Exploiting the advantages of object oriented programming in the implementation of a database design environment

Nigel Stanger
Richard Pascoe

*Department of Information Science,
University of Otago
Dunedin, New Zealand.*

*Email
<nigel.stanger@stonebow.otago.ac.nz>
<rpascoe@commerce.otago.ac.nz>*

[*Submitted to the Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC '97 and ICSC'97), December 2–5 1997, Hong Kong*]

Exploiting the advantages of object oriented programming in the implementation of a database design environment

Abstract

In this paper, we describe the implementation of a database design environment (*Swift*) that incorporates several novel features: Swift's data modelling approach is derived from viewpoint-oriented methods; Swift is implemented in Java, which allows us to easily construct a client/server based environment; the repository is implemented using PostgreSQL, which allows us to store the actual application code in the database; and the combination of Java and PostgreSQL reduces the impedance mismatch between the application and the repository.

1. Introduction

In this paper, we describe the implementation of a database design environment named *Swift*, which has these novel aspects:

- Swift's data modelling approach is derived from viewpoint-oriented methods;
- Swift is built to take advantage of recent developments in object-oriented programming, in particular the advent of Java; and
- Swift itself will be stored in an underlying database that acts as a repository.

In the remainder of this section, we shall expand briefly upon these three points, the first in section 1.1 and the remaining two points in section 1.2. In section 2 we discuss the framework for the implementation of Swift. In section 3 we describe the implementation as it currently stands. In section 4 we discuss some useful side effects of our implementation approach, and in section 5 we discuss directions for future research.

1.1 A brief review of viewpoint concepts

The underlying philosophy of Swift is based on the concepts of perspectives, viewpoints, representations, techniques and schemes, as shown in Figure 1, and suggested by Finkelstein (1989), Easterbrook (1991a) and Darke and Shanks (1995).

A *perspective* is a description of some real-world phenomenon that has internal consistency and a specified focus (Easterbrook 1991). During the requirements definition phase of systems analysis, developers often encounter many different perspectives on the problem being modelled. Perspectives may overlap, or even conflict with each other, and part of the process of database design is deciding how to deal with these multiple perspectives. This is an active area of research that has been discussed by several authors (Leite and Freeman 1991; Easterbrook, Finkelstein et al. 1994; Kotonya and Sommerville 1996).

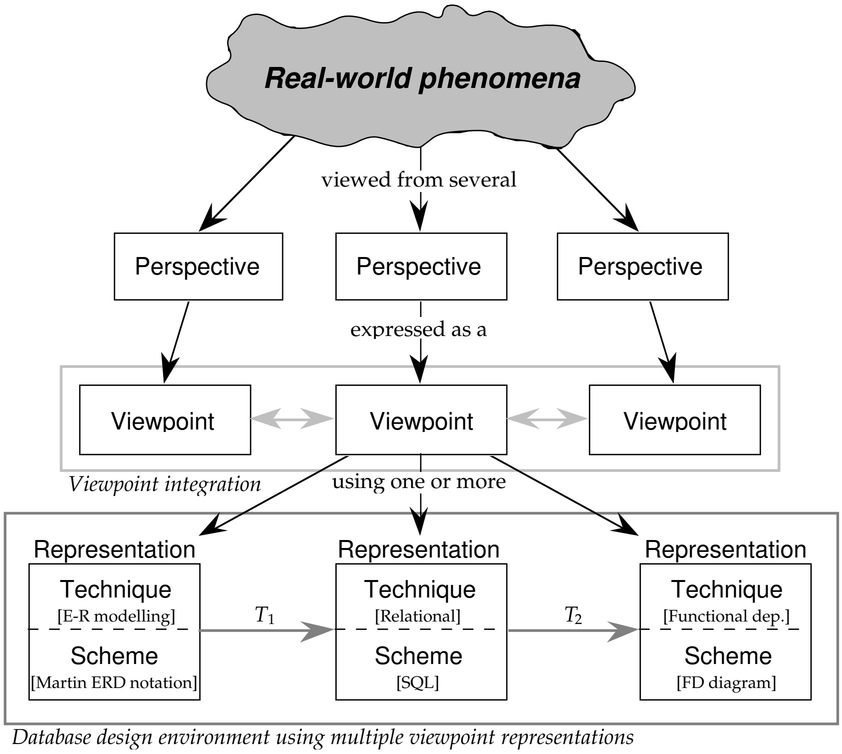

Figure 1. Perspectives, viewpoints and representations.

