# Table of Contents

# Between dream and dread: a repository

Usually most discussions of the "new economy", and the "new digital age", tend to be full of poets & dreamers. Lot’s of"big picture", no details.

And most discussions of software & technology tend to be developers asking questions about how to install Visibroker 3.1, and is its Naming service CORBA compliant? All details, no "big picture".

(Ron Resnick, dist-obj mailing list)

# SimpleMind - a distributed repository

## 3. Purpose of this document

This document describes the requirements for a distributed repository for development and runtime systems. It discusses the technologies that could be used to implement it and proposes the implementation of a simple but effective repository dubbed "SimpleMind"

SimpleMind is a generic repository, more mechanism than policy and can cover a wide range of meta-data needs. It even supports meta-data driven systems and programming.

## 4. Next Steps

- Define the paradigms for distributed applications (programming model, build process, runtime and system management) using the SimpleMind concept
- workshop on XML/XLL/HyTime with specialists
- get a HyTime (nowadays a TopicMap) engine provider

## 5. A Repository – a Hypermedia System?

The following is a collection of - some might say religious - statements on what the current state of affairs is, what a repository nowadays has to provide and on which technology it should be based.

The most radical approach used in this document is the decision to consider a repository to be some kind of Hypermedia system. Structure, addressing, linking and schedules are what a generic hypermedia system provides and those are also the features needed to form a repository.

Following this approach the repository content for a bank is seen as a hyperdocument. The sturctures and relations expressed in this –ever growing and changing- hyperdocument are handled by a generic hypermedia engine with specialized behavior attached to it.

Behind all this is the believe that even with object technology being used, the concept of information is still valid and that not everything is usefully represented as a **behavioral** object. Representation of information needs to be separate from behavior to enable processing of this information in different contexts. Just as I can use a book to derive articles from it, to search for topics in it and maybe even use it to level my table with it, I cannot foresee all the contexts a piece of information might be used in.

Does this mean that object orientation with its tight coupling of data and behavior is wrong? Not at all. It turns out that OO is an excellent approach to handle information. The difference to regular OO concepts lies in the fact that behavior and information are dynamically associated and can therefor be combined in ever new ways. Typical examples for this approach are Applets or the Coins project. We will see how e.g. the bureaucracy pattern can be used to implement a system that ensures structural and semantic validity during constant changes from outside. (PLOPD 3)

And there is also the believe of the "machine" working on a higher abstraction level than the usual use case based business analysis. It needs to have higher abstractions, otherwise every change in the business use would end up needing a new processing machine. And no matter how good the prototyping or development environments are, a large bank could not survive with this model.

We also see the current trend to more and more meta-information to continue, especially in distributed environments using static typed languages. Large CORBA systems are either meta-information driven or they won't survive the need for constant changes. Not only the business related functions need meta-information - the machine needs a reflective view of itself, e.g. to find out about the consequences of a certain object/server being unavailable.

Another assumption targets the new relations between programming model, build process, runtime system and

system management. While with previous programming paradigms (see below "Old style static programs" these aspect were kept quite separate of each other, distributed system make the borders fuzzy and require a new look at their relations. The idea here is that repositories used to be some kind of add-on for change impact analysis they are now necessary at runtime just to keep the system going and in a valid state. Developments like the "deployment descriptors" in Enterprise Java Beans and the massive amount of meta-information generated and used in Component Broker are only a first glance at things to come.

The biggest problem:

After working several years with meta-information driven systems I came to the conclusion that it is not so much the technology that is missing or hard to understand. The real problem is to get people to come up with a model of the problems and to make this model available to the machine at runtime using a standard markup language.

## 5.1. Current State

1. No common data formats available.

This is changing rapidly with XML currently being the fastest moving internet standard. Almost every software vendor already announced compliance with XML (Oracle, Sybase, OMG, ...)

Still, there is a lot of legacy that needs to be accessed and connected. Therefor a generic translator framework is necessary.

1. No COTS Repository available

Currently there seems to be no 3rd party tool available that would fit the bill of a distributed, active repository for development and runtime. Quite the opposite is true: almost every tool nowadays comes with its own repository and all formats are incompatible.

It is not acceptable that to integrate a new product into an existing repository, the repository vendor has to develop a new interface module.

1. Incompatible repositories

Two ingredients are needed to enable compatible repositories:

1. a common syntax and data format
2. a common meta-format

The industry is currently standardizing on XML as a base level syntax and data format. Above that a way to express entities and their relations are needed. This will be (at least in parts) provided by XLL, the XML linking specification. Currently only HyTime provides all the necessary addressing and linking concepts to express and locate meta-informations. HyTime is based on SGML and can be expressed with XML (it is just an enabling technology that re-uses SGML/XML)

1. Heavyweight solutions

A key element of the success of WEB technology has been the introduction of a document metaphor instead of

concentrating on heavy-weight programming API based solutions.

