iMatix Corporation sprl

13/15 rue des Ateliers 1080 Bruxelles, Belgium
Tel.+32-75-235-984, fax.+32-2-414-8404
http://www.imatix.com
info@imatix.com

Technical White Paper

The iMatix Application Framework (iAF)

The iMatix Application Framework (iAF) is a design and construction process for three-tier web-based or GUI applications. The iAF Repository stores the design model of an application and drives a template-based code generation and documentation process. The iAF Repository has three layers: presentation, business objects, and database. These layers are described using XML framework languages: the Presentation Framework Language, the Object Framework Language, and the Database Framework Language.

This document describes the overall principles and operation of iAF.

Copyright

Version Information

Written: 15 November 1999
Revised: 31 January 2000

Disclaimer

The information contained in this document is distributed on an “as-is” basis without any warranty either expressed or implied. The customer is responsible for the use of this information and/or implementation of any techniques mentioned. iMatix Corporation has reviewed the information for accuracy, but there is no guarantee that a customer using the these techniques and/or information will obtain the same or similar results in its own operating environment.

It is possible that this material may contain references to, or information about, iMatix Corporation products or services that have not been announced. Such references or information must not be construed to mean that iMatix intends to announce such products or services.

iMatix Corporation retains the title to the copyright in this paper, as well as title to the copyright in all underlying works. iMatix Corporation retains the right to make derivative works and to republish and distribute this paper to whoever it chooses to.

iAF Principles and Operation

Designing a Modern Application

A software program basically gets input data, does some processing, and produces some output data. In the same way, a business application basically consists of code to handle the user interface ('presentation logic'), code to do some work ('business logic'), and a database to store information.

Monolithic Applications

A *monolithic application* combines the presentation logic and business logic in one package. The programmer will typically read from the database and write to the screen in one breath.

Monolithic applications are exemplified by the full range of applications from large-scale mainframe applications to single-user Windows programs.

This approach to software design is reasonably simple and can produce fast applications. It has one principal drawback: when the technology used for the presentation layer changes, the entire application must be revised, rewritten, or discarded.

In the Internet age, where presentation technologies change very rapidly, monolithic designs cannot cope. It is possible to add 'web front ends' to existing monolithic applications, but this is generally slow and cumbersome. The best design for modern applications is the 3-tier design, in which presentation logic is separated from business logic.

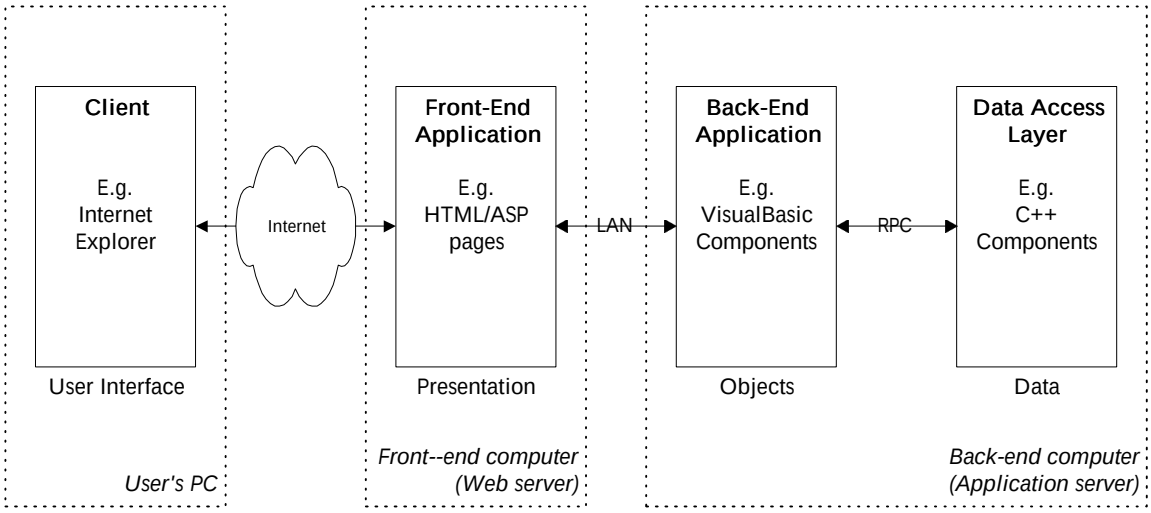
The 3-Tier Application Architecture

The 3 tiers in our design are:

- The user-interface, typically consisting of a Windows GUI or Internet browser.
- The presentation layer, typically consisting of a web application.
- The business logic layer with database access.

