

# RIGR - A Model Based Approach to Repository Management

Ray Trask

Qwest, Dublin, Ohio  
ray.trask@qwest.com

Robert B. France

Department of Computer Science,  
Colorado State University,  
Fort Collins, Colorado  
[france@cs.colostate.com](mailto:france@cs.colostate.com)

**Abstract:** Organizations with large integrated computer-based systems are often faced with the challenge of evolving their mission-critical systems in a manner that maximizes reuse opportunities, minimizes redundancies, and addresses business goals. To support these goals system developers must be provided with tools that allow them effectively assess the impact of change across the systems (impact analyses), identify reuse opportunities (reuse analyses), identify gaps in existing system functionalities (gap analyses), and identify redundancies (redundancy analyses). In this paper we discuss some of the issues related to implementing a model-based repository capable of supporting Redundancy, Impact, Gap, and Reuse (RIGR) analyses.

## 1. Introduction

System architects are often responsible for ensuring that (1) an organization's computer-based systems evolve in a manner consistent with changing business goals, (2) changes in one or more systems do not impact negatively on the services provided by other systems within the organization, and that (3) effective mechanisms are in place to support the timely development of high-quality solutions. Model-driven, repository-based approaches to managing integrated systems have been proposed as effective means of managing an organization's integrated systems [Scheer2000,Erik2000]. These approaches can be used to build and manage models of systems using a variety of views. The models form the *Enterprise Architecture* (EA) of an organization.

Organizations often view their integrated systems as critical assets that need to be managed. EAs can be viewed as mechanisms that help an organization manage its system assets. In order to effectively manage the system assets, system architects and developers require

techniques and processes that support Reuse, Impact, Gap, and Redundancy (RIGR) analyses.

We are currently formulating a model-driven, repository-based approach to system development and management that is intended to satisfy the following criteria:

- Support for rapid development: The ability to develop and evolve systems products in a timely manner is a strong determinant of the success of an organization in a highly competitive business environment. The system development approach must support the use of development processes that allow developers to focus only on the information they need, when needed, in order to produce results in a timely manner. The use of agile (“lightweight”) processes to develop systems in “internet” time, coupled with the incorporation of systematic reuse practices can lead to order-of-magnitude reduction in development cycle times.
- Support for delivery of high-quality systems: The drive to reduce development cycle time must be tempered by quality forces. Rapid development should not come at the expense of quality - an organization that can produce high quality products in a timely manner has a significant competitive advantage. Systematic reuse practices, modeling and analysis techniques, and quality control mechanisms play an important role in producing high quality systems.

The proposed approach is model-driven in the sense that data and object-oriented (OO) models (expressed in the Unified Modeling Language (UML) [UML99]) are used to represent systems at various levels of abstractions (business requirements, system requirements, logical and physical design), and from a variety of views (static structural, dynamic/behavioral). The centerpiece of the model-based approach is a repository of interrelated system assets. System assets include existing implementation artifacts (e.g., components, reusable classes, code frameworks) and their models, as well as model and implementation artifacts from ongoing projects. Traceability links relate elements within and across system artifacts in the repository. Using the repository, system architects and developers can do the following:

- Perform impact analyses: Impact analysis is concerned with determining the impact of change on other systems, that is determining the impact on the organization’s ability to effectively meet business needs. The relationships among the artifacts in the repository (for example, data/object create, read, update, and delete relationships between applications and data/objects) can be used to determine the impact of planned changes and new features on existing systems and other ongoing projects.
- Perform gap analyses: As new processes and system functionality are developed, gaps in the existing integrated system need to be identified and filled. Gap analysis is concerned with determining the missing functional and process elements that need to be present in order to implement new functionality of processes. The repository can be used to determine what parts of a system are under development or already exists, and what parts need to be obtained from outside vendors or be built in-house.