A new repository should follow this metaphor. This does not mean that SimpleMind needs to be a WEB solution, especially since there are currently a lot of restrictions due to limitations of the http protocol with respect to cache control, replication, transactions etc.

1. central and closed repositories

It will not be possible to always integrate (extract and source into) information from other tools and repositories into one central repository. A repository needs to be fronted by entity manager that can lookup information from several physical locations (files, databases and the web) at runtime and tie them together. New physical media need to be integrated dynamically.

Tying information together does no longer mean to MOVE this information into one database. Linking mechanisms can be used to achieve the same effect while using federated repositories that look like one big repository to clients.

The technology necessary to do this already exists and focuses on a unified naming concept (so called formal public identifiers that do not encode physical location properties directly), entity management based on chain-of-responsibility patterns and distributed computing using CORBA servers.

1. development repositories only

The methodology behind development repositories allows change impact control. It does not allow dynamic evolution of systems. It does not support a defensive, generic way of programming (supported by frameworks) that would require meta-information at runtime

As we currently see with CB-Series, more and more meta-information is caught during build-time and sourced into the runtime system via CB System Management. This trend will continue.

1. Performance and replication issues

Performance - especially when we talk about a runtime repository - is essential if programs have been written according to the development standards. These standard say that no meta-information has to be hard coded. A consequence of this is that without runtime access to this information the applications or services will not run.

One way to achieve performance in lookups is to replicate the information. With CB-Series Workload Managed Objects we can simply configure a repository service to be replicated on several machines. This is a simple configuration issue.

Updates, aging mechanisms and even clients registering for real-time updates are possible using standard CORBA and CB-Series features like event handling.

1. Active Repositories

To achieve an active repository the tool integration must be driven much further than it is now the case. It is certainly a very attractive goal but nothing that can be achieved over night because even with the "SimpleMind" approach many things like certain tools are just not under our control.

1. Validation problems

Validation requires an explicit meta-model of content against which instances can be validated automatically. If

these meta-models do not exist (or ar hidden in code or word documents) no machine can check for inconsistencies.

Validation has two aspects:

- structural validation
- semantic validation

structural validation is fairly easy and already provided in an SGMl/XML environment by standard parsing technology.

Semantic validation requires code to be associated with the information.

A current validation problem lies also in its local-only and static behavior as opposed to a concept of distributed validation. Most compilers e.g. validate statically against a local copy of a meta-information schema (e.g. class definition file). In a distributed environment this can lead to runtime errors. In addition to that large-scale systems need to run several versions of programs at the same time because it might be impossible to do all upgrades at once. In CB-object versioning we see e.g. meta-information used to validate a request from a client for a specific service.

## 5.2. Requirements for a repository

### 5.2.1. Functional Repository Requirements

1. Complete decentralization of architectural authority while providing for perfect reusability of all architectures.

The repository can maintain and control different architectures (DTDs, Meta-DTDs, see Architectural Forms in the appendix)

1. No loss of control over information in the repository

While most software technology takes away control over a document/information from an author the bank repository will not allow vendor lock in, proprietary binary formats etc.

1. Information human and machine readable

The goal is to provide repository services to developers AND programs, at development AND runtime. This of course requires that the information is both human and machine-readable.

1. A concept of structural and semantic validation

Semantic validation needs to work both atomically (with an element or object) and global (within a whole environment), e.g. making sure that changes in one part of the repository are propagated to all dependent elements. In addition to that, validation needs to work distributed, e.g. drawing information from abstract storage systems (entity management) at the moment they are needed.

1. descriptive information dynamically combined with behavior

The information in the repository is descriptive. Necessary special behavior gets associated/created - again driven by descriptions

1. SLL (single logical location): Thou shallst not copy!

All meta-information used comes from the single logical location in the repository. It flows from a tool to the repository and back to this or other tools.

This is true for GUI building information (we don't allow local gui resource files to be used. These resource files usually just copy information about objects like customer etc. As soon as customer changes in the model the GUI resource files need to be updated locally too. Database schema files also do not originate in a database to be sourced into a repository later. The model information comes to live in a repository and database schema files are generated from it.

1. content: both hierarchical and linked

While a lot of information can be expressed as a virtual huge document (e.g. the web) most information contains non-hierarchical information that is expressed via links between elements in the same or different documents.

1. Flexible addressing and queries

To address elements of information various address modes must be provided. Examples are tree addressing modes on position, element properties etc. as well as document centric queries like "find me all X with an attribute of Y that have links of type Z to elements E".

