Exploiting Patterns for Rapidly Delivered, Runtime Extensible Systems

Graham McLeod
University of Cape Town www.commerce.uct.ac.za
Inspired www.inspired.org mcleod@iafrica.com

Abstract

The paper describes a proto-pattern for a system architecture which facilitates very rapid development and extension of an application. The overall pattern has four major components: Overall Architecture; Process Patterns for standard editing, viewing and navigation; Meta Data/Instance patterns for knowledge storage; and Interface patterns for user interaction. The architecture level pattern uses meta-data at run time to customize process patterns implemented in an interpreted language, rendering them executable. The meta-data structure in turn uses patterns which facilitate persistent storage of extensible data structures and relationships. The meta data is used to customize interface patterns which are served for rendering by a client environment (e.g. Web browser). The use of the proto-pattern in the design of a commercial web-based knowledge management product illustrates its potential. Benefits are highlighted and areas for further work indicated.

1 Problem Statement

Business pressures and unclear or rapidly changing requirements require systems that can be developed very rapidly and extended with minimal delay. Conventional development techniques appear to be inadequate in addressing this problem. Object oriented techniques have long held much promise and have provided benefits, but many organizations are still frustrated by slow, expensive development which simply does not meet their business needs. [Taylor, 1995]. The advent of the Internet and e-Commerce has simply exacerbated the position.

Applications are increasingly mission critical, having to operate 24 hours per day, 365 days a year in a global economy that does not pause. There is thus a need for very high reliability and availability in the systems produced.

2 Previous Approaches

Some of the approaches already used to speed development are listed below, with a discussion of their effectiveness.

- *High level languages* - assist by allowing developers to work at a higher level of abstraction. Figures from [Jones, 1997] indicate that language level is a major determinant of developer productivity. Some researchers, e.g. Taylor [1995] have proposed the ideal of a language at a business domain level, where users can specify requirements in their own terms
- Use of *class libraries* has certainly facilitated reuse of code and saved effort in the implementation of commonly required facilities. This has achieved much success in the areas of graphical user interfaces (GUIs), relational database and complex internal data structures (e.g. Collections). The success of Java in the marketplace is largely due to an increasingly rich environment, delivered in the form of Java classes.
- *Components* have been very successful at the developer level, in much the same way as class libraries. There are vast component libraries available for C++, Visual Basic, Delphi, Java and Smalltalk. Again, these primarily address technical aspects, such as GUI, collections, communications and database access
- *Application Generators and CASE Tools* aim to take models and turn them rapidly into reliable executable code. While there have been some seminal successes, the general experience is that these approaches require a great many critical success factors to simultaneously be met, resulting in a high failure rate [McLeod, 1993]
- *Visual Assembly Integrated Development Environments*, such as Visual Basic (Microsoft), Delphi (Borland) and Visual Age (IBM) have greatly facilitated rapid construction of graphical interfaces, and, in the case of the latter, entire applications, by providing visual construction kits where applications can be built by visually combining and connecting parts

We do not discard any of the above approaches: indeed, they are integral to the approach suggested in this paper. We *do* contend that alone they do not provide sufficient acceleration of delivery to satisfy business demands. We believe that the above, augmented by the judicious use of meta data, patterns and innovative system architecture, provides a promising way of addressing the delivery problem.

3 Relevant Concepts

3.1 Meta Data

Meta data is simply data about data. It typically describes data items, abstract data types, the structure of records and objects, and sometimes the relationships between these. Meta data is often employed within the data dictionaries of database and query tools, and in the repositories of CASE tools. It is less frequently found explicitly in application systems, except in the code of table definitions, record layouts and the like. Meta data is usually fairly abstract: it describes things at a generic level. A meta-schema or meta-model will generally be concerned with *types of things*, rather than the things themselves. Typical elements within a meta schema within an object oriented context include:

- Object types/classes
- Relationships between the above, including: inheritance, containment, reference (association) and the multiplicity and constraints of these
- Attributes (names, types, multiplicity, legal ranges or values)
- Behaviour (methods, functions, rules..)