In some 3-tier designs, the database access is itself separated from the business logic. In others, the presentation logic and user-interface are combined into one layer. The critical element in all of these designs is the separation of presentation logic from business logic.

This diagram shows a typical 3-tier web application:



Design Guidelines

1. The user-interface and presentation logic is separated from the business logic. The way that data and actions are shown is not defined by the business logic. This is entirely the responsibility of the presentation and user-interface layers. The business logic is responsible for deciding what data and actions are shown, and how actions are handled.
2. The business logic is placed as close to the database as possible. The reason for this is performance: when data is fetched across a network, searches and calculations that read large amounts of data are slow.
3. Information passed between layers is formalised into messages that obey a formal protocol. For instance, between the web user-interface and presentation logic layers, this protocol is HTML/HTTP. Between the presentation and business logic layers, we use an XML language called OAL (Object Access Language).

A Code Generation Approach

Why and How?

iAF is based on cheap, flexible code generation. This technology is unusual. The iMatix GSLgen code generator is the only general-purpose code generator that we know of. The original paper proposing a general-purpose code generator was written in 1994 by Pieter Hintjens, based on his experience using and writing CASE tool code generators. It took about five years to develop GSLgen from the original design paper to a working tool, through a long series of projects and toolkits.

GSLgen is adequately described elsewhere, but it is useful to re-examine the reasons why flexible code generation gives us impressive leverage over the design process.

- We can raise our ambitions from the tedious business of managing large amounts of hand-built code to a much more valuable approach, namely the abstraction of the problem in hand as a set of templates and meta-data definitions.
- A more abstracted approach means building formal models of the application's work. When these can be turned into code automatically, we eliminate the error-prone business of manually converting an analysis into code.
- The abstraction of the application as a set of templates encourages us to think of the application as a set of standardised components, rather than many special cases. This is always beneficial. A code-generation approach reinforces this.
- Since our effort can be focussed on a much smaller amount of code, the results are of significantly higher quality.

GSLgen is so sophisticated that we can handle any situation where the problem can be abstracted as templates working on meta-data. We are not limited to generating a predefined set of source code models.

We do not suggest that every single program can be generated. Only about 75-90% of typical business applications. The principal alternatives to code-generation are:

- Writing everything by hand. We can see no arguments for this except that some cases are too simple to benefit from code-generation.
- Interpreting a meta-data model dynamically. This can provide the flexibility of an abstracted approach, but it imposes a runtime overhead that is unnecessary in most cases. When meta-data can be converted into source code, there is no reason to consume resources at run-time to achieve the same effect.

Design Impact of a Code Generation Approach

The main impact of a code generation approach on application design is that the design exists principally as a set of meta-data definitions, rather than a body of documented source code. We call these meta-data definitions a 'repository'. The iAF Repository is a standardised design that can describe many kinds of application.

Deployment Impact of a Code Generation Approach

The iAF code generation approach can be quickly customised to suit changing circumstances. Deployment issues that are highly significant in hand-built applications become trivial when the code is generated. For example, the choice of database product ("SQLServer 6.5 or SQLServer 7.0?") is of little consequence when all accesses to the database are generated. By replacing or modifying one set of templates, code can be tuned for new releases of a database product, or even different database products.

Comparing the iAF Repository with other Repositories

Repositories are commonly-used in formal design and modelling tools, such as those that use UML (the Universal Modelling Language). The principal distinguishing feature of the iAF Repository is that it is directed fundamentally towards the generation of a complete and working application plus documentation, while UML-style repositories are directed principally at capturing the application analysis and design.

It may be possible to exploit UML repositories with GSLgen-style code generation, and it is certainly possible to capture analysis and design in the iAF Repository. However, the iAF Repository is not supported by a graphical user interface and sophisticated graphing tools, and UML repositories are not supported by flexible code-generation.