1. availability at runtime

The repository must be available at runtime from clients or server/middleware programs. Its availability is mission critical. A repository service must therefor scale well with a growing and changing organisation.

1. Reference Codes, Dependencies, structural information

The repository must be able to serve all types of meta-information. This information could be used for change impact analysis, generation of source code or runtime dynamic lookups.

1. A concept of time

The meta-model of the repository must be able to address entities across time and space.

1. safe structural transformation

The repository meta-information allows the safe structural transformation from system release X to release Y. This is possible because content descriptions (DTD's) exist for all information and instances of information elements can be validated against them.

1. Granularity

The repository must allow access on 3 levels of granularity: Blob, entity and element. Blob e.g. is a ready-to-print document that has been released. Entity is a (reusable) part of information. Element and ElementCon-

tent are fine grained pieces of structure and information that can be accessed with queries or Xpointers. Elements reflect e.g. the things that make a class in an OO-language.

### 5.2.2. Technical Repository Requirements

1. Express syntax and meta-syntax using a markup language
2. distributed
3. federated
4. replicated
5. scaleable and available
6. open input/output framework, no vendor lock in.
7. associate behavior dynamically
8. one document based interface
9. re-use of basic elements for meta-elements and links (meta-circularity)
10. links typed and expressed explicitly
11. automatic content validation through meta-information
12. mapping to reliable RDB
13. document centric query language
14. use of standard XML tools Web Browsers and editors
15. support for fully automated setup of development stations and a fully automated build process from scratch (I-make pattern)
16. integration with all development tools

### 5.2.3. Programming and Use Requirements

1. absolutely no hard-coding of meta-information
2. use of symbolic information restricted to typed XML/DOM composite objects. No encoding of complex information in strings.
3. Information exchange between Regions or Applications uses XML document types or objects but no private/binary formats.
4. use of formal public identifiers to name and reference information
5. Modeling information goes right into the repository without loss of information
6. absolutely no use of private configuration files, information access interfaces etc.
7. Capture as much meta-information as possible during development and express it using a markup language (e.g. use a tagging scheme for comments in code to extract meta-information.)

### 5.2.4. Political requirements

no tools are allowed to be used that are not able to export information in XML

## 5.3. Technologies to be used

Again, the goal of SimpleMind is to use existing technology like XML, CORBA, CB-Series and WWW to provide repository services to developers AND programs, at development AND runtime. The reason for this is that there is nothing special about "meta" information processing. (There is no "meta" property anyway. An information is a meta-information only because of its relation to other information: it talks about it)

If I would start building a repository from scratch, only about 10% of the code might deal with handling and tracking the meta-information. 30% of the code would be conversion modules that handle different data formats for input and output and the final 60% would deal with those processing requirements that almost every other application has to do with too:

1. Storage Systems (databases, files, web)
2. connectivity (finding, access, transport of information)
3. Notification mechanisms
4. security (access control)
5. reliability and performance (replication)
6. modeling (syntax, meta-languages, modeling languages)
7. interfaces (API for GUIs and programmatic access
8. publishing/printing

Instead of building a repository from scratch (or buying one that does not do the job) another alternative is to use what we have to build a repository from components.

What do we have now that might facilitate a component based repository?

1. CB-Series as a CORBA based framework that handles locating, persistence, transactions, replication etc. No need to define/program our own persistence layer just for the repository. Language independent access is provided too.
2. XML. No need to define our own low level syntax. We can buy utilities (e.g. editors, converters, translators). We can buy education and knowledge too.
3. DOM, the document object model. No need to define a new interface to the repository content. The content is represented by DOM objects. We can buy tools that understand this interface (e.g. Microsoft or Netscape Browsers)
4. RDF, the resource description framework. No finished yet but gives a clear impression on how to express resources and dependencies using XML.

What is missing then?

1. Informational Objects - A framework with generic objects (supporting DOM) that carry informational content. The Application Infrastructure Group within BLD is providing an implementation including entity management (how to map e.g. public identifiers to system identifiers and the proper storage subsystem) OO-RDB mapping (how to map information trees to DB2 efficiently using CB).

2. A meta-model of what we want to express or capture in the repository (.e.g. using XLL/HyTime linking definitions).

3. the build process needs to be modeled.

4. reference codes need to be expressed with XML/HyTime features

5. A translater/converter framework. This is necessary to get new non-XML information into the system (and out again) as well as to translate one structure description into a new one (a totally different one or just a new release)

6. A service concept, using information servers (high level entries to meta-information), entity managers (catalog driven dispatch of entity requests to different storage managers). This is also the way to integrate existing repository servers and services like "reference codes"

7. Test on how many replicas of repository services are needed to serve large numbers of clients.

8. A web server that serves repository data through XML elements

9. Behavioral objects that perform specific requests for meta-data. The activation of these behavioral objects finally form the repository front end. They are again assembled from and driven by meta-information. These behavioral objects are finally an equivalent for a HyTime engine: they track links (document internal links) and new behavior can be added dynamically.