A *viewpoint* is a formatted expression of a perspective (Finkelstein, Goedicke et al. 1989). Darke and Shanks (1995) define two main types of viewpoint, *user viewpoints* and *developer viewpoints*. These viewpoints may be described using various *representations*, each of which comprises a *technique* expressed in some notation or *scheme*. A technique may have one or more associated schemes, but we define each combination of a technique and a scheme to describe a viewpoint to be a distinct representation. For example, the relational model (RM) is a technique, with SQL and QUEL being two possible schemes, but the combinations RM + SQL and RM + QUEL form two distinct representations. Figure 1 does not show techniques with multiple schemes for the sake of clarity.

The use of a particular representation to describe a viewpoint can be said to form a *description* of that viewpoint. In general, no single representation will be adequate to fully describe all types of viewpoint, and indeed, the current plethora of modelling techniques suggests that a single representation is inadequate to fully describe even a *single* viewpoint. The approach taken by Swift is to use multiple representations to form multiple descriptions of a single viewpoint, which, when combined, result in a more complete description of the viewpoint than that formed using a single representation. This approach conceptually mirrors the idea of using multiple viewpoints to describe a phenomenon, but at a lower level.

Much of the research into viewpoints has been from a software engineering perspective. Our work started in the area of data modelling and the main focus remains there, but we have found that the viewpoint concepts described above provide a useful framework for our work.

The use of multiple representations within a particular viewpoint is an area that has only recently begun to receive attention (Darke and Shanks 1995). One interesting aspect of Swift is that it allows the developer to perform transformations between different representations (and by extension, descriptions), for example, from functional dependencies to an entity-relationship diagram (ERD). This provides some useful advantages in terms of re-use of existing models, and potential improvements in the integrity of the overall design (Stanger and Pascoe 1997).

1.2. General overview of Swift

Swift consists of two main parts: a Java applet front end and a PostgreSQL (formerly known as Postgres95) repository. Java is a recent development in object-oriented programming, and use of this language provides us with some useful benefits:

- Swift is potentially fully cross-platform, as the applet may be run in any Java-capable browser or other Java run-time environment.

- Swift can be implemented as a client/server system, that is, the applet acts as a client to the repository. Ideally, the front end will run on a separate machine from the repository.

Java is designed around the idea of small classes that may move around on a network. One immediate benefit this provides is the ability to extend the client to support a new representation by adding classes at the server end that implement the new representation. These classes are loaded at run-time by the client. This is conceptually somewhat similar to the approach taken by Informix's DataBlades (Informix Software 1996; Keeler 1996).

This leads us to the repository, which is built using PostgreSQL. PostgreSQL was chosen because of its object capabilities (particularly in the handling of large objects), its temporal features and ready availability. PostgreSQL's ability to handle complex object types is also useful in reducing the impedance mismatch between the client applet and the repository, that is, the data structures used in the repository closely mirror those used by the client. We shall discuss this further in section 4.

The client and repository communicate via JavaPostgres95 (McLean, Medeiros et al. 1997), which was originally developed as a Java implementation of PostgreSQL's *libpq* programming interface (Yu and Chen 1995), but has recently evolved into a full JDBC driver. This allows Java programs to easily access data stored in a PostgreSQL database.

The combination of Java and PostgreSQL's large object features gives us the opportunity to store all of Swift's code in the database itself. This is particularly useful for the classes that define a particular representation's behaviour, as these are not loaded until they are actually needed, but it is equally feasible to store other parts of Swift in the repository as well. Using PostgreSQL's large object support, we can store compiled Java classes as attributes within the repository. In addition, Java applets are usually accessed through an HTML page, and this page could also be served directly from the repository.