- Perform redundancy analyses: As an organization's pool of systems grows, the need to identify redundancies to reduce inefficiencies and avoid conflicts arising from multiple representations of a single concept across an organization becomes evident. Redundancy analysis is concerned with identifying redundancies and resolving associated conflicts. The repository can be used to determine whether proposals for new system features can already be met by existing systems and to determine wasteful overlaps in system functionality.
- Perform reuse analyses: Order-of-magnitude improvement in productivity and system quality can be accomplished if developers reuse product experiences. A well-managed integrated system can form the basis for identifying potentially reusable experiences across an organization. Reusability analysis is concerned with identifying potentially reusable artifacts. Commonality analyses can be carried out on the repository to identify organization-wide and domain-specific patterns that can be packaged for reuse (e.g., as product frameworks, components, reusable models)

The ability to carry out effective RIGR analyses is key to providing support for rapid development of high quality system products. In this paper we discuss some of the major technical issues and challenges that we have confronted in our work, and present our approach to tackling some of these challenges and issues. We focus particularly on the role of the UML in our work. In section 2 we discuss the models and the proposed traceability relationships that we feel support Reuse, Impact, Gap, and Redundancy (RIGR) analyses. In section 3 we give an overview of some of the issues related to implementing a repository to support RIGR analyses. We conclude in section 4 with a list of open issues.

## 2. Providing Support for RIGR Analyses

Our approach is best illustrated through an example. In the following we present a fictitious company that has implemented a repository that supports the traces supporting effective RIGR analyses. We then describe our trace model in more detail and discuss some of the issues related to implementing a repository supporting RIGR analyses.

### 2.1. Effective RIGR Analyses: The AJAX Inc. Scenario

AJAX Inc., a large organization that produces a variety of communication system products and services, has implemented a repository that supports RIGR analyses. To focus our discussion, we limit our attention to the repository assets that are related to the business concept of a *Customer Account*.

The following reflects the current state-of-affairs for the Customer Account concept within AJAX:

- Realization traces: The Customer Account business concept in the repository has a realization trace to an implementation as an Enterprise Java Bean (EJB) component in AJAX's EJB entity system and to a CORBA object in the AJAX's CORBA entity system.

- Persistence traces: The Customer Account concept persists in an *Order Database* (OE), the *Billing Database* (BILL) and the *Accounts Receivable Database* (AR). Compounding this redundancy challenge, each database has a different table name for the Customer Account concept; OE.CUST\_BILLING\_ACCT, BILL.CUST\_ACCT, and AR.CUSTOMER\_ACCOUNT.
- Create/Retrieve/Update/Delete (CRUD) traces: The Customer Account entity is acted on by the Create Order, Bill Customer, Block Service for Non-Pay business processes (each described by a use case that has a trace link to the Customer Account concept). Within AJAX there are two order entry systems that realize the Create Order business use case: a legacy client-server order-entry system for older services and a web based order-entry system for newer services. The Bill Customer business use case is supported by a single billing system and accesses the Customer Account tables directly. The “Block Service for Non-Pay” business use case is supported by the AR system. For each business use case there is one or more representative interaction diagrams. The client-server Order-entry system accesses OE.CUSTOMER\_BILLING\_ACCT and BILL.CUST\_ACCT directly via a non-OO API. The new web-based order-entry system uses the EJB entity object and the A/R system uses the CORBA entity object to access the Customer Account information. The billing system accesses the BILL.CUST\_ACCT directly.

The above relationships are captured in the AJAX enterprise-wide system asset repository.

AJAX Inc. decides to start treating each customer account differently based on the total size of the account in terms of revenue from the previous month. It is apparent that a feature capturing the revenue of the previous month is required (this can be accomplished by adding a “Last Month’s Revenue” attribute to the Customer Account business concept). In the absence of a current system asset repository, a new analyst tasked with planning and implementing this change has the additional challenge of identifying and understanding the dependencies associated with the Customer Account concept from a variety of sources. Fortunately, AJAX Inc. has an enterprise-wide repository of its system assets that provides a single structured source of system information from a variety of perspectives. In the AJAX repository, business concepts are used as indices into the repository (other forms of indexing are also available, e.g., indexing by project, program).

The analyst uses the repository to produce RIGR reports that helps him/her determine potential reuse, impacts, gaps and redundancies. The analysis process can start by generating the following report that shows the CRUD relationships between the concept and the system-supported business processes.