10. More tools that understand and generate their model in XML. Rational Rose and the CB-tools like Object Builder do this already now (OB) or in the near future.

# 6. Meta-Information

The use of meta-information in programming is tightly connected to the roles a repository plays in a system. We will look at three different programming paradigms and how they affect repositories.

**This chapter is fairly "development" centric. This does not mean that the repository concepts introduced later a in any way limited to support development.**

## 6.1. A Meta-information standard?

I don’t expect a common meta-format to exist in the near future. While this is a problem for programming centric approaches it has never been one for the SGML community. Remember: SGML/XML is based on the belief that an author should not lose control over his or her information. So SGML/XML are designed to let people express their information with markup according to THEIR needs and to let the processing applications deal with different markup.

Should we even try to achieve one central Meta-Model? I don’t think so. Why should e.g. operational information be forced into a OO-class analysis and design?

The most important prerequisite for a repository actually has nothing to do with the repository itself. If the users/feeders of a repository do not tag their information to make the content structure explicit a repository is pretty much useless. But if the information comes with markup – any markup – other tools/users can work with it and there is no need to define the one huge and global environment encompassing all tools.

## 6.2. Old style static programs

### 6.2.1. Programming Paradigm

The typical old-style program is created using a compiler that works locally. The compiler controls its environment and ensures that types declared match those found in instances etc. Another typical feature of those pro-

grams is that they end up being physical "blobs" in most cases. This means that one can literally carry around a floppy disk and install the program on a machine.

Those programs carry around most of their environment embedded in the "blob" and are usually replaced/updated by installing a new version of the blob on a machine.

All system management (installation, versioning) and product development (responsibility) is focused on the blob idea. The identity of the blob is the anchor for version management.

Another typical feature of this type of programs is that they are pretty much self-contained. In many case the only dependency they have is on data schemas of a database. If they use other functionality they follow the "link pattern": libraries are linked to the program that provide the needed functionality. It is no conincidence that re-use of other components is limited in those programs because the infrastructure to support that re-use is not in place. Re-use also violates the principle of "self-containedness" that is used in this paradigm.

### 6.2.2. The build paradigm

"Building" such a program in the sense of an automatic build, e.g. for integration tests, requires the assembly of the correct source files and the correct compiler options. Correctness is defined as working together to produce a runnable image. It does not mean that other programs could not at the same time be built with newer versions of classes. This results from the self-containedness of this type of programming model.

The most critical build items are:

- timestamps
- version numbers

### 6.2.3. The runtime paradigm

The most critical runtime items are:

- Operating system versions

### 6.2.4. Program Evolution paradigm

If central data structures in databases are changed this type of program usually needs to be changed and recompiled. The answer to environment changes is the change of the program.

### 6.2.5. The role of a repository

Within the above programming paradigm repositories are mostly used for change impact control. Information about database schemas is published as well as information of which program identity works with what schema. Changes to the schema are published using the organization infrastructure, e.g. by issuing remedy requests against a change control system.

This system relies on organizational efficiency in performing the necessary adjustments to programs if schemas were changed.

The progams themselves in many cases contain hard-coded expectations about schemas. In some cases local environment variables or configuration files can be used to adapt the program to new schemas.

The runtime environments of large companies contain often a runtime component called "CodeDB" that is fed from a static repository "DBDOC" and serves so called reference codes. These codes are fairly stable and hard coding them is avoided by using CodeDB.

A third type of repository is sometimes hidden in the development process: Project configuration files, either

from IDEs or from an automatic build environment like IMAKE.

## 6.3. Programs using Dynamic Load Libraries or Packages

### 6.3.1. Programming Paradigm

Before we tackle the problems of "real" distributed programs we need to talk about a very primitive form of distribution that nevertheless has never been performing in a safe way: The use and sharing of Dynamic Load Libraries (DLLs) or Java Packages.

This approach tries to provide better re-use of code by seperating it into different logical and physical modules as well as to achieve better maintenance through transparent exchange of those packages without re-generating clients.

The technical means to achieve this is via interfaces. Interfaces separate the client from a module.

### 6.3.2. The build paradigm

Building systems that use DLLs requires besides the timestamp and version features something new: A interface module for all external DLLs that gets linked into the program. The program then "knows" how to call services from DLLs. What it does NOT KNOW is: Will the DLLs that is present on the final system be compatible with the interface module?