2. Conceptual framework

The main feature that differentiates Swift from other database design tools is the ability to transform between representations/descriptions. This approach provides two main advantages (Stanger and Pascoe 1997):

- existing descriptions may be re-used by transforming them into other representations; and

- transformations can be used to verify the integrity of a design by transforming different descriptions to use the same representation and then comparing them.

A description in one representation can be transformed to another representation as a starting point for documenting different aspects of the same data model. Note that both share a common basis, thus forming a cohesive model containing different descriptions. For example, a developer might create a FDD, then transform that into a partial ERD that they can then use as a foundation for further work. Because the ERD is based on the original FDD, there is a greater likelihood that the two will be consistent with each other.

Now consider a situation where we have several descriptions of the same viewpoint, and we want to be sure that the these descriptions are consistent with each other. In other words, we want to check the integrity of the combined description. Re-using existing descriptions helps to some extent, as described above, but we may also have descriptions that were produced "from scratch". One way of checking consistency is to transform the descriptions to be compared so that they all use the same representation, for example, we could transform a FDD into an ERD so that it can be compared with an existing ERD. If the two ERDs are not consistent, then there may be a problem with the integrity of the design.

2.1 Implementing transformations

One possible approach to implementing these transformations is to attempt to define a uniform representation that acts as a kind of "interchange format", otherwise known as the *interchange interfacing strategy* (Pascoe and Penny 1990). All information is described using this representation, and it is transformed to other representations as required. The main disadvantage is that the uniform representation may become a "moving target" — as new representations are added, the uniform representation must be updated to handle them. This can lead to a proliferation of incompatible versions of the uniform representation that hinders transformations between different representations.

Another argument against this strategy is that it represents a kind of "representation integration" step, which is analogous to the objectivist approach of unifying all viewpoints into a single "correct" viewpoint (Klein and Hirschheim 1987). In other words, we are conceptually moving away from the viewpoint-based approach.

Also consider that the end result of the database design process is usually the generation of a database schema. If we are going to take our multiple representations and transform them into a single representation anyway, it seems somewhat unnecessary to introduce yet another representation into the mix, especially if the "target" representation is one of the original set of representations (for example, SQL).

A better approach in this case seems to be to perform transformations between representations as needed, an approach known as the *individual interfacing strategy* (Pascoe and Penny 1990). The total number of transformation "engines" required increases, but adding a new representation does not impact upon existing transformations. Also, as stated above, this approach is analogous to the way in which viewpoints are handled.

Figure 2 illustrates a framework that follows this strategy. The transformations between representations are represented by the grey arrows; T_1 and T_2 correspond to the similarly labelled arrows in Figure 1. Each representation is placed into its own conceptual “module”, which handles all the needs of that representation. This modular approach makes the environment much easier to extend.

Each module has a *technique component*, which deals with storage issues, and a *scheme component*, which deals with the user interface. Technique components can be shared across representations, as shown by the relational technique in Figure 2, which is shared across two relational representation modules, one that uses SQL and another that uses QUEL.

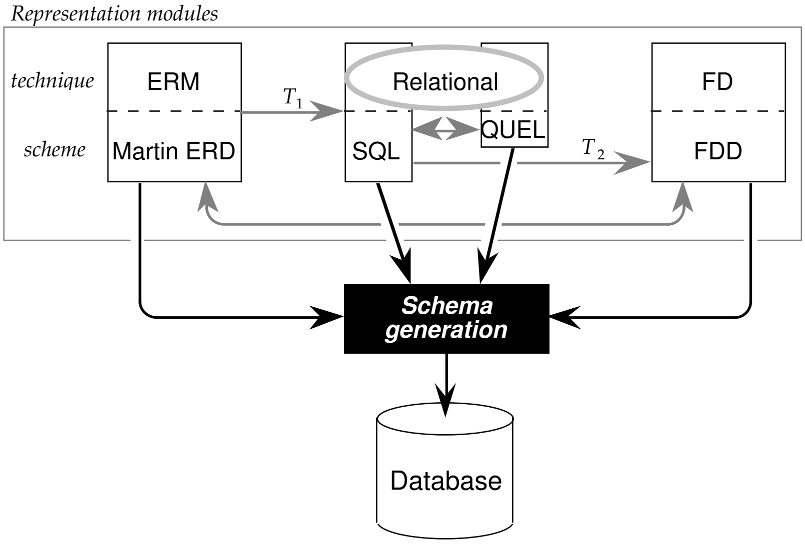

Figure 2. Conceptual framework for the database design environment.