3.2 System Architecture

System architecture refers to the overall structure of the application: its components, layers and interfaces and their respective functions and responsibilities. We have, for many years, used a modified form of the model, view and controller architecture to design scalable and easily adapted applications [McLeod, 2001]. In the proto-pattern described, this architecture is adapted further to cater for the use of process patterns within the business process layer, and run time use of meta-data in the model and view layers.

3.3 Patterns

Patterns are abstracted recurring solutions to similar problems, within a context. They normally express a general structure, approach, algorithm or design which will (with appropriate adaptation for detail) solve a certain problem which recurs in a variety of situations. Patterns are normally expressed as a model, or using a structured pattern language template. See [Appleton, 2001] for a good introduction. The concept can also be relevant for programming level best practices, where it is more frequently called an *idiom*. Patterns can occur at a variety of levels, e.g.:

- *Design patterns* provide guidance on how to use features within a given environment; or how to arrange certain components; or how to solve a particular design problem [Gamma, Helm, Johnson, Vlissides, 1994]
- *Analysis patterns* provide examples of good abstractions of problems, expressed as models. They encapsulate experience of expert analysts in an accessible form [Fowler, 1997]
- *Architecture patterns* provide guidance in arranging the subsystems or layers of a solution, allocation of responsibility to components and/or management of interfaces in an architectural solution. A famous example is the Model/View/Controller (MVC) approach of Trygve Reenskaugh [Reenskaug, Wold and Lehne, 1996]

In this paper we propose an architectural level pattern for systems which provides rapid delivery and adaptability. This pattern, in turn, makes use of other embedded patterns, respectively dealing with extensible data structures, common processing requirements and a variety of user interfaces.

We will express our patterns using a pattern language template synthesized from the work of [Buschmann *et. al*. 1996] and [Mowbray and Malveau, 1997]. This combined template has the following structure:

- *Name* - a unique and descriptive name. To be used as a shorthand in referring to the pattern
- *Abstract* - concise summary of the following information
- *Most applicable scale* - at what scale (e.g. Industry, enterprise, system, module) does the solution best apply?
- *Problem* - what problem does the pattern solve or address? What is the intent of the solution?
- *Context* - what is the extant situation before application of the pattern?

- *Forces* - what are the relevant constraints and issues impacting upon the choice of a solution?
- *Solution* - concise description of the pattern solving the problem, often expressed as a model
- *Examples / Known Uses* - sample applications illustrating suitability of pattern and how it is adapted. References to where the pattern is known to be used, preferably in production systems
- *Resulting Context / Benefits / Caveats* - the situation after the application of the pattern and any appropriate cautions
- *Rationale* - justifies the approach pursued in the pattern or the design decisions taken
- *Related Patterns* - identifies other patterns used with the one documented

4 Proto-Pattern

4.1 Architecture Level Pattern

4.1.1 Name

PAMELA (Pattern and Meta data Leveraging Architecture)

4.1.2 Abstract

The pattern describes a system architecture which takes advantage of three layer architecture and runtime customization of process patterns using online meta data to deliver extremely rapid application development and extensibility.

4.1.3 Most Applicable Scale

Application system architecture

4.1.4 Problem

Traditionally developed applications suffer from poor delivery rate (time to deployment from specification) and are difficult to extend or adapt to changing requirements. Traditional development methods often assume that requirements can be accurately defined before development is finalized.

Current business climates demands very rapid development of applications, easy and quick extension and an ability to have applications up and running before all requirements are fully articulated. In addition, these applications must be robust and offer high uptime and reliability. It is desirable that applications are database based, multi-user, transactional, secure, web enabled and scalable.

4.1.5 Context

The context is a business environment requiring systems support where applications must be brought into production very rapidly. Examples would include some financial environments, medical applications and knowledge based applications. Some business requirements will be known at the outset and can be articulated in the form of models. Others will only become apparent through use of the application and application of information technology, structured information and knowledge to the problem area. The application may require the storage of both structured data (traditional style records), as well as documents, multimedia and references to external knowledge assets. It will be desirable that the solution be transactional, secure and support multiple users, preferably through commodity interface technology, such as an Internet browser.