The build process must:

- generate package version information that can be queried at runtime
- generate dependency information, e.g. if inheritance is used ACROSS DLLs so that a package can state its physical dependence of a certain version of a base class. In most cases these derived packages will have to be regenerated, at least in C++.
- Make sure that the proper interface libraries are used.
- make sure that the proper packages are used, especially in the case where the same set of classes are used in different packages, e.g. as CB does with the Java client/Server runtime package.
- Generate physical module information about the requirements of a client on the runtime environment. This information is sourced into the repository.

### 6.3.3. The runtime paradigm

The most important thing for a system that uses DLLs is to control loading of DLLs. This is an automatic process on most operating systems. The semantic is like this: a general "path" string tells the OS where to look for DLLs in case one is needed. It will load the first one it finds with a matching name. There is NO guarantee whatsoever that the program interface module fits to the DLLs. Also, if two programs would need two different versions of one DLL, they would have to run on two different machines.

The most critical runtime items with DLLs are:

- Absolute control over DLL locations by generating path lookup information
- Strict rules against copying of DLLs to different locations ( I’ve seen it happen already in CB development: application dlls were copied and after re-generating the app and dlls the system still picked the old ones)
- Careful maintenance of interface compatibility
- Careful maintenance of all compiler options as these might generate DLLs that have the same interface but

work quite differently (e.g. single/multithreaded versions)

- Checking of "package" information at DLL load time (the program loads the DLLs under its control
- Operating system versions

Solutions like time-stamps, version numbers of source modules and package version numbers are not really sufficient to provide a safe and convenient use of DLLs. We are facing the following dilemma:

- We want to replace DLLs transparently for the client programs, e.g. for software maintenance reasons.
- We also want to make sure that a program that has been tested with DLL version A also works with the newer version B. Possibly without running the same big testsuite as we did when we tested the first time against DLL version A.
- But in case the new version is not really compatible we would like to see the older client to stop gracefully instead of crashing or in the worst case to perform wrong calculations.

On top of these problems operating systems like NT do not allow two different versions of one DLL to be used concurrently. This is a physical limitation that affects CB versioning too.

### 6.3.4. Program Evolution paradigm

In many cases newer versions of DLLs are installed on a system e.g. to fix bugs. The new DLLs needs to be upward compatible (this in turn requires a binary standard for the programming language, something that did not exist for C++ for many years).

It is not necessary to update all systems at the same time. Every host is kind of an island and as long all the DLLs and packages are consistent the host should be fine. This is different to the case with real distributed programs.

### 6.3.5. The role of a repository

Besides the previous roles a repository has to meet new requirements:

- Not only DB changes now have an impact on clients. When DLLs or packages need to be changed the repository needs to provide impact analysis data on which clients are affected.
- The repository needs to be able to tell administrators about the requirements of a client with respect to other DLLs or packages. This information comes from the build process.

It turns out that without the meta-information already coded or generated in the build process, there is little control over the runtime system. This finally turned out to be a problem with the concept of interfaces. Interfaces are now known to provide not enough information for a safe connection between clients and services. Implementation characteristics (properties), the usage protocol, preconditions and constraints etc. are all missing.

## 6.4. Distributed Programs

There are a number of issues that add more problems to distributed programs compared to just using DLLs or packages:

- The physical granularity is much finer: instead of large modules we talk about individual objects
- A result from the finer granularity is a much bigger number of "classes".

- Finding a needed module or class is much more complex than in the local case.
- Distributed objects are GLOBAL (at least they have the potential). Updates can affect a huge number of systems all at once (which is nice but also very dangerous and sometimes not even wanted)

### 6.4.1. Programming Paradigm

The programming paradigm for distributed programs turned out to be very different compared to the static-local ones. It is centered around "Black box" and "interface/implementation" separation. The basic idea behind is to create programs that do not include every object they use locally and to hide the possible "remoteness" of those objects. Little changes happened to the sequential aspects of programming because of the mostly synchronous calling features of systems like OMGs CORBA.

The CORBA service programming paradigm faces 3 problems:

- The "Black box" technology is useful only for certain clients. It certainly does not make system management easier since it usually requieres knowledge about the implementation, e.g. to make sure that services required are also available.

- Interface definitions do not cover the whole contract between a client and a server. A new server might need a different order of calls to perform a certain task (protocol) or provide an implementation that violates client expectation even so the interface is compatible.
- Composition is possible only on the implementation level, in other words, it is programmed composition and as such not very flexible.

### 6.4.2. The build paradigm

Building distributed systems happens today in a fairly static environment. A development environment uses static IDL definitions to generate client access stubs that are able to talk to remote services. If these IDL definitions are not compliant with the services used a runtime error happens.

From the beginning of CORBA programming there was a socalled interface repository. It contains a description of the IDL modules, interfaces and attributes. It is mostly used for Dynamic Invocations (DII).