3. Implementation architecture

There are four distinct parts to the Swift environment: a modelling applet, a transformation processor, a repository and a schema generator. We shall discuss each of these parts in turn in this section, apart from the schema generator which is briefly discussed in the future research section of this paper.

Figure 3 shows how these different parts fit together. We can relate this architecture back to the framework of Figure 2 as follows:

- The combination of the modelling applet with a particular R corresponds to one of the representation module boxes of Figure 2.
- The combination of the transformation processor with a particular T corresponds to one of the transformation arrows T_i of Figure 2.
- Schema generation is not shown in Figure 3, as stated above.

These parts could be implemented either as a single integrated module, or as separate modules that communicate with each other. Since Swift is intended to run in client/server mode across a network, we have chosen to implement the parts as separate modules. The modelling applet is designed to run within a browser on the client machine, and communicates with the repository on a server machine that may be elsewhere on the network The transformation processor may run on either the client machine or the server machine, or even both, depending on user preference and prevailing conditions.

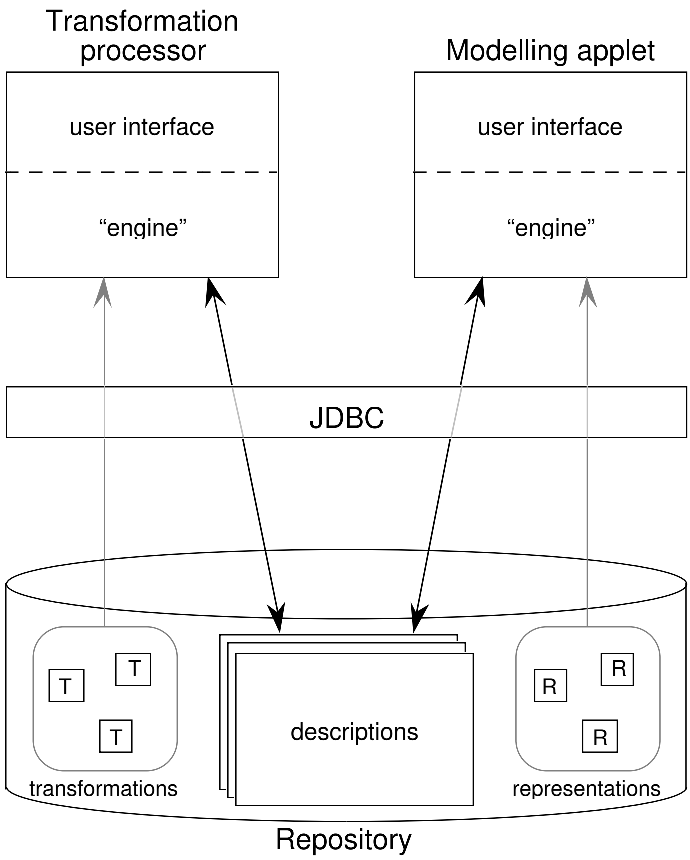

Figure 3. The implementation architecture.

3.1 The modelling applet

The *modelling applet* is used to manipulate and display constructs of a particular representation, and can be implemented in two ways:

- create a customised module for each representation; or
- create a single, generic modelling module and create separate "plug-ins" for each representation.

The first approach has been implemented in several CASE tools, for example Sybase's Deft (O'Brien 1992), and has the advantage of allowing us to tailor the front end (in particular the user interface) for each representation, but it is only really practical if there are a small number of representations, as writing a custom module for each new representation is likely to be a labour-intensive task. It is also possible that adding a new module may require modifications to existing modules to allow it to "fit in". The end result is an architecture that is less easy to extend.