Comparing iAF Code Generation with Other Approaches

Tools like Rational Rose excel at capturing requirements, but deliver mediocre and inflexible code generation targeted towards a limited set of products (e.g. wrappers for objects). GSLgen code generation can target any programming language, and any style or shape of program, with only modest work.

This is quickly seen by the effort needed to generate IDL (interface definition language) code from a repository in some arbitrary language. Imagine you wish to expose a set of services to programs written in Java. This means generating Java functions to read and write messages. Now, imagine the same exercise in COBOL or Perl.

The cost using GSLgen is typically measured in hours or days. With a non-template based code generator, the effort is measured in weeks.

iAF Design Issues

Efficiency

Efficiency is partially the result of a simple design: the user can work faster and with less effort. In terms of the end-user, efficiency means:

- Tasks can be completed quickly and with a minimum of work.
- The user-interface is flexible and obvious.
- The software is responsive and fast.

An Efficient User Interface

To make the user interface simple and obvious, we apply three rules:

1. Check input as early as possible - tell the user when something is wrong or missing right away.
2. Don't restrict the user artificially -the user should be able to do anything that's meaningful and legal at any point.
3. Don't try to show too much at once - keep the user's focus on what's important.

Efficient Software

Ensuring good response time means building faster software, rather than using faster computer systems. Software efficiency is largely determined by database access. The fewer accesses are done, the faster an application will run. To minimise database accesses, we define two rules:

1. Never fetch data that is not going to be used.
2. Never fetch data across a network.

The first rule is broken by programs that show long lists of data that the user will never read, or which fetch entire trees of objects, only to show a small part of this.

The second rule is more subtle. It says: 'when you have an application and a database, put the business logic on the same system as the database'. In many applications the business of searching for data is the heaviest work. A long search may return only a small amount of information. Shifting data over a network is several orders of magnitude slower than processing it locally[1]. A search that fetches data across the network will slow-down

[1] In some cases, when a fast local-area network is used, this rule no longer applies, and splitting an application server into two systems – a database server and an application server – connected by a high-speed network can actually improve performance.

the application. Searching should be done by code that is as close as possible to the database.

The User Interface

Defining the User Interface

Strictly-speaking, the user interface presents a set of data objects and other information to the user, and accepts data objects and actions back from the user.

The principal role of the user interface is to provide the user with a mental model of the application, and allow the user to navigate and use this model. A good user interface reinforces this model so that the user works confidently and intelligently.

Building the User Interface Economically

Capturing data (called 'data entry') is critical. The problem appears simple at first sight: show the user some data, and allow the user to modify this. In practice there are many delicate issues at hand. These must be well solved. This often takes time and feedback from the users, leading to a dilemma for a traditional approach to software development: the more that the application is finalised and usable, the more it actually needs modification to handle feedback about the user interface. The user interface can often become an extremely expensive part of the development process.

Any approach that tries to rigorously pre-define the interface always runs up against unforeseen requirements. Our experience is that an iterative approach built on top of strong fundamentals is more valuable.

Our specifications for the user interface are the set of programs that capture data and actions from the user, and show the user data. These programs are very rich in respect to the ways they can show data, but are very limited in terms of business intelligence. The programs tend to follow standardised components, and are thus ideal donors for a code-generation approach.

Business Objects

Objects vs. Tables

Traditional relational databases define data in terms of 'tables'. Tables can have relationships to other tables, and the user can access the data in a 'useful' manner by exploiting these relationships.

An object-oriented design defines the application in terms of 'objects', and the work done with or by each object. It is only at a late stage that objects are mapped to database tables in some automatic or manual manner.

We believe that:

- the use of objects to map data and business logic into reusable components is necessary and valuable;
- the database tables should only be accessible through such objects;
- such an object-oriented approach should be used in moderation. Overuse of any abstraction technique leads to unnecessary complexity. The main issue in software engineering is the management and reduction of complexity and one must be very wary of re-introducing it by accident.

Our attitude to objects is based on experience with different types of object-oriented design, and analysis of the software industry's experience in this area. Full object-oriented (OO) projects (i.e. using classes, inheritance, and an OO analysis and design method) tend to be expensive, slow, and unstable. Pure OO languages like SmallTalk, Java, and C++ tend to be used only in user-interface design, where OO principles work best. The most successful languages, like Visual Basic, use simple objects, *components*.

iAF uses the term 'object' exclusively to refer to a collection of tables and business logic in one component. The database and user-interface layers are not defined in terms of objects, nor are iAF applications necessarily constructed using OO techniques or OO languages.