Business Concept	Business Process	Concept Impact
Customer Account	<a href="#">Create Order</a>	CRUD
	<a href="#">Bill Customer</a>	R
	<a href="#">Block Service for Non-Pay</a>	U

The entries in the Business Process column provide links to the use cases that are associated with the implementation of the processes. Using these links the analyst can identify use cases that are impacted by the change and that can be reused to describe the changed process.

To see what data stores trace to Customer Account the analyst can generate the following report:

Business Concept	Data Store
Customer Account	OE.CUST_BILLING_ACCT
	BILL.CUST_ACCT
	AR.CUSTOMER_ACCOUNT

The analyst can examine the models (physical and logical data models) of the data stores to determine the tables that can be reused, are impacted by the change, and that provide redundant information.

To determine what contracts and contract owners may be interested in changes to the affected data stores the following report is generated:

Business Concept	Data Store	Contract Impact	Contract	Contract Owner
Customer Account	OE.CUST_BILLING_ACCT	C	newOrder(...)	NON-OO Order-Entry System
		R	GetThisMonths Orders (...)	NON-OO Order-Entry System
		U	UpdateOrder(..)	NON-OO Order-Entry System
		...	...	...
		R	GetCustomerAc	EJB Entity Tier

			count ()	System
		U	SetCustomerAccount ()	EJB Entity Tier System
	BILL. CUST_ACCT	RU	...	Billing System
		D	...	NON-OO Order-Entry System
		CRUD	...	EJB Entity Tier System
	AR. CUSTOMER_ACCOUNT	CRUD	...	CORBA Entity Tier System

This report contains valuable information regarding the functions that directly access the data store. The report does not provide precise information on who calls these functions. The middle tier objects are obscuring the methods that are doing the interesting work. The following report on the interaction diagrams associated with the processes can provide such information:

Business Process	Interaction Diagram	Participating Contract	Contract Owner
<u>Create Order</u>	<u>Create Order</u>	CreateOrder(...)	Non_OO Order Entry System
		readyToBill(...)	Billing System
		validateCredit(...)	A/R System
		...	...
	<u>Create Web Order</u>	createSalesOrder(...)	Web
		readyToBill(...)	Billing System
			A/R System
<u>Bill Customer</u>	...	...	...
<u>Block Service for Non-Pay</u>	...	...	...

Only a representative set of interaction diagrams need be collected. Getting the analyst in the ballpark is sufficient in most situations.

The above reports do not give all the information needed to fully determine the potential impacts. More information could be derived in a dependency report that is filtered by items that may create, read, update or delete a Customer Account concept for a particular contract.

Concept	Contract	Dependency	Dependency CRUD
Customer Account	CreateOrder	ValidateOrder()	R
		SubmitOrder()	C
		...	...

The above reports provide a sample of the type of information that can be generated from a structured repository of system assets. In the next subsection we give an overview of the types of relationships that must exist between system artifacts in order to support the generation of such reports.