The second approach, by contrast, allows us to more easily extend the environment's capabilities, and is an approach that has also been implemented in several CASE tools, for example, Evergreen Software's EasyCASE (Evergreen Software Tools 1995). In the case of Swift, we can use the inheritance and specialisation aspects of the object-oriented paradigm to implement this approach. In particular, aspects of a specific representation are implemented as subclasses of a collection of abstract representation classes, which allows us to add a new representation without having to rewrite existing code. Instead, the modelling applet gets the representation-specific information (such as terminology and notation) from the representation subclass and uses this to specialise its behaviour accordingly.

This architecture is more flexible and extensible than one based on the first approach, and is the approach we have taken for Swift — the modelling applet has a collection of abstract representation classes that are overridden by representation-specific subclasses that are loaded from the repository. These subclasses know how to manipulate and draw the elements of that particular representation.

Figure 4 shows two screenshots of a very early version of the modelling applet running in the HotJava browser. The screenshots show two different partial descriptions of the same viewpoint, a functional dependency diagram on the left, and an entity-relationship diagram on the right.

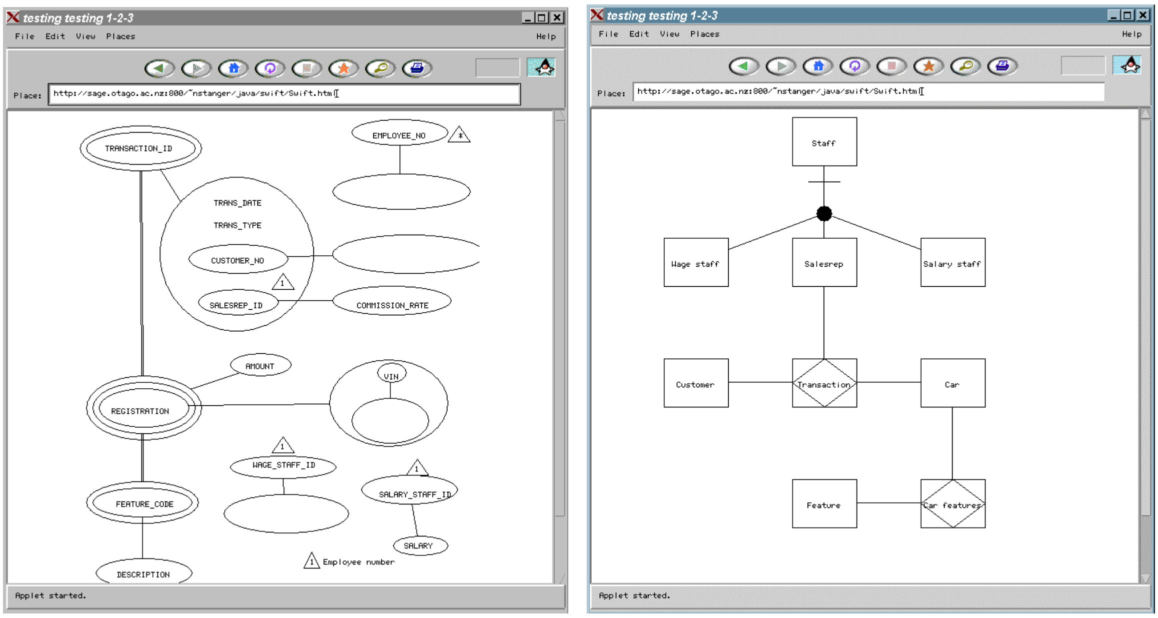

Figure 4. Screenshots of the modelling applet.

3.2 The transformation processor

We can also apply the subclass/specialisation approach to the *transformation processor*, which is used to initiate and control transformations between representations. Indeed, the subclass/specialisation approach is even more applicable in this case, as each new representation requires the addition of a potentially large number of new transformations. That is, if we have *n* existing representations, and we add a new representation, we must potentially create *n* new transformations. The subclass/specialisation approach would seem to be the most effective way of implementing this. Each transformation description would consist of a set of rules describing how to transform objects of the source representation into objects of the destination representation.