4.1.6 Forces

The solution must balance the rapid delivery with the needs for robustness, integrity of data management, high availability, multi-user support and scalability. The latter implies a multi-tier architecture.

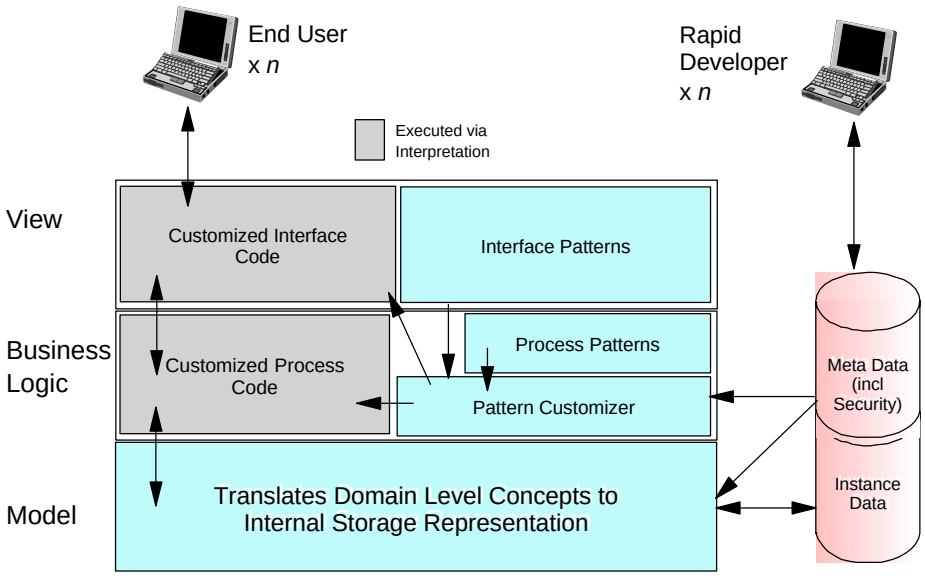

Figure 1 - PAMELA system architecture pattern

4.1.7 Solution

The PAMELA pattern makes use of the following devices to achieve very rapid development and adaptability of resulting application:

- Multi-layer architecture with separation of domain knowledge, business process and view management. This is based on the traditional model-view-controller architecture, as extended by the author within the *Inspired* method [McLeod, 1998]
- Meta-data is stored persistently in a repository. Structures do not reflect application or domain constructs, but rather highly abstracted general structures capable of holding models of the application domain concepts and structures - effectively a meta model. Patterns similar to the observations and measurements patterns of [Fowler, 1997], but extended to cater for dynamic types and relationships, are used to hold the meta model
- Processing is achieved by the runtime customization of process patterns and user interface patterns which contain necessary logic to create a variety of interfaces customized to a variety of tasks in viewing and maintaining application and meta-data. These allow for all common functions: definition of types, capture of data, querying (in a variety of modes, including graphical), editing, navigation, export and import.
- Customized patterns are executed via interpretation in both the business process layer (process patterns) and the view layer (interface patterns)
- Security is ensured by carrying a security model in the meta-data and using this to filter the generated user interfaces to ensure that no data, features or controls which should not be accessible to a given user are served
- Processes are transactional and ensure that logical transactions are either committed in their entirety or rolled back in the event of a failure
- Application data is stored in an atomic form that does not require any new database structures, allowing the physical schema to remain unchanged while new types are defined in the user application

4.1.8 Examples / Known Uses

The application of the pattern in the *Inspired* Archi knowledge management tool will be discussed later in the paper. At the moment, this is the only production use of the pattern as defined by the author: (It is possible others have defined similar or equivalent structures). This is one of the main reasons this is currently suggested as a proto-pattern until further implementations prove wider applicability.