Most CORBA literature mentions its possible use for compilers to gather runtime interface information but I have yet to see one that does it.

The interface repository is also not useful to store other information like deployment related data. If you look carefully at CORBA programs, they contain a lot of string encoded information about the environment, e.g. naming service path information, names of factories or services etc.

### 6.4.3. The runtime paradigm

CORBA programs face mostly 3 runtime problems:

- a service must be available
- a service must provide a compliant interface
- several versions of a service need to be available

### 6.4.4. Program Evolution paradigm

If a service interface changes in an incompatible way, clients need to be re-generated or re-written. Since IDL

provides static typed interfaces this is is major problem for large scale systems.

### 6.4.5. The role of a repository

## 6.5. Component Models

### 6.5.1. Programming Paradigm

### 6.5.2. The build paradigm

### 6.5.3. The runtime paradigm

### 6.5.4. Program Evolution paradigm

### 6.5.5. The role of a repository

## 6.6. Meta-data driven programs

Under the topic of "design patterns for change" a new breed of software architecture has been developed over the last couple of years.

The central idea is that the whole behavior of e.g. a framework is DRIVEN by meta-data. Changing meta-data changes runtime behavior. The location of change is moved from the base level to the meta-level. This results in the base level being much more stable than before. Runtime composition of the whole processing context is quite normal in meta-data driven frameworks.

### 6.6.1. Programming Paradigm

An example describing the programming paradigm behind ARGO, a repository based framework for evolutionary software development:

*"The framework is based on a repository in two ways. First, it consists of a set of tools for managing a repository of documents, data and processes, including their history and status. These tools enable users to select applications, enter and view data, query the repository, access the thesaurus and manage electronic documents, workflow processes, and task assignments.*

*More importantly, the framework behavior is driven by the repository. The repository captures formal and informal knowledge of the business model. None of the business model is hard coded. The tools consult the repository at runtime. End-users can change the behavior of the system by using high-level tools to change the business model. Thus we separate descriptions of an organization's business logic from the application functionality."*

### 6.6.2. The build paradigm

### 6.6.3. The runtime paradigm

### 6.6.4. Program Evolution paradigm

### 6.6.5. The role of a repository

## 6.7. The "liveness" problem of distributed systems

A distributed algorithm is an emergent phenomenon. It is not completely defined by the sum of the local algorithms.

This fact changes the role and function of validation: It has to work at runtime too.

# 7. SimpleMind

In the following chapters we will develop a design for SimpleMind, going to increasingly more involved steps to finally achieve a distributed and federated repository.

Each step will have a clear deliverable and the experience won will influence the next step.

## 7.1. First step: a simple WEB-based repository

To test the concept of a component based repository we should implement it using a web server that serves XML information to clients.

Please note:

This is not just a GUI interface to the repository information based on WEB browsers. It is a generic API to access structured information either interactive or from within programs and used the WEB document model for access.

Here we need to have a look at:

1. What is the information we want to serve/maintain?
2. How much is static/dynamic?
3. How do we describe it in XML?
4. How do we describe dependencies?
5. How do we add behavior?
6. What is the status of web technologies like RDF?
7. Which DOM mapping can we use to access the information on the client side?
8. How do we model our name space? (identifiers, entities etc.?
9. How do we structure the information (logical and entity structure)
10. A name space and service "shell" around the repository information to provide single logical access. This is actually most important for the federated repository. Here we just need the interface definition in place so we don’t need to change clients later.

Since we are basically using existing products/standards etc. most of the work will be in how we want to model our repository information. Getting familiar with the latest web technologies is another topic.

Within 3 month programs should be able to access this repository through standard interfaces.

What could it be used for?

1. Programs can use it to access configuration information in a standard way. This includes handling of "environment variables"
2. Programs can access "Reference codes" through the web-based repository

3. The build environment can use it to create specific development platforms and environments.

4. Initial change impact analysis can be developed.

5. Browsing the repository information can be done using standard WEB browsers and tools. Editing is done with standard XML editors. (These editors will by default only be able to ensure instance validation against a DTD, no semantic validation can be done)

The functionality of step one would be enough to start application programming without the need to hard-code meta-information or reference codes.

At the end of step one we need to have a close look at where the current web-computing might fall short, e.g. cache control, change notifications.

## 7.2. Second step: additional dedicated repository servers

These servers will use CORBA and IIOP to serve repository content to clients. They implement a document based API just like the WEB servers

Here we need more "heavy-weight" technology. Luckily most of the basic functionality is already provided e.g. by Component Broker.