This is the approach that we have taken for Swift — transformations are implemented by a collection of Java classes that specialise the behaviour of abstract transformation classes. Each element of a representation has a "transform" method, so the transformation "engine" tells each element to transform itself in turn, and then collects and assembles the results for display by the modelling applet.

As stated earlier, the transformation processor could run on either the client machine or the server machine. The user interface for initiating and controlling transformations must obviously run on the client, but the actual transformation process itself could be carried out anywhere.

3.3 The repository

The repository is a persistent data store for all data relating to a particular viewpoint, storing everything from the representation used by a description, to the structure or syntax of a particular description (which may include such things as entities, data flows, attributes and so on). Ultimately, it is intended that the repository will also store Swift itself.

There are two ways of implementing such a repository: either *internally*, that is, build a set of custom data structures and files for storing the repository data; or *externally*, that is, use pre-existing DBMS software to store the repository data. The latter is the preferable option, as it reduces the amount of work required to implement the repository. The only real disadvantage is that the performance of a DBMS-based repository may not be quite as good as that of a custom-built repository that is tuned specifically for handling this kind of data, but this is far outweighed by the implementation advantages gained. It is also worth noting that some CASE tools already follow this approach. For example, EasyCASE stores its repository data in dBASE III Plus database files (Evergreen Software Tools 1995).

Four choices were available for implementing the repository: Oracle 7.2 running under Windows NT; PostgreSQL or Interbase running under Solaris; and Rdb 6.2 running under OpenVMS. The choice of Java as the implementation language immediately removed Rdb from the list, as there was no infrastructure in place for communicating between a Java applet and Rdb. The next best options were Oracle and PostgreSQL. PostgreSQL was chosen over Oracle because of its object capabilities (as noted earlier) and the availability of source code, which allows the potential for further tailoring of the DBMS in addition to PostgreSQL's abstract data type facilities. Interfaces were also readily available to allow Java to communicate with PostgreSQL.

The repository itself is currently designed to be as generic as possible. Representation-specific information is encapsulated within the representation classes, and it is up to the modelling engine to make sense of the repository data based on the rules encoded within these classes. This makes the repository capable of holding information for a new representation without having to alter its structure. In other words, the repository stores only the *syntax* of a description — the semantics is provided by the representation class. Figure 5 shows an entity-relationship diagram of the repository structure.

The repository stores data about four main types of object:

- *Projects*, which are just a means of grouping together a collection of related objects (descriptions, items, and so on) into single logical unit.

- *Descriptions*, which were discussed in the introduction, for example, an entity-relationship diagram expressed using Martin notation, or a collection of functional dependencies written in the standard $A \rightarrow B$ notation.

- *Graphic items*, which are the "visual" elements of a representation (for example, entities, data flows and attributes). Graphic items are a specialisation of the generic *Item* class, and may be specialised further into three subclasses: *symbols* (for example, entities or data stores); *connectors* (for example, data flows or relationships) that connect symbols together; and *text* blocks.

