue of seen items. This queue can be fixed size and probably relatively short. The forward comparisons are, by their nature, not dependent on the path taken so far. They are therefore static in nature and so could be built into the stored content potential. This would require one or more passes over the data but would be very efficient in performance terms.

Description of the Process

A user request comes to the user request module probably in the form of an object reference. This is passed down through the calling chain to the Content Potential Retrieval module. The CPR module will pull out a subset of the content potential centred around that object. This could be something like the object and everything within 3 links from that object.

Based on this subset of the content potential and on the comparisons, the text selection module will decide what to say and build some sort of representation of this. This may be some sort of graph structure.

This structure specifying what will be said is then returned to the Text Generation or Text Realisation Module. This will package the information on what to say into a form for CCG pass this to CCG and get some structured text back, probably XML. This is then returned to the User Request Module which will convert it into whatever form is desired for delivery to the user.

Support for Authoring Tools

in the assessment of exiting software, we identified that a single, all purpose authoring tool for end-users was not a desirable objective. However, it is inevitable that in the process of setting up test domains for Methodius, we will develop tools which will address some of the requirements of an authoring environment. These tools will form a workbench from which a future authoring tool or (more likely) authoring tools can be built. It is anticipated that these tools will hook into various API's in Methodius and provide ways to manipulate the data at that level.

Design Process (decisions still to be made)

The design process will be a gradual refinement of this document, making decisions about data structures, interfaces between modules and processes which take place in each module. This will be an iterative process. We will make one pass through the system from bottom to top filling in the first level of detail and then consider multi-user issues. This is likely to force a degree of rethinking. Then a series of further iterations should take us to the final design.

Issues we need to decide on still

User identification
User Modelling
Multi-user capabilities
Data formats at each stage
Algorithms at each stage