1. efficient mapping of XML tree information to DB2 or other databases via CB adapters

2. a concept of information granularities: when do we need documents (blobs) entities (parts of blobs that might get reused) and elements (fine grained logical parts) and how do we map those to storage mechanisms

3. Implementation of front end servers: information service, entity manager

4. An efficient DOM mapping to CORBA/CB (informational objects)

5. The result and biggest difference to step one lies is the fact that now specialized behavioral objects can also control modifications to the repository. The semantics are expressed in XML too and behavioral plug-ins enforce them.

Using Component Broker features and map the DOM to it will take around 1-2 month.

## 7.3. Third step: a federated, distributed repository

In this step we create a logically centralized repository by federating web-servers and dedicated CORBA servers behind front-end information servers. This finally creates a distributed HyTime engine.

# 8. Applying SimpleMind in the BLD environment

This chapter tries to tackle certain BLD specific problems

## 8.1. Meta-information provided by CB-Series tools

## 8.2. A BLD meta-model

## 8.3. BLD programming paradigms

### 8.3.1. Programming Paradigm

### 8.3.2. Build Paradigm

### 8.3.3. Runtime Paradigm

### 8.3.4. Program Evolution Paradigm

### 8.3.5. The Role of SimpleMind in BLD

# 9. Appendix

## 9.1. The Power of Markup

Markup is the technology to extract information from the heads of developers or from specific source code and make it publicly available. Other users/programs can now work with the information.

And technically speaking, markup is META information. In the case of SGML/XML this meta-information usually comes embedded into the information and makes the information self-describing.

An example shows the biggest difference between information with and without explicit markup:

A piece of program code:

Class X {

The same piese of information with markup:

<ClassDefinition lang=C++> class

<ClassName> X </ClassName>

<BlockBegin> { </BlockBegin>

</ClassDefinition>

What is the difference?

To process the first information I need a C++ parser. Only it will now that the class keyword starts a class definition. And it usually won't let me interfere with its activities e.g. whenever the parser found a class definition it should replace it with a certain macro "CLASSDEF".

To process the information that has been marked up no C++ aware parser is needed. Any XML Browser can display it. Any XML parser will send me events about what it found. A tiny appication can easily change "class" to "CLASSDEF" because it need not deal with any physical characteristics of the information e.g. it need not try to find "class" strings because the parser will autimatically notify it when it found an element "ClassDefinition" in the input stream.

And finally: if there is a DTD for this information the parser can validate the instance against it.

Of course, a C++ parser wouldn't accept the code with mark-up. But if the marked-up information is run through an XML parser the result would be ready for a C++ compiler.

How can markup be used?

Lets assume the sample above is generated code. If something changes it is much easier for tools (and not only the one that generated the code) to perform those changes in a structurally safe way. It is easy to insert e.g. bank specific code pieces because the proper locations are easy to find!

## 9.2. Why HyTime - or Naming, Addressing and Linking

Mark-up alone is not enough. The example C++ code above with markup let’s us do some processing without knowing C++. But we are still unable to express how things in this code might relate to other parts or how to find them. HyTime uses markup to express exactly this.

While for a web person concepts like XLL or HyTime might be very natural they are much harder to understand for programmers. All HyTime e.g. does is to clarify what’s behind a name, how to address things (entities, within trees, using finite-coordinate systems **over time and space** etc.) and how to express any kind of relation with external links. All of these features can be combined and form "location ladders", including queries. A key concept of those links is that they can be document external. That means they need not be anywhere near to the document or contained in it. They can e.g. address and link into a CD-rom with read-only content. Another consequence of those links is that clients are free to form ANY relation (e.g. a dependency) between ANY aggregation of objects AFTER the objects came into existance.

This concept seems to me one of the top requirements for a multi-entity system (german: Mandantenfaehigkeit)

This may not sound like a lot but it turns out that even nowadays these concepts are the base for almost everything in IT and HyTime as an enabling technology provides a clear and precise definition (these definitions can even be controlled and validated by machines)

Why are these concepts necessary?

One example are "dependencies". The concept of dependencies has evolved quite a bit in the last couple of years. In a development environment it used to be like this: a file contains a definition of a dependency by declaring file X to be dependent on file A. A time stamp was used to check the dependency. Especially OO-development is badly affected by this kind of dependency checking: it is too coarse grained. OO systems would like to know if a change affected the compatibility and they would also like to limit the actions needed to the minimum. Instead of starting a big compile only the affected methods are changed.

Developers at the same time would like to know about the effects on a whole environment if a certain method change is made. Or even that the system automatically changes names in other modules if the main name was changed.

What is needed is a better way to structure things: No longer is "file" an adequate concept. One needs to structure the content into e.g. method, parameter etc.

This of course requires a fine grained form of addressing that allows to locate small objects within larger ones.

And finally updating other objects after a change requires the system to KNOW (in other words to have a machine representation of dependencies) about dependencies and how to act on them. This is where a generic linking concept comes into the play.