4.1.9 Resulting Context/ Benefits /Caveats

☺ Very rapid development is possible. Essentially, a developer will create meta data describing the domain objects and relationships of the problem space at a level roughly equivalent to a UML Static Structure model [OMG, 1999]. Once this is done, the pattern customizer can render usable processes and interfaces and the model layer can map domain concepts to storage and back again. Once the model is created, the application is essentially live. Security can then be customized as desired

☺ Applications are very easily extended by extending the model. This can be achieved without disruption of the existing application, since model extensions occur within a prescribed set of rules

☺ The application has a very small footprint, since there are a limited number of process patterns and interface patterns which are reused for many user domain types

☺ The database is very stable, since only types permitted in the modeling are required and these are predefined to the storage engine. Only new instances of existing structures are created

☺ Reliability in the processing system is achieved since no new logic is added, merely customizations of existing debugged processes

☺ The system can be highly available, since the meta-data can be edited at runtime, and becomes active once committed. The application does not need to be replaced, so there is no downtime, unless fundamentally new process pattern customization is added

☺ Deployment can be eliminated if the developer is equipped to update meta data remotely

😐 Domain rather than technical skills are required of application developers, but higher than normal commercial skills are required from "tool" developers of the meta data maintenance, pattern customizer and model layer

☹ Performance will be slower than compiled systems without run-time meta data

4.1.10 Rationale

The approach chosen is based on long experience with the need for flexible systems in commercial settings. Many systems have been designed by the author and colleagues over many years (more than two decades) with elements of the solution proposed. The formalization of pattern concepts and the maturation of interpretive tools and interface environments provided the necessary context and the need to develop an advanced design for a knowledge management product capable of end user extension at run time provided the necessary impetus to bring the ideas into focus and formulate them as a pattern. Reasons for major design choices are given below:

- The multi-layer architecture facilitates ease of maintenance, separation of concerns and scalability of the implementation. It also provides the possibility of multiple user interface styles operating with one business logic and domain knowledge layer
- Dynamic and runtime accessible meta data was necessary to achieve the level of rapid development and extensibility required. In effect, we are borrowing from techniques used in query and prototyping tools which have proven effective in meeting demanding ad hoc requirements for many years
- Process patterns allowed us to capture common processing requirements in a reusable form which hides much of the complexity from users. They are pre-tested and incorporate transaction management, promoting system integrity and reliability
- Interface patterns allow us to create well behaved interfaces for commonly performed tasks, which can be customized for the particulars of the domain structures using the meta data mentioned previously. Like the process patterns, these are pre-tested and support proven interaction models
- The customizer effectively performs the job of a pattern adapter or instantiator in software. While this is obviously not as powerful as the ministrations of a competent human analyst / designer, it is extremely rapid, being performed at run time within a normal response time
- Generating the customized patterns for process and interface in an interpretive language allows their immediate execution
- Keeping security at the meta data level allows the customizer to generate interfaces tailored to legal activities of a given user
- Storing information in the database in an atomic form provides a stable database structure, requiring no schema extension to accommodate newly defined user /domain types

4.1.11 Related Patterns (described in following three sections)

- The object pattern for storage of types, attributes, relationships and constraints

- The various process patterns for customization. These will be related to the following generic activities:
  - Creating and amending types and relationships
  - Adding, amending and deleting items
  - Query and printing
  - Navigation through related items
- User interface patterns.

4.2 Meta Data Structure Pattern

4.2.1 Name

Domain Knowledge Structure Pattern.

4.2.2 Abstract

Provides a way of dynamically storing meta data structures and instance data structures within a stable schema data store. Meta data allows the modeling of domain concepts and the management of instances of the corresponding data.

4.2.3 Most applicable scale

System Architecture for Storage.

4.2.4 Problem

Conventional databases require schema definitions to be stable and defined before instances can be created. Altering the design once instances are created will often involve: editing, compiling, changing schemas, taking database down, offloading data, changing schema and reloading data. These activities inhibit schema evolution and result in unacceptable or expensive downtime. We sought a structure where the schema as perceived by the database engine was static. This was achieved by abstracting the structures commonly required by individual types to a generic level.