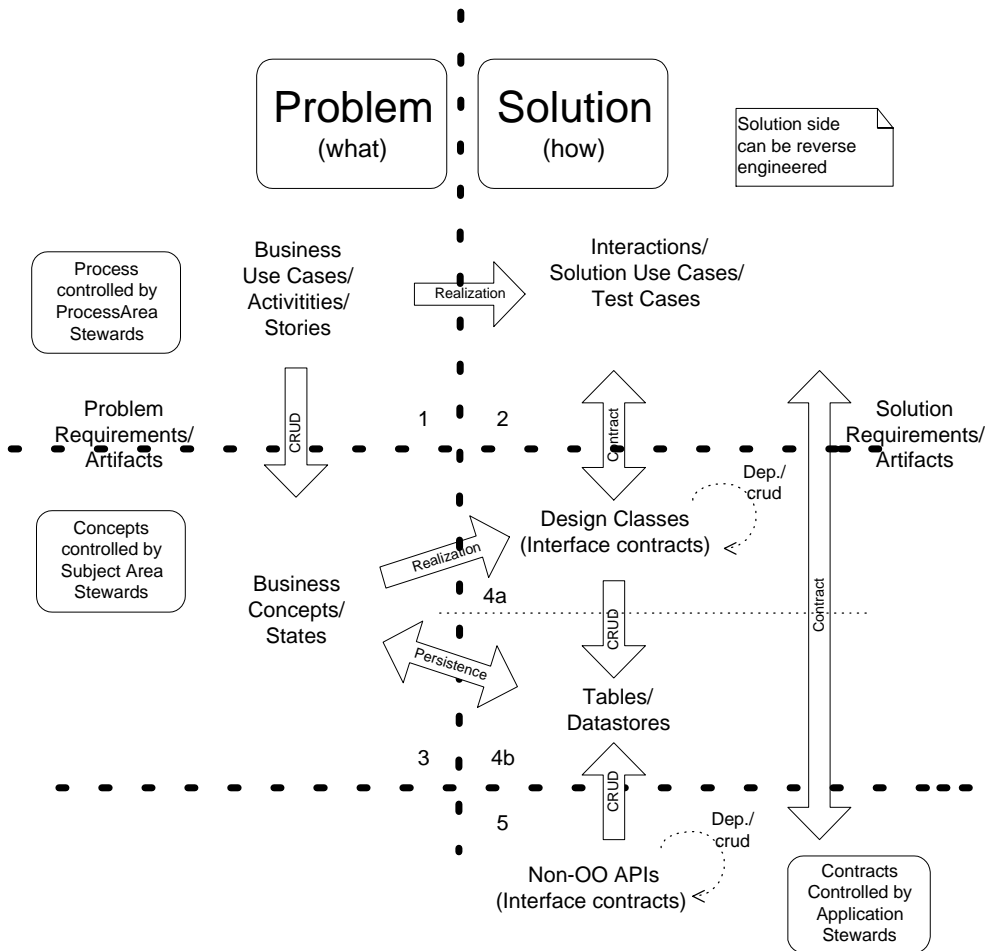
## 2.2. Traces Supporting RIGR Analyses

Figure 1 shows an overview of the types of artifact relationships that can support the generation of reports that facilitate RIGR analyses. Systems are represented by a set of system artifacts (model and implementation artifacts) that are broadly categorized as problem and solution artifacts in the repository. Problem artifacts (e.g., business concepts and processes) reflect the business perspective of the system asset. They describe the business processes and concepts in system-independent terms. The descriptions take the form of use cases, activity diagrams, state diagrams, and conceptual models (UML class models in which the classes represent concepts, not implementation classes). Solution artifacts describe the implementation of the concepts and processes in the business view. The descriptions take the form of UML interaction diagrams, component and deployment diagrams, and subsystem diagrams and design class diagrams.

The problem artifacts are further categorized as static (e.g., business concepts found in conceptual static structural models and states found in state diagrams) and dynamic (e.g., uses cases, activity diagrams describing processes). Processes are organized by process area and controlled by a steward committee comprised of users, developers and potential enterprise architects. Processes act on subjects (concepts). On the problem side subjects are reflected in conceptual models. Business concepts on the problem side may also have complex states that justify modeling.

# Reuse, Impact, Gap, and Redundancy Traces (RIGR)

This diagram shows traces that support reuse, impact, gap, and redundancy analysis.



Like processes, subjects are grouped by subject area and are controlled by a steward committee comprised of users, developers, and, potentially, enterprise architects. Process and subject areas may be hierarchical.

Processes act on subjects and the repository contains CRUD (Create, Read, Update, Delete) traces between processes and the concepts they impact (see CRUD arrow between grids 1 and 3 in Figure 1).

Interaction diagrams and solution use cases are realizations of business processes and trace links between the problem process artifacts and these diagrams are stored within the repository (as indicated by the Realization arrow between grids 1 and 2 in Figure 1). Design classes are solution artifacts that realize subjects (see Realization arrow between grids 3 and 4a). Business concepts may also persist in tables (see Persistence relationship between grids 3 and 4b). Business concepts may include reusable components (e.g., calendar or linked list).

Solution artifacts can also be Non-OO APIs (see grid 5 of Figure 1). Ideally, a thin OO layer would encapsulate these APIs, but in the real world this is not always the case. Fortunately, one can use UML static structures (e.g., interfaces, utilities) and interaction diagrams to refer to non-OO APIs.