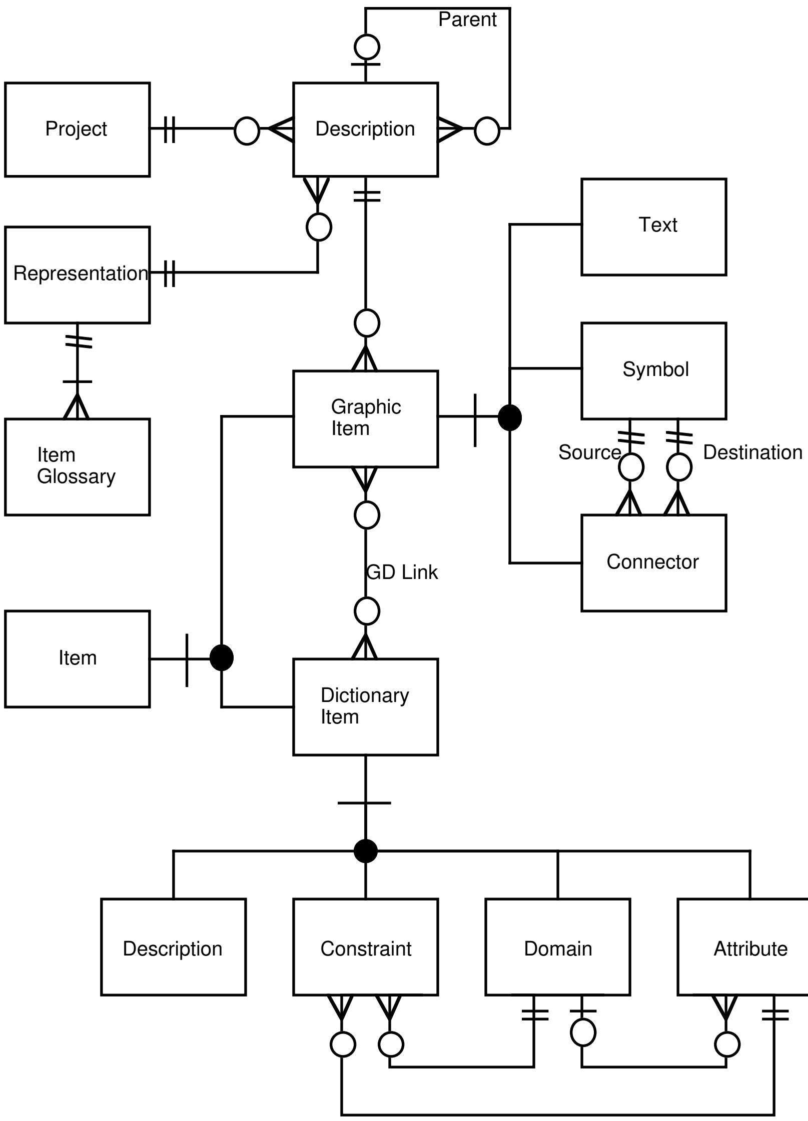

Figure 5. Logical structure of the repository.

- *Dictionary items*, which are the "non-visual" elements of a representation (for example, attributes, domains, constraints). Dictionary items are also a specialisation of the Item class, and may be specialised into at least the following subtypes: *attributes*, *domains*, *constraints* and *descriptions*. Currently the environment only implements attributes.

4. Reducing impedance mismatch

Three languages were considered for implementing Swift: C++, Java and Tcl/Tk. Each of these had its strengths and weaknesses, but we have chosen Java for the following reasons:

- it is relatively less complex to develop in than in C++;

- Java is truly cross-platform (there are some compatibility issues with different virtual machines, but we do not consider this a major problem);

- Java is sufficiently similar to C++ that the learning curve is relatively small;

- using Java allows us to make Swift a client/server application, as discussed earlier; and

- the combination of Java and PostgreSQL has the useful side effect of reducing the *impedance mismatch* between the client applet and the server database.

Impedance mismatch comes about because the way data is structured and manipulated in a DBMS usually differs from the way data is structured and manipulated in typical programming languages such as C++ (Cattell 1991). The combination of Java and PostgreSQL used to implement Swift has had the useful side effect of reducing the impedance mismatch between the modelling applet and the repository, as PostgreSQL's object capabilities allow us to build a data structure that is very similar to that used in the modelling applet.

As briefly noted in the previous section, the repository does not currently implement representation-specific subclasses, although this is under investigation. The inclusion of representation-specific subclasses would further reduce the impedance mismatch between the modelling applet and the repository. The class structure in the modelling applet is currently more complex than that in the repository, which requires more work on the part of the modelling applet to decipher the repository contents.

If the repository data were already specialised into subclasses appropriate to a particular representation, the modelling applet's logic could be simplified, resulting in a "thinner" client than at present. This does have the disadvantage, however, of making it more complicated to add a new representation to the repository — we must add the appropriate PostgreSQL subclasses (that is, alter the repository structure) in addition to the Java classes for the representation.

As noted earlier, we also intend to store Swift itself in the repository, which mean that the server not only serves data, but also the application that manipulates that data.