4.2.5 Context

Applications requiring rapid development or extension without redefinition of the data storage schema. Volume of data and transactions should not require very high performance.

4.2.6 Forces

It is important to provide flexibility, but to maintain integrity management and reliability. Performance was considered to be secondary, but should remain acceptable.

4.2.7 Solution

We designed a structure at a level of abstraction similar to that normally found in a data dictionary or repository for a query tool or CASE tool. We included the following classes in our design:

- Types - which are analogous to classes
- Relationships - which are named relationships between types. These can be semantically rich and represent domain concepts
- Attributes - which are used to define the structure of instances of types, called items in our model
- Instance’s types (items) and values of attributes for an item were stored independently
- Domains - which group types and relationship types into sets for management purposes. Domains are not disjoint

The pattern can be extended to add constraints on relationships and history or versioning for attribute values.

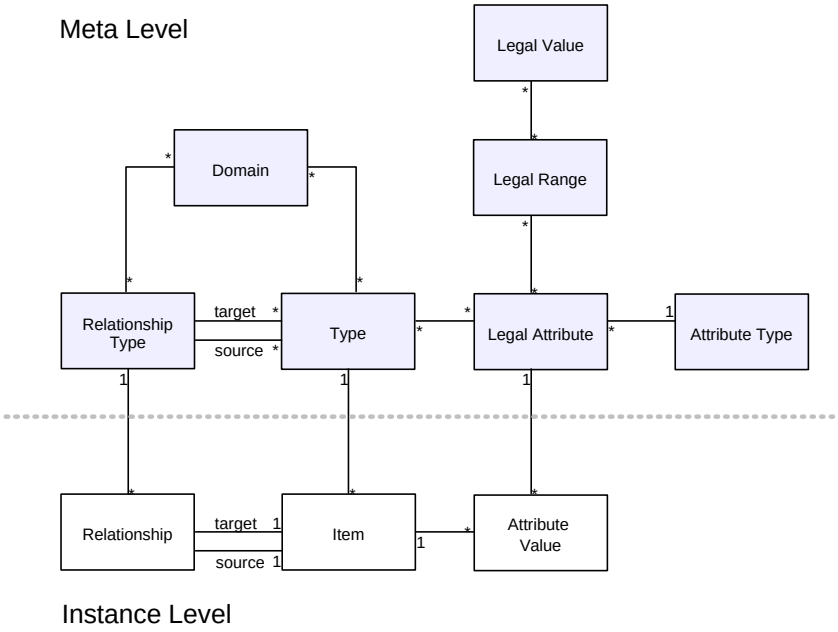

Figure 2 - Domain Meta and Instance storage pattern

4.2.8 Examples / Known Uses

We use these structures within the PAMELA pattern described above. They are also implemented (in modified form to accommodate production relational database technology) within the production Archi tool.

4.2.9 Resulting Context / Benefits / Caveats

☺ The resulting data model can accommodate runtime definition of types, relationships, attributes, typing of attributes, ordering of attributes, and mapping of domain structures defined by the meta data into the storage structures for instance data and vice versa. New types, attributes and relationships can be added and are immediately available for use without disturbing existing instances. Existing instances of an extended type will acquire the new attributes with a *nil* value

☺ The data store remains relatively compact, since only actual values of attributes and instances are stored

☹ Performance of the resulting store can be slower than traditional approaches with a fixed schema

4.2.10 Rationale

We have obviously traded performance for flexibility. We have found runtime performance for our implementation to be more than satisfactory for small to medium volumes. Very high volumes of instances and users have not been tested. For our target class of applications, we feel that flexibility is paramount. Performance issues are likely to be overcome in all but the most demanding applications by use of powerful hardware and good database technology at the engine level. We believe that even very high volume applications could be supported in this manner if implementations make use of an efficient object database management system.

4.2.11 Related Patterns

We use this pattern as part of the PAMELA pattern described earlier.

4.3 Example Process Pattern

We will only describe an example process pattern, since many are employed within the PAMELA context and space will not permit explication of all here.