Traces are the key to RIGR analysis. Only a subset of the traces shown in Figure 1 is required to receive some value. Starting with Business Concepts/States, that Business Use Cases/Activities may *create, read, update or delete* a business concept or the concept's state. We propose that this be documented using a stereotyped UML dependency called a *CRUD Trace*. A CRUD trace has the attribute flags (tag definitions) of Create, Read, Update and Delete.

The realization relationship between business concepts/states and design classes can be documented by a stereotyped UML realization dependency. This stereotype can also be to document the other realization relationship in Figure 1.

The persistence relationship between a business concept and a database table or other persistent data store is documented as a stereotyped dependency called a *Persistence trace*. This trace is critical when an organization stores the same entity in several redundant data store. Here the concept acts as an index to the redundant data stores.

Each interaction (particularly between systems and sub-systems) involves a *contract*. A contract can be supported by a design class or a non-OO API. Each design class may have *dependency* and *CRUD* traces to other design classes. The dependency trace is important in order to work down from a high level façade to a low level access to a data store and visa-versa. By qualifying the dependency with CRUD information, impacted artifacts can be pinpointed. Each non-OO API also has the same needs as design class for dependency/CRUD traces.

A design class may create, read, update or delete a table/data store. This is documented with a *CRUD* trace. This is documented with a *CRUD* Trace. The trace has the attribute flags of Create, Read, Update and Delete. Similarly, the table or data store may be created, read,

updated or deleted by a Non-OO API. In both cases, This is documented with a CRUD Trace. The CRUD trace has the attribute flags of Create, Read, Update and Delete.

The following traces are not explicitly shown on the diagram but worth mentioning:

- All elements should directly, or indirectly, trace to textual requirements (both problem and solution).
- All solution elements should trace back to their application system and data store.
- Each application system or data store should also have a steward or owner.
- There may be other types of artifacts or documents associated with any grid, system or project. At Qwest these will be tracked as URLs to the artifact.

### **2.3. Modeling Support**

To organize the repository, the problem space of the repository is tactically populated as an index into the solution space. Only the critical or reusable business concepts and business use cases are identified. Without models one has only the code as a source of documentation.

In our approach UML-based models are used to represent the non-code repository artifacts. The models and the information they capture are listed below:

- Problem Space Models
  - Conceptual model: This is a UML static structural model in which classes represent concepts. Concepts can have attributes (optional) but they have no operations associated with them. Concepts can have states associated with them (here, a state reflects the processes that have acted on the concept in the past). The concepts in the conceptual are the business concepts (subjects) that form the vocabulary of the business, thus they are good candidates for indexing the repository.
  - Business use case: A business use case describes a business process in terms of interactions between external agents (actors) and the entities responsible for performing the process. We developed a use case template that allows modelers to specify CRUD relationships between business use cases and the concepts in the conceptual model. Activity diagrams are used as an alternative representation of the process flows expressed textually in a use case. An advantage of using activity diagrams is their ability to show object flows. While some may argue that this violates the intent that use cases focus only on interactions between users and process executors, we have found it useful to document impacted objects (via object flows) in activity diagrams. This allows one to determine at an early stage the impact business processes have on existing system entities.
- Solution Space Models
  - Interaction diagrams: Collaboration diagrams or sequence diagrams are used to model realizations of business processes in terms of interactions

among subsystems, and, at a lower level, among objects. The subsystem and model constructs of the UML theoretically allow one to model the trace relationships between the business use cases and the interaction diagrams, though we have found very poor tool support for this feature.

- Design structural models: Subsystem diagrams with contracts, and lower-level design class diagrams are used to capture the macro- and micro-architectures of solutions. One can view a contract as a stereotyped interface construct. We have developed a contract template that supports CRUD traceability to solution artifacts (e.g., database tables, solution classes).