Working With Objects

The basic things we can do to an object (or more precisely, an instance of an object) are:

- Create a new object.
- Search for a set of objects that meet some criteria.
- Fetch one object to display or work with.
- Modify an object's properties.
- Delete an object.

In keeping with object-oriented (OO) terminology, we call these methods.

The create, fetch, modify, and delete methods are reasonably easy to define and implement. They correspond to the SQL instructions 'INSERT', 'SELECT', 'UPDATE', and 'DELETE'. For simple objects, one such action matches one SQL command. For constructed objects, it matches several SQL commands, on different tables.

The search method is a different problem. Searching is a fundamental need, and ranges from the trivial ('get me the next ten objects, following the primary ID key field') to the hairy ('get me all articles that match these criteria and which I have not read yet').

Object Searches

A search operation requires some input (the search conditions, and arguments for the search) and returns some output (the set of objects that match). We need some way to formalise and describe these inputs and outputs.

Using SQL For Specifying Searches

A widely-used technique is to use SQL, a language specifically defined for accessing databases. SQL may seem ideal as a way to describe searches, but in fact its use creates many problems:

1. SQL is tied directly to the real database tables. Any program that makes SQL calls or uses the results must know the real database table structure. This breaks the notion of composed objects.
2. Data may be stored and indexed using techniques other than the native database. For instance, some searches can be done much faster if special indexes are constructed before-hand. Such indexes may not be accessible through SQL, even if they are stored in the database.
3. SQL can create serious performance issues if it is not carefully controlled. This is because SQL theoretically hides the notion of table 'indices'. However, any search that does not use indexes correctly will scan the entire database table.

To summarise: passing SQL commands directly to the object handling layer causes havoc with performance, security, and the 3-tier model.

An Abstracted Interface for Requesting Searches

iAF uses an abstracted search interface that works as follows:

- Each search type that an object supports has a name and a set of criteria.
- We ask the data-access program for an object to conduct a specific search, and we provide the criteria for the search. For example, 'find all donors with budget available for this project…'.
- We can also limit the search: 'If you find more than 20 donors, stop'.
- The object access layer returns us the results: 'here are the donors I found'.
- The returned list of objects is sorted, according to some criteria that are predefined. In general, the sort order is closely tied to the search technique and indexes, so is not arbitrary.

What is "Workflow"?

When we look at the criteria for a 'task-oriented' user interface, we see that the notion of 'workflow' plays a central role. Data objects have a life-cycle, in which different things can happen at different moments (and be done by different people).

Process vs. Life-Cycle

Workflow can be considered as a 'process', but we prefer to consider it as the life-cycle of one particular object (instance). This is useful because it ties the workflow to the application database in a concrete manner. It also lets us define the workflow

completely, with respect to that object. A more general 'process' that involves many objects may not be so cleanly defined.

To describe the workflow, we define the set of 'states' in which the object can exist. If we consider a project definition, these states might be: project initiated, project validated, project rejected, etc. Every object is always in a well-defined state.

In each state, we allow (or require) one of a set of actions from the user. For instance, validating, modifying, or rejecting a project that has been initiated. By clearing defining all the actions that are allowed, we implicitly exclude 'illegal' actions, such as modifying a project that has already been validated.

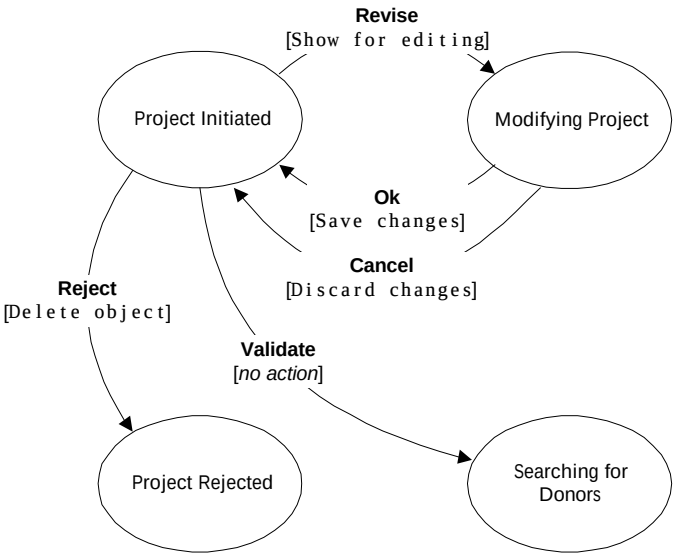
Finally, each user action moves the object to a new state (though sometimes the object just remains in the same state). During this move, some internal actions take place – this is the actual work that the application does to work with the object.