4.3.1 Name

Item Edit.

4.3.2 Abstract
Provides the basic logic for editing an item.

4.3.3 Most applicable scale
System process - controller script.

4.3.4 Problem
Provide generic editing logic for an item.

4.3.5 Context
Functions within the context of the meta data for types, relationships and attributes. Must respect security and validation criteria. Historically, programmers would code this functionality for each new type/ record/ object, sometimes by customizing a template.

4.3.6 Forces
Need to provide a good user interface, respect security and manage integrity of data. In our case, we also want to provide for customization to support a variety of types.

4.3.7 Solution
A pattern is developed in the target interpreted language. This provides a template, which is later parsed to insert/replace/modify where necessary to accommodate the attributes of the specific type to be edited. The pattern contains logic for communicating with the interface layer, retrieval of domain data from the model layer, and managing transactions. It will ensure integrity by validating keys and entered data against attribute types and valid values held in the meta data. The variables from the meta data are used to customize the process pattern, rendering it as a specific instance, customized to the selected type. The customized pattern is executed interpretively. The template for edit might appear as follows:
(we have used pseudo code to remain implementation neutral)

```
EDIT
Obtain list of items of type <Type>
Pass to user interface <Type:Edit> for list display
Receive selected item key from interface <Type:Edit>
Retrieve selected item attribute values from Model passing <Type> and <Key>
Retrieve selected item type <Type> valid values from Meta Data
Begin transaction
Lock <item> for potential update
Pass retrieved  value/type pairs to interface <Type:Edit>
Pass valid values collection to interface <Type:Edit>
Receive modified values from interface <Type:Edit>
Validate modified items using valid values
        Return error to interface <Type:Edit> if validation fails
Update <item> via Model layer
End transaction
```

4.3.8 Examples / Known Uses
Used within the PAMELA architecture pattern previously described. Used within the Archi knowledge management tool.

4.3.9 Resulting Context / Benefits / Caveats
- ☺ Process patterns provide tested logic to generically implement processing for new types
- ☺ Customization process allows rapid delivery of high functionality for new types added without additional effort
- ☺ Application is small, since code is reused for many types
- ☹ Errors will affect all types equally - maintenance of the patterns must be very carefully performed when necessary
- ☹ Only anticipated processing can be performed using patterns already in the “repertoire” of the tool. This has subsequently been overcome by supporting calculated fields and event management (not discussed in this paper)

4.3.10 Rationale
Process patterns capture the common logic required to perform generically required functions common to all types. These include the basic functions of adding objects, deleting objects, editing values, querying data, navigation through relationships etc.

Runtime customization provides rapid system extension capability. Reuse is promoted and system stability improved through the use of well tested logic.

4.3.11 Related Patterns

This patterns is used as part of the PAMELA architecture pattern described previously. It is implemented in a modified form in the production Archi knowledge management tool.

4.4 Interface Patterns

Space will not permit a full discussion of these in this paper. Some 12 unique interface patterns have been developed to support a rich range of interactions. These include: basic item editing and linking, with viewing and navigation to related items; cross referencing; tree and graphical viewing of relationships; temporal (calendar) views for time related items; user defined spatial models; customized menus; query and editing of master/detail related items concurrently; rich content viewing; worksheets.

Patterns are implemented as templates for a chosen client environment (e.g. HTML, Java Script, XML) into which instance data is inserted. Appropriate forms of instance data are derived from database values parsed via a method of the appropriate data type.