Now, what’s good for developers is also necessary at runtime: object versioning in CB uses meta-information to associate clients with services depending on meta-information collected at build time. Simple Interface name versioning won’t do the job in a large environment.

## 9.3. WWW and Markup standards to watch

### 9.3.1. XML

On February 10, 1998, the Extensible Markup Language (XML) 1.0 syntax

became a W3C recommendation; see

http://www.w3.org/TR/1998/REC-xml-19980210

### 9.3.2. XLINK

XLink, is the linking language that governs linking from XML to anything. The latest working draft was released March 3, 1998 at

http://www.w3.org/TR/WD-xlink

The second part, XPointer, provides advanced addressing into XML

document structures -- the linking language that governs linking to XML

from anything. Their latest working draft was also released March 3,

1998 (same editors as XLink) at

http://www.w3.org/TR/WD-xptr

### 9.3.3. DOM

The DOM specification defines a programmatic interface for XML and HTML, and is separated into three parts: Core, HTML, and XML. Looks like DOM is stabilizing – the level one is almost finished (including navigation and content and structure manipulation). Incorporating events and styles, among other things, will come next. So far the specification has language bindings for Java and ECMA Script, and uses OMG IDL for specs. A new version of the DOM specification was released on April 16, 1998 at

http://www.w3.org/TR/WD-DOM/

### 9.3.4. XSL

The goal of XSL is to have XML syntax and be based on both DSSSL and CSS. XSL pulls from CSS: r-browsebased presentation, color, font description and "web fonts", and aural CSS. XSL pulls from DSSSL: multi-column text, multiple writing directions (e.g., vertical), footnotes, page templates, headers, and footers. Read about the latest XSL developments at

http://www.w3.org/Style/XSL/

### 9.3.5. Schemas

1. RDF – the resource description framework

http://www.w3.org/TR/1998/RDF

It is a schema using XML for its syntax that supposedly should turn the WEB into the largest repository ever built. Allows detailed descriptions of resources.

A nice introduction:

http://www.ccil.org/~cowan/XML/RDF-made-easy.html (by John Cowan)

1. XML-DATA

XML-Data's W3C note came out January 5, 1998 at

http://www.w3.org/TR/1998/NOTE-XML-data

and I'm not sure what its status is. Any resemblance to Microsoft Office data types is purely coincidental.

### 9.3.6. Namespaces

The Namespaces in XML specification came out March 27, 1998 at

http://www.w3.org/TR/WD-xml-names

### 9.3.7. General Info

General Info - XML's home base at the W3C

http://www.w3.org/XML/

General Info - Robin Cover's XML page

http://www.sil.org/sgml/xml.html

General Info - O'Reilly and Seybold

http://www.xml.com

General Info - Peter Flynn's XML FAQ

http://www.ucc.ie/xml

### 9.3.8. HyTime

SGML based ISO standard for Hypermedia structures

www.hytime.org

### 9.3.9. HTTP- Next Generation

www.w3c.org [http://www.w3c.org]

Provides more and better distributed computing mechanisms for the base web protocol. Watch out for transaction, cache control, client and server side proxy agents (interceptors), server callbacks etc.

This could turn into quite a surprise for our heavy-weight distributed objects friends (CORBA, DCOM etc.) How about just using XML for marshaling AND to express services (what IDL e.g. does today in CORBA environments?)

# 10. Literature


1. XML: www.sil.org/SGML/XML
2. XML tools: www.falch.no/pepper (whirlwind guide to SGML/XML tools)
3. XML: www.xml.org
4. HyTime: www.hytime.org
5. HyTime: David Durand, Steve DeRose, HyTime Introduction
6. HyTime: Practical Hypermedia, An Introduction to HyTime, Eliot Kimber
7. http next generation: www.w3c.org
8. www/http deficiencies (see also http-ng): Ken Birman, Building Secure and Reliable Network Applications
9. DOM: www.w3c.org
10. RDF: www.w3c.org
11. CB-Series: www.software.ibm.com/ad/cb

12. Design Pattern that support evolution and adaptability: Linda Seiters, Design patterns for change

13. Bureaucracy pattern: PLOPD Book 3

14. CORBA and meta-data: CORBA design patterns, Mowbray/Malveau

15. SGML and Object System integration: Frameworking – Strukturen und Verbindungen in einem Framework. Walter Kriha. Zeigt die implementierung eines meta-data getriebenen frameworks für document image processing und workflow. Leider in Deutsch.

16. XML and Java integration: The coins project by Bill la Forge, www.opengroup.org/~laforge

This chapter will include over time more and more things related to meta-data driven architectures, automatic and dynamic GUI generation, architectural patterns for evolution, automatic structural conversion and translation etc.