Defining Life-Cycle as a Finite State Machine

This description is formally known as a 'finite state machine', or FSM. There are other names for it, and many variants of it. The principal feature of a FSM is that it is executable. A FSM can be considered equivalent to a chunk of application source code. Concretely: an FSM can be mechanically converted into application source code. This is not at all the case for some other techniques used to describe workflow, like flow charts.

The main difficulty for most people in grasping the finesses of FSMs is the distinction between state and action, and between the user actions and the application's actions. To make this clearer, we call user actions 'methods'.

Let's look at an example FSM fragment that describes how the user validates a project request:



Here the methods are Revise, Ok, Cancel, Validate, and Reject. The states are circled, and the actions are indicated [like this]. Note the difference between the Reject method, the [delete object] action, and the 'Project Rejected' state.

To summarise the differences between defining the workflow as a process and as a FSM:

- A process description says 'do this, then do that…', while an FSM description says 'when this happens, do this. What that happens, do that'. The first approach is procedural, while the second is event-driven.
- Both these descriptions can be turned into application code, but an FSM is a clearer, more explicit, and more rigorous way of defining the exact state of an object at any time, and from that, the set of methods that are permitted at any time.

A process-oriented design is generally built as a program, by hand. FSMs can be turned into code mechanically very easily, after one has understood how this works.

Implementing FSMs as Code

At iMatix Corporation we have over 15 years' experience with FSM tools. The most sophisticated of these tools is Libero, an iMatix Open Source project that started in 1992 and now generates code in seventeen different programming languages. This tool can generate anything from subroutines to scripts to entire programs. The GSLgen code generator can replace Libero in certain types of FSM, and iAF uses GSLgen for the workflow FSMs.

To implement a workflow FSM, we start by implementing the workflow as an XML data file. We then use this to generate a subroutine that implements the workflow. When we generate code for workflows this way, we find that:

- It is much easier to define new workflows, and modify and tune workflows;
- There is less risk of errors;
- Much more ambitious implementations are possible, since a very complex workflow can be defined and understood without requiring specific programming.
- It becomes possible to provide a large set of workflows, even customised workflows.

The essential FSM application-program interface (API) is this: given a state S, and method M, please tell me the next state N and the action A. In some implementations, the FSM engine can keep track of the state, so that the API is simply: given method M, please tell me action A.

Using Life-Cycle FSMs in Applications

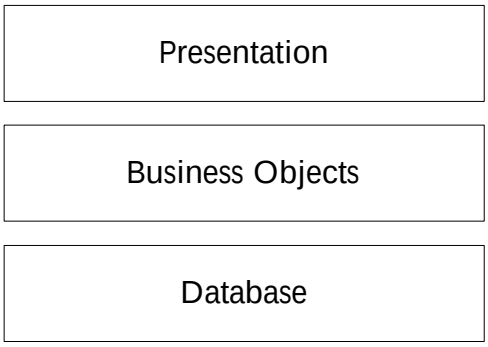
One useful consequence of being able to easily generate application code that implements any particular FSM is that the cost of defining, adding, and modifying these is very low.

This can be contrasted to the cost of changing a process which is implemented manually in a chain of programs. Which brings us to the conclusion: FSMs are a cheap and robust way to describe workflow, so long as we accept the restriction that workflow applies to one principal object.

The iAF Repository Architecture

Repository Layers

The iAF Repository uses three separate layers of definition to describe applications:



The Database Definition Layer

This describes the physical database, and everything that can be usefully attached to this. We describe the tables, table columns, and domains (data types) for each column. We define the indexes on tables, and relationships between tables. In many cases, the database definition can be extracted automatically from an existing database (called 'reverse engineering').

We use the Database Framework Language (DFL) to describe the database definition layer.