5 Example: Archi from Inspired

Many of the principles discussed in the foregoing have been applied in a production web based knowledge management tool from *Inspired*, called Archi [Inspired, 2001]. This was initially developed to provide support in managing enterprise architectures, methods and programme management activities within corporate clients. It has become a more generic tool suited to managing knowledge for collaborating groups of professionals or consultants. Design goals included the following:

- Support for multiple concurrent users in a web based environment
- Persistent, shared storage of both meta and instance data in a commodity relational database engine on a server
- Minimal requirements for client - specifically, a current level (Version 4) popular browser with Java Script, but no plug ins, Java Applets or ActiveX Controls
- Very rapid modeling to accommodate a specific domain and to extend capabilities in that domain with experience
- High availability and reliability
- Support for structured data (held internally), rich content (held externally) and document and knowledge assets (held on server and managed via versions and linked to via hyperlinks, respectively)

The architecture, tool and design choices were as follows:

- The tool resides on a server, between a standard web server and a relational database engine
- It uses a web browser for the user interface, running as a remote client. This interface is used for both end users and meta data development users
- Server side processes were implemented in Smalltalk, which was also used to execute the customized process patterns. The model layer translating between domain objects and storage was also implemented in Smalltalk. Smalltalk was chosen because of prior experience, high language level [Jones, 1997], rich class library and runtime extensibility, supporting execution of customized code
- IBM Visual Age was used, as a productive development environment with a rich class and component library, visual construction and valuable frameworks for object persistence (Object Extender) and web session management (Web Connect)

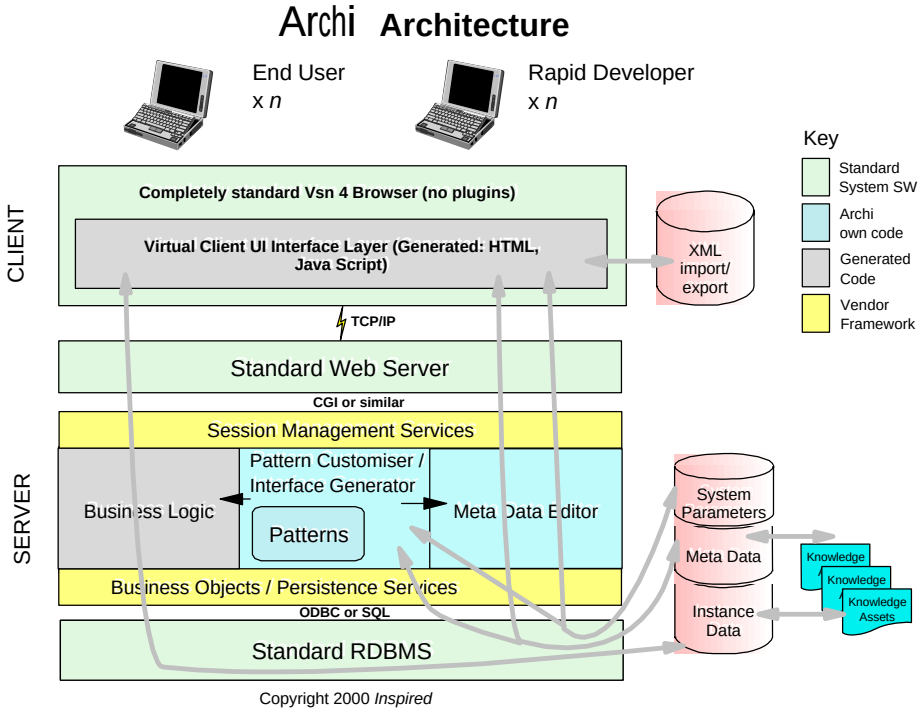

Figure 3 - Archi as example of adaptable architecture

- Client interface customized patterns were rendered in standard HTML and Java Script as the most widely supported vehicle for current generation browsers. We may, in future, use XML and XSL when support for these has improved. Unlike the picture shown in the PAMELA pattern, Archi holds the interface template code in the process layer, to have this available on the server and renders the interface code to the client. In this way, the client need never have connected to the server before or have anything installed to successfully interact with the server and render a usable interface
- Meta data editing was also accomplished via the same interface strategy, and meta data storage was achieved through the model layer, mainly to take advantage of the sophisticated object to relational framework and transaction management provided by Object Extender
- Web Connect was used between the business logic and interface layers to take care of session management and preserving session data between client invocations of server side logic. This is a major help in the HTTP protocol, which is inherently stateless
- System parameters are also held in the database. These allow customization of look and feel, hold details of server paths, and allow tailoring of many features of the tool without programming or taking the system down

6 Conclusion

6.1 Benefits

The design goals of the tool were met. The tool has proven tailorable to a variety of contexts, including Enterprise Architecture, Customer Relationship Management data modeling, Programme Management and Methods Management. A software house has used the tool to bring up a production system for fault reporting, developer allocation, release scheduling and tracking within less than a day.