5. Summary and future research

In this paper we have discussed the implementation of a database design environment called *Swift*. The novel features of Swift are:

- Swift implements a viewpoint-oriented approach, in particular the ability to use multiple representations to describe a viewpoint, and to transform between these representations;

- Swift takes advantage of various object-oriented features; and

- Swift itself is stored in an object-relational DBMS.

There are many issues arising from Swift's implementation that will require further research. These issues include:

- A useful feature of many object DBMS's is the ability to store data about multiple versions of an object. We intend to use PostgreSQL's temporal features to implement this capability in Swift. This will allow us to do such things as track the lineage of a particular description as it evolves over time.

- Our current research is limited to using multiple representation within a *single* viewpoint. It is hoped that we can eventually scale the work done with multiple representations up to multiple viewpoints.

- The process of schema generation from multiple "source" descriptions is very interesting. "Traditional" schema generation generally involves only a single source description, so we must determine how multiple descriptions may be applied to generate a schema.

To conclude, we have found that the object-oriented paradigm has provided many advantages that allowed us to rapidly develop a client/server database design environment.

6. References

Cattell, R. G. G. (1991). *Object Data Management*. Addison-Wesley, Reading, Massachusetts.

Darke, P. and G. Shanks (1995). Viewpoint development for requirements definition: Towards a conceptual framework. In *Proceedings of the 6th Australasian Conference on Information Systems (ACIS '95)*, Perth, Australia, pages 277-288.

Easterbrook, S. M. (1991). *Elicitation of requirements from multiple perspectives*. PhD thesis, Imperial College of Science Technology and Medicine, University of London, London.
[http://research.ivv.nasa.gov/~steve/papers/thesis/thesis.ps]

Easterbrook, S. M., A. C. W. Finkelstein, J. Kramer and B. A. Nuseibeh (1994). Co-ordinating distributed ViewPoints: the anatomy of a consistency check. *Journal of Concurrent Engineering: Research and Applications*, **2**(3).

Evergreen Software Tools (1995). *EasyCASE® User's Guide*, Evergreen Software Tools, Inc., Redmond, Washington, version 4.2.

Finkelstein, A. C. W., M. Goedicke, J. Kramer and C. Niskier (1989). ViewPoint oriented software development: Methods and viewpoints in requirements engineering. In *Proceedings of the Second Meteor Workshop on Methods for Formal Specification*, Springer-Verlag.

Informix Software (1996). *Developing DataBlade® modules for INFORMIX®-Universal Server*. White paper.
[http://www.informix.com/informix/corpinfo/zines/whitpprs/databld/21360f0.htm]

Keeler, M. (1996). Database of all trades. *Database Programming & Design*, **9**(11): 34-42.

Klein, H. K. and R. A. Hirschheim (1987). A comparative framework of data modelling paradigms and approaches. *The Computer Journal*, **30**(1): 8-15.

Kotonya, G. and I. Sommerville (1996). Requirements engineering with viewpoints. *Software Engineering Journal*, **11**(1): 5-18.

Leite, J. C. S. P. and P. A. Freeman (1991). Requirements validation through viewpoint resolution. *IEEE Transactions on Software Engineering*, **17**(12): 1253-1269.

McLean, B., J. Medeiros and P. T. Mount (1997). *JavaPostgres95*, version 0.3. [http://www.demon.co.uk/finder/postgres/index.html]

O'Brien, D. (1992). *Deft Editors and Utilities*, Sybase, Inc., Emeryville, California, version 4.2.

Pascoe, R. T. and J. T. Penny (1990). Construction of interfaces for the exchange of geographic data. *International Journal of Geographical Information Systems*, **4**(2): 147-156.

Stanger, N. and R. Pascoe (1997). Environments for viewpoint representations. In *Proceedings of the Fifth European Conference on Information Systems (ECIS '97)*, Cork, Ireland.

Yu, A. and J. Chen (1995). *The POSTGRES95 User Manual*, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.
[http://www.eol.ists.ca/~dunlop/postgres95-manual/]