The Object Definition Layer

This describes the business objects (we use objects only in this sense). An object consists of a set of database tables, and some intelligence. The object data can be shown in one of multiple 'views', possibly depending on its state.

We use the Object Framework Language (OFL) to describe the object definition layer.

The Presentation Definition Layer

This describes the user-interface: what data is shown, what actions are possible, and how the user-interface is constructed from screens.

We use the Presentation Framework Language (PFL) to describe the presentation definition layer.

General Comments about XML Files

XML files are text files formatted using a mark-up syntax similar to that used in HTML. XML is an updated and simplified incarnation of the SGML (Standard Generalised Markup-Language), used to define HTML. XML and SGML are both 'meta-languages', while HTML is one specific instance of a mark-up language[2]. This means that HTML tags (e.g. <BODY>, <TABLE>, <A>,...) are predefined, agreed by a committee. SGML and XML allow you to define your own, arbitrary tag languages.

XML has more or less superseded SGML because while it offers similar possibilities, it is much more friendly. SGML will not do anything useful until you have defined a raft of document type definitions (DTDs), parsers, etc. Making a useful XML file can imply as little as opening 'notepad' and typing a few lines. XML DTDs are optional and much useful work can be done without them.

XML does not replace HTML. XML represents data, while HTML represents hypertext. You can convert data into hypertext, but you cannot usefully do the opposite. To give an example: you can store donor information as XML. You can display it as HTML.

We often talk about 'XML files'. In fact this is short-hand for saying 'a text file constructed using an XML-specified tag language'. We may give this tag language a name, for instance 'XSL' for 'XML Stylesheet Language', or 'OAL' for 'Object Access Language'.

An XML file, whatever the specific tag language, follows certain rules that make it possible (and easy) to construct a parser that will read *any* XML file. Like HTML, items are tagged using '<' and '>', and can contain attributes in the form 'name = "value"', e.g.:

```
<donor email="me@somehost.com" name="John Doe" ></donor>
```

Items can contain a value between the opening and closing tag:

```
<donor email="me@somehost.com">John Doe</donor>
```

Unlike HTML, the closing tag is always required. You cannot omit this. There is one exception – XML lets you use a shorthand form for items which have no value:

```
<donor email="me@somehost.com" name="John Doe" />
```

Here the closing tag is implied by the '/>' sequence.

The main differences between XML and HTML syntax are:

- As we said above, XML items must be marked by closing tags. HTML lets you omit closing tags for many item types, like <P>.
- XML item and attribute names are case-sensitive. <Donor and <donor do not mean the same thing, although the program handling the XML may decide to forgive case differences (GSLgen does this, for instance).
- The XML file may have only one root item.
- XML items must be strictly hierarchical.

[2] Strictly, HTML is a set of languages originally based on formal SGML document-type definitions, but much abused by proprietary extensions.

On sites like xml.org and xml.com you will see references to DTD's, stylesheets, and so on. These are optional techniques for making XML files more and more formal. XML is usable and useful with very low formality levels. This is important because each additional level of formality makes your work process slower, more complex, and more rigid.

Design Features of the iAF Repository

1. The iAF Repository is ***executable***. This means that it can be turned into program code through a code-generation process.
2. The iAF Repository is ***normalised***. This means that it is designed to be as efficient as possible in terms of avoiding repeated definitions. This is a key to reducing the cost of maintenance of applications designed using the iAF Repository. Normalisation means that each layer plays an important part in the overall design. For instance, adding a certain data element to a table can cause changes in the behaviour of objects based on this table. The rules that describes this specific behaviour can be attached to the data element, rather than to the objects that indirectly use it.

The iAF Repository is used at build time, not at run time. A run-time repository suffers from two major problems. Firstly, it slow-downs all applications that use it. Secondly, it encourages the design of a non-normalised meta-model to improve response times. We avoid these issues by using the iAF Repository as the input for a code-generation process.

We used these criteria when designing the iAF Repository:

- Performance. There are a number of rules that we can apply to avoid obvious performance problems in database applications (such as indexes for searches). We try to integrate these rules into the design.
- Portability. The iAF Repository can target applications running on any database product and platform. We integrated specific techniques for achieving portability into the design, such as the use of abstracted data types in the DFL.
- Clarity. We designed the iAF Repository languages to be as clear and efficient (in terms of notation) as reasonable. The intention here is that it should be easy to read and manipulate for human readers.