Rapid development of meta data to address a new domain was recently demonstrated by deploying the tool to manage data related to certain University of Cape Town postgraduate courses, including: courses, students, schedules, deliverables, readings, reference resources, student submissions, subject index and various other aspects. Modeling the new domain and loading sufficient test data to verify the model's viability was accomplished in under two days. The resultant model included about 17 domain "classes" and some 70 relationships. The model has been extended as course administration progressed and new

requirements can usually be implemented in a matter of minutes. To summarize, the approach has provided:

- Extremely rapid development, runtime extension
- No downtime for alterations to the system
- Capability for remote development, adding or extending meta data from a browser client
- Development concurrent with production use by end users
- Support for multiple developers working concurrently

The application is relatively small (some 20Mb plus about 4Mb of virtual machine and support code). Reliability has been excellent.

6.2 Further Work

The PAMELA pattern should be developed further to incorporate performance optimizations, allowing the approach to be used in demanding performance and volume applications.

We would like to see the pattern applied using an object database for persistence.

The pattern can be applied to other technical environments e.g. Java2 Enterprise Edition, where the client side interface can be rendered as Java beans and applets, and the server side processing accomplished as Enterprise Java Beans. XML and XSL could also replace the current HTML for client formatting.

References

Appleton, Brad, 2001, Patterns and Software: Essential Concepts and Terminology, www.enteract.com/~bradapp/docs/patterns-intro.html, accessed Feb 2001

Beck, Kent, 1997, Smalltalk Best Practice Patterns, Prentice Hall, 1997

Buschmann, F; Meunier, R; Rohnert, H; Sommerlad, P and Stal, M, 1996, Pattern-Oriented Software Architecture - A System of Patterns. Wiley and Sons, 1996

Coplien, James, 1998, Software Design Patterns: Common Questions and Answers, ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps, accessed Jan 2001

Jones, Capers, 1997, Programming Languages Table, Software Productivity Research database, www.spr.com/library/0langtbl.htm, accessed Jan 2001

Fowler, Martin, 1997, Analysis Patterns - Reusable Object Models, Addison Wesley, 1997

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, 1994, Design Patterns, Addison Wesley, 1994

Inspired, 2001, Archi by Inspired: Knowledge Management for Professionals and Consultants, Inspired, www.inspired.org, accessed Nov 2001

Lea, Doug, 1994, Christopher Alexander: An Introduction for Object-Oriented Designers in Software Engineering Notes, ACM SIGSOFT, vol 19, no 1, p 39, Jan 1994

McLeod, Graham, 1993, An empirical investigation into the use of CASE technology in South Africa, University of Cape Town working paper

McLeod, Graham, 1998, Linking Business Object Analysis to a Model View Controller Based Design Architecture, Proceedings of the Third CAiSE/IFIP 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design EMMSAD'98, Pisa, Italy, June 8-9, 1998

McLeod, Graham, 2001, Beyond UML: Advanced Systems Delivery with Objects, Components, Patterns and Middleware, Inspired Press, 2001

Mowbray, Thomas & Malveau, Raphael, 1997, CORBA Design Patterns, John Wiley, 1997

Object Management Group (OMG), 1999, OMG Unified Modeling Language Specification Version 1.3, www.omg.org, accessed 1999

Pree, Wolfgang, 1995, Design Patterns for Object-Oriented Software Development, Addison Wesley, 1995

Reenskaug, T;Wold, P; and Lehne, O.A., 1996, Working with objects: the OOram software engineering method, Manning Publications Company, 1996

Taylor, David, 1995, Business Engineering with Object Technology, John Wiley, 1995