### 3. Implementing the Repository: Issues

The following points should be stressed about implementing our proposed traces in a repository:

- Obtaining, entering, and maintaining an exhaustive set of artifacts and tracing information can be an expensive, time-consuming, error-prone endeavor if done manually. In the absence of automated passive collection of information it may be more cost-effective to maintain information on only shared assets. Granularity of tracing can be scaled to what provides value (e.g. a concept may have an expensive CRUD trace to low-level design classes or a less expensive, less granular CRUD trace to the application system).
- Problem artifacts (in particular, the conceptual model) provide the concepts needed to effectively index a repository. They reflect the business concepts that are familiar to users and developers in the domain, and are relatively stable.
- The solution artifacts can be reversed engineered from code if not present, and then cataloged by the concepts in the problem space.

Below we outline some of the issues that arise when implementing the repository to support effective RIGR analyses.

#### 3.1. Artifact Versioning

Ideally, the repository artifacts are versioned by project and application with “As Is” (reflecting current information) and “To Be” (reflecting information on future systems) tags. Changes to a shared subject or business process should be controlled by a single entity. We propose that steward committees consisting of developers and/or possibly enterprise architects be charged with controlling such changes. The stewards could receive automated notices of potential impacts from the repository whenever a request to make a change is indicated in the repository.

Out of context, Adds little value. I know what your referring to, but not sure this explains it well. Just delete for simplicity.

### **3.2. Repository Population**

Repository population can be optional and still be effective. Analyst and developers seem to instinctively know that similar efforts exist somewhere else. The problem is they can never seem to find them (ironically, it's often something they did themselves a few years ago). Once the repository container is built, people will want to use it to store all their work in.

To be truly effective, the population of the repository should be built into the process. How and when this is done determines what is collected. At one extreme, a "shoestring" budget virtual repository can be a database that collects and categorizes website pointers (URLs). The URLs point to sites generated by web reports generated by various CASE tools. They sites are maintained and versioned by their creators in their work areas.

A more complex repository would be a fully integrated environment containing "to be" and "as is" models and code. In this system, a central repository would act as glue for open, best-of-breed tools. Repository use would be a seamless part of the development effort. Anyone in the company could go to a single place to perform RIGR analysis and pinpoint the relevant artifacts without having to install any software on their computer. For updates, models would be checked out, worked on, validated against collisions with other "to be" models, and then checked back in for release.

### **3.3. Tool Support**

Unfortunately, no one tool seems to do it all out of the box. One of the motivations of this document is to illicit tool recommendations. An integrated set of tools consisting of a UML modeling environment (supporting modeling of data, objects, and processes), code analyzers (to facilitate reverse engineering and impact analyses), version control, and requirements management is needed to fully support our repository approach.

## **4. Conclusions**

In this paper we have outlined how a well-structured repository of UML models and code artifacts to support RIGR analyses. We are considering implementing these ideas within Qwest. The following are the major challenges that we foresee:

- Providing effective tool support for RIGR analyses: No one tool will deliver the full functionality needed. In particular, tools that facilitate automated passive collection of artifacts and trace information are needed. The challenge is to identify and integrate a set of tools that together can provide the needed functionality.
- Understanding relationships among UML models: Precise definition of trace and other dependency relationships requires a clear understanding of the relationships among UML modeling concepts. The current UML document defines the semantic aspects of concepts informally thus it is difficult to precisely determine such relationships. We have developed interpretations of the UML models that are of interest to Qwest. This allows us to specify these relationships precisely.

- Using models in agile processes: Within industry increasing attention is being turned to agile processes (“lightweight” processes that facilitate development of systems in “internet” time). There is a tendency to use agile processes as an excuse not to model. In fact, as systems become more complex the need to model becomes more apparent. A challenge is to define modeling techniques that facilitate rapid development. Such techniques should allow a developer to produce only those models that produce value to the project. All artifacts are candidates for informal modeling. Candidate artifacts for more rigorous modeling include strategic architectures, stable software structures, shareable and reusable modules.

### **Bibliography**

- [Erik2000] Hans-Erik Eriksson, Magnus Penker, *Business Modeling with UML: Business Patterns at Work*, Wiley, OMG Press, 2000.
- [Scheer2000] August-Wilhelm Scheer, Frank Habermann, Making ERP A Success, *Communications of the ACM*, 43(4), 2000, pp. 57-61.
- [UML99] OMG, Unified Modeling Language Standard, Version 1.3, Object Management Group, <http://www.omg.org>, June 1999.