Support for Development Teams

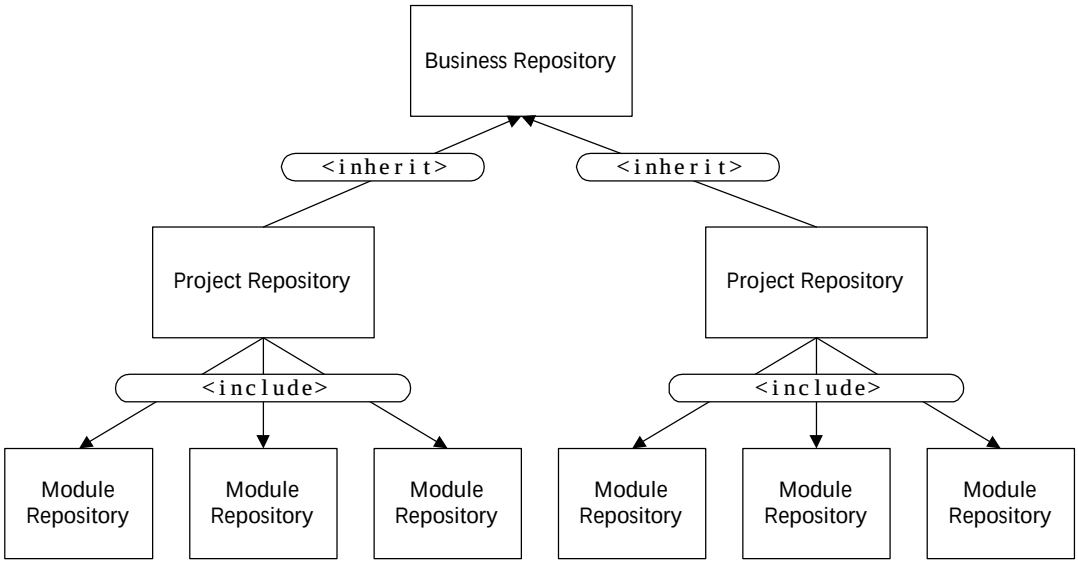
The need for normalisation leads us to define meta meta data. Principally, we define object *classes* and data *domains*. These definitions are special in that they can easily be shared between projects using the inheritance functions described below.

The iAF Repository exists as a set of XML meta-data files, which are stored on disk as simple text files. These can be accessed and modified by users with the appropriate rights. Two users cannot modify a text file at the same time, so the iAF Repository provides functions to split repositories into multiple files, using two basic mechanisms:

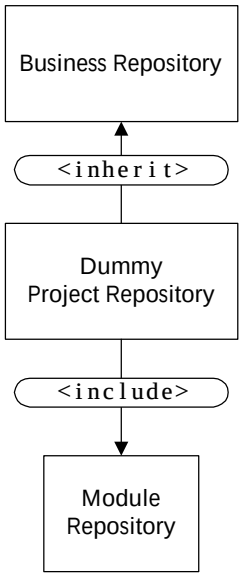
- A repository file can ***inherit*** definitions from another file. Inheritance incorporates specific parts of one repository file into another, and allows common definitions to be centralised and managed across applications.
- A repository file can ***include*** the contents of another repository file and allows project repositories to be constructed from multiple repository files.

Inclusion and inheritance appear similar but are not. Inheritance allows one project to reuse meta meta data (domains and classes) from another project without including its objects and tables.

A typical large-scale iAF Repository organisation might look like this:



A single developer might work on just part of the project, using a dummy project repository:



Source Code Management

When we use a code-generation approach we cannot use traditional source-code management techniques. That is: there is no value in recording changes to generated code, or being able to revert to earlier versions of generated code.

Using iAF, a source code management scheme must track:

1. The iAF Repository files.
2. Hand-written source code and source code libraries used by the iAF templates.
3. All GSL templates used in the project, including the standard iAF templates if these are customised in any manner.
4. The source code for certain tools, especially the GSLgen code generator, since this can be expected to evolve during the life-span of any project using it.

All these files are plain text files and can be stored and managed using conventional source code management techniques.